

FLTK 1.3.0 Programming Manual



Revision 9 by F. Costantini, D. Gibson, M. Melcher,
A. Schlosser, B. Spitzak and M. Sweet.

Copyright 1998-2009 by Bill Spitzak and others.

Generated by Doxygen 1.5.7.1

March 23, 2009

Contents

1	FLTK Programming Manual	1
2	Preface	3
2.1	Organization	4
2.2	Conventions	5
2.3	Abbreviations	5
2.4	Copyrights and Trademarks	5
3	1 - Introduction to FLTK	7
3.1	History of FLTK	8
3.2	Features	8
3.3	Licensing	9
3.4	What Does "FLTK" Mean?	9
3.5	Building and Installing FLTK Under UNIX and MacOS X	10
3.6	Building FLTK Under Microsoft Windows	11
3.7	Building FLTK Under OS/2	12
3.8	Internet Resources	12
3.9	Reporting Bugs	13
4	2 - FLTK Basics	15
4.1	Writing Your First FLTK Program	16
4.2	Compiling Programs with Standard Compilers	18
4.3	Compiling Programs with Makefiles	19
4.4	Compiling Programs with Microsoft Visual C++	20
4.5	Naming	20
4.6	Header Files	20
5	3 - Common Widgets and Attributes	23
5.1	Buttons	24
5.2	Text	25

5.3	Valuators	25
5.4	Groups	26
5.5	Setting the Size and Position of Widgets	27
5.6	Colors	27
5.7	Box Types	28
5.8	Labels and Label Types	29
5.9	Callbacks	32
5.10	Shortcuts	33
6	5 - Drawing Things in FLTK	35
6.1	When Can You Draw Things in FLTK?	36
6.2	Drawing Functions	36
6.3	Colors	38
6.4	Drawing Images	47
7	4 - Designing a Simple Text Editor	51
7.1	Determining the Goals of the Text Editor	52
7.2	Designing the Main Window	52
7.3	Variables	52
7.4	Menubars and Menus	53
7.5	Editing the Text	53
7.6	The Replace Dialog	54
7.7	Callbacks	54
7.8	Other Functions	59
7.9	The main() Function	60
7.10	Compiling the Editor	60
7.11	The Final Product	61
7.12	Advanced Features	61
8	6 - Handling Events	67
8.1	The FLTK Event Model	68
8.2	Mouse Events	68
8.3	Focus Events	69
8.4	Keyboard Events	69
8.5	Widget Events	70
8.6	Clipboard Events	70
8.7	Drag and Drop Events	71
8.8	Fl::event_*() methods	71

8.9	Event Propagation	72
8.10	FLTK Compose-Character Sequences	73
9	7 - Adding and Extending Widgets	75
9.1	Subclassing	76
9.2	Making a Subclass of Fl_Widget	76
9.3	The Constructor	76
9.4	Protected Methods of Fl_Widget	77
9.5	Handling Events	79
9.6	Drawing the Widget	80
9.7	Resizing the Widget	81
9.8	Making a Composite Widget	81
9.9	Cut and Paste Support	83
9.10	Drag And Drop Support	83
9.11	Making a subclass of Fl_Window	83
10	8 - Using OpenGL	85
10.1	Using OpenGL in FLTK	86
10.2	Making a Subclass of Fl_Gl_Window	86
10.3	Using OpenGL in Normal FLTK Windows	88
10.4	OpenGL Drawing Functions	89
10.5	Speeding up OpenGL	90
10.6	Using OpenGL Optimizer with FLTK	90
11	9 - Programming with FLUID	93
11.1	What is FLUID?	94
11.2	Running FLUID Under UNIX	96
11.3	Running FLUID Under Microsoft Windows	96
11.4	Compiling .fl files	96
11.5	A Short Tutorial	97
11.6	FLUID Reference	104
11.7	GUI Attributes	111
11.8	Selecting and Moving Widgets	117
11.9	Image Labels	118
11.10	Internationalization with FLUID	119
11.11	Known limitations	121
12	10 - Advanced FLTK	123

12.1 Multithreading	124
13 11 - Unicode and utf-8 Support	127
13.1 About Unicode and utf-8	128
13.2 Unicode in FLTK	128
14 C - FLTK Enumerations	131
14.1 Version Numbers	132
14.2 Events	132
14.3 Callback "When" Conditions	133
14.4 Fl::event_button() Values	133
14.5 Fl::event_key() Values	134
14.6 Fl::event_state() Values	135
14.7 Alignment Values	135
14.8 Fonts	136
14.9 Colors	136
14.10Cursors	138
14.11FD "When" Conditions	138
14.12Damage Masks	138
15 D - GLUT Compatibility	139
15.1 Using the GLUT Compatibility Header File	140
15.2 Known Problems	140
15.3 Mixing GLUT and FLTK Code	141
15.4 class Fl_Glut_Window	141
16 E - Forms Compatibility	145
16.1 Importing Forms Layout Files	146
16.2 Using the Compatibility Header File	146
16.3 Problems You Will Encounter	146
16.4 Additional Notes	148
17 F - Operating System Issues	151
17.1 Accessing the OS Interfaces	152
17.2 The UNIX (X11) Interface	152
17.3 The Windows (WIN32) Interface	157
17.4 The MacOS Interface	159
18 G - Migrating Code from FLTK 1.0 to 1.1	163

18.1 Color Values	164
18.2 Cut and Paste Support	164
18.3 File Chooser	164
18.4 Function Names	164
18.5 Image Support	165
18.6 Keyboard Navigation	165
19 H - Migrating Code from FLTK 1.1 to 1.3	167
19.1 Migrating From FLTK 1.0	168
19.2 Fl_Scroll Widget	168
19.3 Unicode (utf-8)	168
19.4 Widget Coordinate Representation	168
20 I - Developer Information	169
20.1 Non-ASCII characters	172
20.2 Document Structure	173
20.3 Creating Links	173
20.4 Changing Old Links	174
20.5 Paragraph Layout	175
20.6 Hack for missing "tiny.gif" file	176
20.7 Navigation Elements	176
21 J - Software License	177
22 K - Example Source Code	185
22.1 Example Applications	186
23 Deprecated List	197
24 Todo List	199
25 Module Index	203
25.1 Modules	203
26 Class Index	205
26.1 Class Hierarchy	205
27 Class Index	209
27.1 Class List	209
28 File Index	213

28.1 File List	213
29 Module Documentation	219
29.1 Windows handling functions	219
29.2 Events handling functions	222
29.3 Selection & Clipboard functions	232
29.4 Screen functions	235
29.5 Color & Font functions	237
29.6 Drawing functions	246
29.7 Multithreading support functions	265
29.8 Safe widget deletion support functions	267
29.9 Cairo support functions and classes	270
29.10 Common Dialogs classes and functions	272
29.11 File names and URI utility functions	279
30 Class Documentation	283
30.1 Fl Class Reference	283
30.2 Fl_Adjuster Class Reference	305
30.3 Fl_Bitmap Class Reference	308
30.4 Fl_BMP_Image Class Reference	311
30.5 Fl_Box Class Reference	312
30.6 Fl_Browser Class Reference	314
30.7 Fl_Browser_ Class Reference	332
30.8 Fl_Button Class Reference	342
30.9 Fl_Cairo_State Class Reference	347
30.10 Fl_Cairo_Window Class Reference	348
30.11 Fl_Chart Class Reference	350
30.12 FL_CHART_ENTRY Struct Reference	356
30.13 Fl_Check_Browser Class Reference	357
30.14 Fl_Check_Button Class Reference	361
30.15 Fl_Choice Class Reference	363
30.16 Fl_Clock Class Reference	367
30.17 Fl_Clock_Output Class Reference	370
30.18 Fl_Color_Chooser Class Reference	374
30.19 Fl_Counter Class Reference	379
30.20 Fl_Dial Class Reference	383
30.21 Fl_Double_Window Class Reference	386

30.22Fl_End Class Reference	389
30.23Fl_File_Browser Class Reference	390
30.24Fl_File_Chooser Class Reference	393
30.25Fl_File_Icon Class Reference	402
30.26Fl_File_Input Class Reference	408
30.27Fl_Fill_Dial Class Reference	411
30.28Fl_Fill_Slider Class Reference	412
30.29Fl_Float_Input Class Reference	413
30.30Fl_Font_Descriptor Class Reference	414
30.31Fl_FormsBitmap Class Reference	415
30.32Fl_FormsPixmap Class Reference	417
30.33Fl_Free Class Reference	419
30.34Fl_GIF_Image Class Reference	422
30.35Fl_GL_Window Class Reference	423
30.36Fl_Glut_Bitmap_Font Struct Reference	430
30.37Fl_Glut_Window Class Reference	431
30.38Fl_Group Class Reference	434
30.39Fl_Help_Dialog Class Reference	444
30.40Fl_Help_Font_Style Struct Reference	448
30.41Fl_Help_Link Struct Reference	449
30.42Fl_Help_Target Struct Reference	450
30.43Fl_Help_View Class Reference	451
30.44Fl_Hold_Browser Class Reference	456
30.45Fl_Image Class Reference	457
30.46Fl_Input Class Reference	462
30.47Fl_Input_ Class Reference	466
30.48Fl_Input_Choice Class Reference	475
30.49Fl_Int_Input Class Reference	479
30.50Fl_JPEG_Image Class Reference	480
30.51Fl_Label Struct Reference	481
30.52Fl_Light_Button Class Reference	483
30.53Fl_Menu_ Class Reference	486
30.54Fl_Menu_Bar Class Reference	494
30.55Fl_Menu_Button Class Reference	497
30.56Fl_Menu_Item Struct Reference	500
30.57Fl_Menu_Window Class Reference	512

30.58Fl_Multi_Browser Class Reference	515
30.59Fl_Multiline_Input Class Reference	516
30.60Fl_Multiline_Output Class Reference	517
30.61Fl_Output Class Reference	518
30.62Fl_Overlay_Window Class Reference	520
30.63Fl_Pack Class Reference	523
30.64Fl_Pixmap Class Reference	525
30.65Fl_PNG_Image Class Reference	529
30.66Fl_PNM_Image Class Reference	530
30.67Fl_Positioner Class Reference	531
30.68Fl_Preferences Class Reference	535
30.69Fl_Preferences::Entry Struct Reference	547
30.70Fl_Preferences::Name Class Reference	548
30.71Fl_Progress Class Reference	550
30.72Fl_Repeat_Button Class Reference	552
30.73Fl_Return_Button Class Reference	554
30.74Fl_RGB_Image Class Reference	557
30.75Fl_Roller Class Reference	561
30.76Fl_Round_Button Class Reference	564
30.77Fl_Round_Clock Class Reference	566
30.78Fl_Scroll Class Reference	567
30.79Fl_Scrollbar Class Reference	572
30.80Fl_Secret_Input Class Reference	576
30.81Fl_Select_Browser Class Reference	577
30.82Fl_Shared_Image Class Reference	578
30.83Fl_Simple_Counter Class Reference	583
30.84Fl_Single_Window Class Reference	584
30.85Fl_Slider Class Reference	586
30.86Fl_Spinner Class Reference	590
30.87Fl_Tabs Class Reference	595
30.88Fl_Text_Buffer Class Reference	599
30.89Fl_Text_Display Class Reference	616
30.90Fl_Text_Display::Style_Table_Entry Struct Reference	633
30.91Fl_Text_Editor Class Reference	634
30.92Fl_Text_Editor::Key_Binding Struct Reference	641
30.93Fl_Text_Selection Class Reference	642

30.94Fl_Tile Class Reference	644
30.95Fl_Tiled_Image Class Reference	647
30.96Fl_Timer Class Reference	650
30.97Fl_Toggle_Button Class Reference	653
30.98Fl_Tooltip Class Reference	654
30.99Fl_Valuator Class Reference	658
30.100Fl_Value_Input Class Reference	664
30.101Fl_Value_Output Class Reference	670
30.102Fl_Value_Slider Class Reference	674
30.103Fl_Widget Class Reference	677
30.104Fl_Widget_Tracker Class Reference	710
30.105Fl_Window Class Reference	712
30.106Fl_Wizard Class Reference	723
30.107Fl_XBM_Image Class Reference	725
30.108Fl_XPM_Image Class Reference	726
31 File Documentation	727
31.1 Enumerations.H File Reference	727
31.2 fl_arc.cxx File Reference	741
31.3 fl_arci.cxx File Reference	742
31.4 fl_boxtype.cxx File Reference	743
31.5 fl_color.cxx File Reference	745
31.6 Fl_Color_Chooser.H File Reference	747
31.7 fl_curve.cxx File Reference	748
31.8 fl_draw.H File Reference	749
31.9 fl_line_style.cxx File Reference	757
31.10fl_rect.cxx File Reference	758
31.11fl_types.h File Reference	761
31.12fl_vertex.cxx File Reference	762

Chapter 1

FLTK Programming Manual



FLTK 1.3.0 Programming Manual

Revision 9 by F. Costantini, D. Gibson, M.
Melcher,
A. Schlosser, B. Spitzak and M. Sweet.
Copyright 1998-2009 by Bill Spitzak and others.

This software and manual are provided under the terms of the GNU Library General Public License. Permission is granted to reproduce this manual or any portion for any purpose, provided this copyright and permission notice are preserved.

<p>Preface</p> <p>1 - Introduction to FLTK</p> <p>2 - FLTK Basics</p> <p>3 - Common Widgets and Attributes</p> <ul style="list-style-type: none">• Colors• Box Types• Labels and Label Types• Drawing Images <p>4 - Designing a Simple Text Editor</p> <p>5 - Drawing Things in FLTK</p> <p>6 - Handling Events</p> <ul style="list-style-type: none">• Fl::event_*() methods• Event Propagation <p>7 - Adding and Extending Widgets</p> <p>8 - Using OpenGL</p> <p>9 - Programming with FLUID</p> <ul style="list-style-type: none">• GUI Attributes• Selecting and Moving Widgets• Image Labels	<p>10 - Advanced FLTK</p> <p>11 - Unicode and utf-8 Support</p> <p>A - Class Reference</p> <p>B - Function Reference</p> <p>C - FLTK Enumerations</p> <p>D - GLUT Compatibility</p> <ul style="list-style-type: none">• class Fl_Glut_Window <p>E - Forms Compatibility</p> <p>F - Operating System Issues</p> <p>G - Migrating Code from FLTK 1.0 to 1.1</p> <p>H - Migrating Code from FLTK 1.1 to 1.3</p> <p>I - Developer Information</p> <p>J - Software License</p> <p>K - Example Source Code</p>
---	--

Chapter 2

Preface

This manual describes the Fast Light Tool Kit ("FLTK") version 1.3.0, a C++ Graphical User Interface ("GUI") toolkit for UNIX, Microsoft Windows and MacOS.

Each of the chapters in this manual is designed as a tutorial for using FLTK, while the appendices provide a convenient reference for all FLTK widgets, functions, and operating system interfaces.

This manual may be printed, modified, and/or used under the terms of the FLTK license provided in [J - Software License](#).

2.1 Organization

This manual is organized into the following chapters and appendices:

- [1 - Introduction to FLTK](#)
- [2 - FLTK Basics](#)
- [3 - Common Widgets and Attributes](#)
- [4 - Designing a Simple Text Editor](#)
- [5 - Drawing Things in FLTK](#)
- [6 - Handling Events](#)
- [7 - Adding and Extending Widgets](#)
- [8 - Using OpenGL](#)
- [9 - Programming with FLUID](#)
- [10 - Advanced FLTK](#)
- [11 - Unicode and utf-8 Support](#)
- [A - Class Reference](#)
- [B - Function Reference](#)
- [C - FLTK Enumerations](#)
- [D - GLUT Compatibility](#)
- [E - Forms Compatibility](#)
- [F - Operating System Issues](#)
- [G - Migrating Code from FLTK 1.0 to 1.1](#)
- [H - Migrating Code from FLTK 1.1 to 1.3](#)
- [I - Developer Information](#)
- [J - Software License](#)
- [K - Example Source Code](#)

2.2 Conventions

The following typeface conventions are used in this manual:

- Function and constant names are shown in **bold courier type**
- Code samples and commands are shown in `regular courier type`

2.3 Abbreviations

The following abbreviations are used in this manual:

X11

The X Window System version 11.

Xlib

The X Window System interface library.

WIN32

The Microsoft Windows 32-bit Application Programmer's Interface.

MacOS

The Apple Macintosh OS 8.6 and later, including OS X.

2.4 Copyrights and Trademarks

FLTK is Copyright 1998-2009 by Bill Spitzak and others. Use and distribution of FLTK is governed by the GNU Library General Public License with 4 exceptions, located in [J - Software License](#).

UNIX is a registered trademark of the X Open Group, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation. OpenGL is a registered trademark of Silicon Graphics, Inc. Apple, Macintosh, MacOS, and Mac OS X are registered trademarks of Apple Computer, Inc.

Chapter 3

1 - Introduction to FLTK

The Fast Light Tool Kit ("FLTK", pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX®/Linux®(X11), Microsoft®Windows®, and MacOS®X.

FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL®and its built-in GLUT emulation. It was originally developed by Mr. Bill Spitzak and is currently maintained by a small group of developers across the world with a central repository in the US.

3.1 History of FLTK

It has always been Bill's belief that the GUI API of all modern systems is much too high level. Toolkits (even FLTK) are *not* what should be provided and documented as part of an operating system. The system only has to provide arbitrary shaped but featureless windows, a powerful set of graphics drawing calls, and a simple *unalterable* method of delivering events to the owners of the windows. NeXT (if you ignored NextStep) provided this, but they chose to hide it and tried to push their own baroque toolkit instead.

Many of the ideas in FLTK were developed on a NeXT (but *not* using NextStep) in 1987 in a C toolkit Bill called "views". Here he came up with passing events downward in the tree and having the handle routine return a value indicating whether it used the event, and the table-driven menus. In general he was trying to prove that complex UI ideas could be entirely implemented in a user space toolkit, with no knowledge or support by the system.

After going to film school for a few years, Bill worked at Sun Microsystems on the (doomed) NeWS project. Here he found an even better and cleaner windowing system, and he reimplemented "views" atop that. NeWS did have an unnecessarily complex method of delivering events which hurt it. But the designers did admit that perhaps the user could write just as good of a button as they could, and officially exposed the lower level interface.

With the death of NeWS Bill realized that he would have to live with X. The biggest problem with X is the "window manager", which means that the toolkit can no longer control the window borders or drag the window around.

At Digital Domain Bill discovered another toolkit, "Forms". Forms was similar to his work, but provided many more widgets, since it was used in many real applications, rather than as theoretical work. He decided to use Forms, except he integrated his table-driven menus into it. Several very large programs were created using this version of Forms.

The need to switch to OpenGL and GLX, portability, and a desire to use C++ subclassing required a rewrite of Forms. This produced the first version of FLTK. The conversion to C++ required so many changes it made it impossible to recompile any Forms objects. Since it was incompatible anyway, Bill decided to incorporate his older ideas as much as possible by simplifying the lower level interface and the event passing mechanism.

Bill received permission to release it for free on the Internet, with the GNU general public license. Response from Internet users indicated that the Linux market dwarfed the SGI and high-speed GL market, so he rewrote it to use X for all drawing, greatly speeding it up on these machines. That is the version you have now.

Digital Domain has since withdrawn support for FLTK. While Bill is no longer able to actively develop it, he still contributes to FLTK in his free time and is a part of the FLTK development team.

3.2 Features

FLTK was designed to be statically linked. This was done by splitting it into many small objects and designing it so that functions that are not used do not have pointers to them in the parts that are used, and thus do not get linked in. This allows you to make an easy-to-install program or to modify FLTK to the

exact requirements of your application without worrying about bloat. FLTK works fine as a shared library, though, and is now included with several Linux distributions.

Here are some of the core features unique to FLTK:

- `sizeof(Fl_Widget) == 64` to 92.
- The "core" (the "hello" program compiled & linked with a static FLTK library using gcc on a 486 and then stripped) is 114K.
- The FLUID program (which includes every widget) is 538k.
- Written directly atop core libraries (Xlib, WIN32 or Carbon) for maximum speed, and carefully optimized for code size and performance.
- Precise low-level compatibility between the X11, WIN32 and MacOS versions - only about 10% of the code is different.
- Interactive user interface builder program. Output is human-readable and editable C++ source code.
- Support for overlay hardware, with emulation if none is available.
- Very small & fast portable 2-D drawing library to hide Xlib, WIN32, or QuickDraw.
- OpenGL/Mesa drawing area widget.
- Support for OpenGL overlay hardware on both X11 and WIN32, with emulation if none is available.
- Text widgets with Emacs key bindings, X cut & paste, and foreign letter compose!
- Compatibility header file for the GLUT library.
- Compatibility header file for the XForms library.

3.3 Licensing

FLTK comes with complete free source code. FLTK is available under the terms of the [GNU Library General Public License](#) with exceptions that allow for static linking. Contrary to popular belief, it can be used in commercial software - even Bill Gates could use it!

3.4 What Does "FLTK" Mean?

FLTK was originally designed to be compatible with the Forms Library written for SGI machines. In that library all the functions and structures started with "fl_". This naming was extended to all new methods and widgets in the C++ library, and this prefix was taken as the name of the library. It is almost impossible to search for "FL" on the Internet, due to the fact that it is also the abbreviation for Florida. After much debating and searching for a new name for the toolkit, which was already in use by several people, Bill came up with "FLTK", including a bogus excuse that it stands for "The Fast Light Toolkit".

3.5 Building and Installing FLTK Under UNIX and MacOS X

In most cases you can just type "make". This will run configure with the default of no options and then compile everything.

FLTK uses GNU autoconf to configure itself for your UNIX platform. The main things that the configure script will look for are the X11 and OpenGL (or Mesa) header and library files. If these cannot be found in the standard include/library locations you'll need to define the CFLAGS, CXXFLAGS, and LDFLAGS environment variables. For the Bourne and Korn shells you'd use:

```
CFLAGS=-Iincludedir; export CFLAGS
CXXFLAGS=-Iincludedir; export CXXFLAGS
LDFLAGS=-Llibdir; export LDFLAGS
```

For C shell and tcsh, use:

```
setenv CFLAGS "-Iincludedir"
setenv CXXFLAGS "-Iincludedir"
setenv LDFLAGS "-Llibdir"
```

By default configure will look for a C++ compiler named CC, c++, g++, or gcc in that order. To use another compiler you need to set the CXX environment variable:

```
CXX=x1C; export CXX
setenv CXX "x1C"
```

The CC environment variable can also be used to override the default C compiler (cc or gcc), which is used for a few FLTK source files.

You can run configure yourself to get the exact setup you need. Type "./configure <options>", where options are:

-enable-cygwin

Enable the Cygwin libraries under WIN32

-enable-debug

Enable debugging code & symbols

-disable-gl

Disable OpenGL support

-enable-shared

Enable generation of shared libraries

-enable-threads

Enable multithreading support

-enable-xdbe

Enable the X double-buffer extension

-enable-xft

Enable the Xft library for anti-aliased fonts under X11

-enable-x11

When targeting cygwin, build with X11 GUI instead of windows GDI

-bindir=/path

Set the location for executables [default = \$prefix/bin]

-datadir=/path

Set the location for data files. [default = \$prefix/share]

-libdir=/path

Set the location for libraries [default = \$prefix/lib]

-includedir=/path

Set the location for include files. [default = \$prefix/include]

-mandir=/path

Set the location for man pages. [default = \$prefix/man]

-prefix=/dir

Set the directory prefix for files [default = /usr/local]

When the configure script is done you can just run the "make" command. This will build the library, FLUID tool, and all of the test programs.

To install the library, become root and type "make install". This will copy the "fluid" executable to "bindir", the header files to "includedir", and the library files to "libdir".

3.6 Building FLTK Under Microsoft Windows

There are three ways to build FLTK under Microsoft Windows. The first is to use the Visual C++ 5.0 project files under the "visualc" directory. Just open (or double-click on) the "fltk.dsw" file to get the whole shebang.

The second method is to use the `configure` script included with the FLTK software; this has only been tested with the Cygwin tools:

```
sh configure --prefix=C:/FLTK
make
```

The final method is to use a GNU-based development tool with the files in the "makefiles" directory. To build using one of these tools simply copy the appropriate makeinclude and config files to the main directory and do a make:

```
copy makefiles\Makefile.<env> Makefile
make
```

3.6.1 Using the Visual C++ DLL Library

The "ftkdll.dsp" project file builds a DLL-version of the FLTK library. Because of name mangling differences between PC compilers (even between different versions of Visual C++!) you can only use the DLL that is generated with the same version compiler that you built it with.

When compiling an application or DLL that uses the FLTK DLL, you will need to define the `FL_DLL` preprocessor symbol to get the correct linkage commands embedded within the FLTK header files.

3.7 Building FLTK Under OS/2

The current OS/2 build requires XFree86 for OS/2 to work. A native Presentation Manager version has not been implemented yet (volunteers are welcome!).

The current set of Makefiles/configuration files assumes that EMX 0.9d and libExt (from posix2.sourceforge.net) is installed.

To build the XFree86 version of FLTK for OS/2, copy the appropriate makeinclude and config files to the main directory and do a make:

```
copy makefiles\Makefile.os2x Makefile
make
```

3.8 Internet Resources

FLTK is available on the 'net in a bunch of locations:

WWW

```
http://www.fltk.org/
http://www.fltk.org/str.php [for reporting bugs]
http://www.fltk.org/software.php [source code]
```

FTP

```
California, USA (ftp.fltk.org)
Maryland, USA (ftp2.fltk.org)
Espoo, Finland (ftp.funet.fi)
Germany (linux.mathematik.tu-darmstadt.de)
Austria (gd.tuwien.ac.at)
```

Email

```
fltk@fltk.org [see instructions below]
fltk-bugs@fltk.org [for reporting bugs]
```

NNTP Newsgroups

```
news.easysw.com
```

To send a message to the FLTK mailing list ("fltk@fltk.org") you must first join the list. Non-member submissions are blocked to avoid problems with unsolicited email.

To join the FLTK mailing list, send a message to "majordomo@fltk.org" with "subscribe fltk" in the message body. A digest of this list is available by subscribing to the "fltk-digest" mailing list.

3.9 Reporting Bugs

To report a bug in FLTK, send an email to "ftk-bugs@ftk.org". Please include the FLTK version, operating system & version, and compiler that you are using when describing the bug or problem. We will be unable to provide any kind of help without that basic information.

Bugs can also be reported to the "ftk.bugs" newsgroup or on the SourceForge bug tracker pages.

For general support and questions, please use the FLTK mailing list at "ftk@ftk.org" or one of the newsgroups.

Chapter 4

2 - FLTK Basics

This chapter teaches you the basics of compiling programs that use FLTK.

4.1 Writing Your First FLTK Program

All programs must include the file `<FL/Fl.H>`. In addition the program must include a header file for each FLTK class it uses. Listing 1 shows a simple "Hello, World!" program that uses FLTK to display the window.

Listing 1 - "hello.cxx"

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
    Fl_Window *window = new Fl_Window(300,180);
    Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
    box->box(FL_UP_BOX);
    box->labelsize(36);
    box->labelfont(FL_BOLD+FL_ITALIC);
    box->labeltype(FL_SHADOW_LABEL);
    window->end();
    window->show(argc, argv);
    return Fl::run();
}
```

After including the required header files, the program then creates a window. All following widgets will automatically be children of this window.

```
Fl_Window *window = new Fl_Window(300,180);
```

Then we create a box with the "Hello, World!" string in it. FLTK automatically adds the new box to window, the current grouping widget.

```
Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
```

Next, we set the type of box and the size, font, and style of the label:

```
box->box(FL_UP_BOX);
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
```

We tell FLTK that we will not add any more widgets to window.

```
window->end();
```

Finally, we show the window and enter the FLTK event loop:

```
window->show(argc, argv);
return Fl::run();
```

The resulting program will display the window in Figure 2-1. You can quit the program by closing the window or pressing the ESCape key.

Figure 4.1: The Hello, World! Window

4.1.1 Creating the Widgets

The widgets are created using the C++ `new` operator. For most widgets the arguments to the constructor are:

```
Fl_Widget(x, y, width, height, label)
```

The `x` and `y` parameters determine where the widget or window is placed on the screen. In FLTK the top left corner of the window or screen is the origin (i.e. `x = 0`, `y = 0`) and the units are in pixels.

The `width` and `height` parameters determine the size of the widget or window in pixels. The maximum widget size is typically governed by the underlying window system or hardware.

`label` is a pointer to a character string to label the widget with or `NULL`. If not specified the label defaults to `NULL`. The label string must be in static storage such as a string constant because FLTK does not make a copy of it - it just uses the pointer.

4.1.2 Creating Widget hierarchies

Widgets are commonly ordered into functional groups, which in turn may be grouped again, creating a hierarchy of widgets. FLTK makes it easy to fill groups by automatically adding all widgets that are created between a `myGroup->begin()` and `myGroup->end()`. In this example, `myGroup` would be the *current* group.

Newly created groups and their derived widgets implicitly call `begin()` in the constructor, effectively adding all subsequently created widgets to itself until `end()` is called.

Setting the current group to `NULL` will stop automatic hierarchies. New widgets can now be added manually using `Fl_Group::add(...)` and `Fl_Group::insert(...)`.

4.1.3 Get/Set Methods

`box->box(FL_UP_BOX)` sets the type of box the `Fl_Box` draws, changing it from the default of `FL_NO_BOX`, which means that no box is drawn. In our "Hello, World!" example we use `FL_UP_BOX`, which means that a raised button border will be drawn around the widget. You can learn more about boxtypes in [Chapter 3](#).

You could examine the boxtype in by doing `box->box()`. FLTK uses method name overloading to make short names for get/set methods. A "set" method is always of the form "void name(type)", and a "get" method is always of the form "type name() const".

4.1.4 Redrawing After Changing Attributes

Almost all of the set/get pairs are very fast, short inline functions and thus very efficient. However, *the "set" methods do not call `redraw()`* - you have to call it yourself. This greatly reduces code size and execution time. The only common exceptions are `value()` which calls `redraw()` and `label()` which calls `redraw_label()` if necessary.

4.1.5 Labels

All widgets support labels. In the case of window widgets, the label is used for the label in the title bar. Our example program calls the `labelfont()`, `labelsize()`, and `labeltype()` methods.

The `labelfont` method sets the typeface and style that is used for the label, which for this example we are using `FL_BOLD` and `FL_ITALIC`. You can also specify typefaces directly.

The `labelsize` method sets the height of the font in pixels.

The `labeltype` method sets the type of label. FLTK supports normal, embossed, and shadowed labels internally, and more types can be added as desired.

A complete list of all label options can be found in [Chapter 3](#).

4.1.6 Showing the Window

The `show()` method shows the widget or window. For windows you can also provide the command-line arguments to allow users to customize the appearance, size, and position of your windows.

4.1.7 The Main Event Loop

All FLTK applications (and most GUI applications in general) are based on a simple event processing model. User actions such as mouse movement, button clicks, and keyboard activity generate events that are sent to an application. The application may then ignore the events or respond to the user, typically by redrawing a button in the "down" position, adding the text to an input field, and so forth.

FLTK also supports idle, timer, and file pseudo-events that cause a function to be called when they occur. Idle functions are called when no user input is present and no timers or files need to be handled - in short, when the application is not doing anything. Idle callbacks are often used to update a 3D display or do other background processing.

Timer functions are called after a specific amount of time has expired. They can be used to pop up a progress dialog after a certain amount of time or do other things that need to happen at more-or-less regular intervals. FLTK timers are not 100% accurate, so they should not be used to measure time intervals, for example.

File functions are called when data is ready to read or write, or when an error condition occurs on a file. They are most often used to monitor network connections (sockets) for data-driven displays.

FLTK applications must periodically check (`Fl::check()`) or wait (`Fl::wait()`) for events or use the `Fl::run()` method to enter a standard event processing loop. Calling `Fl::run()` is equivalent to the following code:

```
while (Fl::wait());
```

`Fl::run()` does not return until all of the windows under FLTK control are closed by the user or your program.

4.2 Compiling Programs with Standard Compilers

Under UNIX (and under Microsoft Windows when using the GNU development tools) you will probably need to tell the compiler where to find the header files. This is usually done using the `-I` option:

```
CC -I/usr/local/include ...  
gcc -I/usr/local/include ...
```

The `fltk-config` script included with FLTK can be used to get the options that are required by your compiler:

```
CC `fltk-config --cxxflags` ...
```

Similarly, when linking your application you will need to tell the compiler to use the FLTK library:

```
CC ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
gcc ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
```

Aside from the "fltk" library, there is also a "fltk_forms" library for the XForms compatibility classes, "fltk_gl" for the OpenGL and GLUT classes, and "fltk_images" for the image file classes, [Fl_Help_Dialog](#) widget, and system icon support.

Note:

The libraries are named "fltk.lib", "fltkgl.lib", "fltkforms.lib", and "fltkimages.lib", respectively under Windows.

As before, the `fltk-config` script included with FLTK can be used to get the options that are required by your linker:

```
CC ... `fltk-config --ldflags`
```

The forms, GL, and images libraries are included with the "--use-foo" options, as follows:

```
CC ... `fltk-config --use-forms --ldflags`
CC ... `fltk-config --use-gl --ldflags`
CC ... `fltk-config --use-images --ldflags`
CC ... `fltk-config --use-forms --use-gl --use-images --ldflags`
```

Finally, you can use the `fltk-config` script to compile a single source file as a FLTK program:

```
fltk-config --compile filename.cpp
fltk-config --use-forms --compile filename.cpp
fltk-config --use-gl --compile filename.cpp
fltk-config --use-images --compile filename.cpp
fltk-config --use-forms --use-gl --use-images --compile filename.cpp
```

Any of these will create an executable named `filename`.

4.3 Compiling Programs with Makefiles

The previous section described how to use `fltk-config` to build a program consisting of a single source file from the command line, and this is very convenient for small test programs. But `fltk-config` can also be used to set the compiler and linker options as variables within a Makefile that can be used to build programs out of multiple source files:

```
CXX      = $(shell fltk-config --cxx)
DEBUG    = -g
CXXFLAGS = $(shell fltk-config --use-gl --use-images --cxxflags ) -I.
LDFLAGS  = $(shell fltk-config --use-gl --use-images --ldflags )
LDSTATIC = $(shell fltk-config --use-gl --use-images --ldstaticflags )
LINK     = $(CXX)

TARGET = cube
OBJS = CubeMain.o CubeView.o CubeViewUI.o
SRCS = CubeMain.cxx CubeView.cxx CubeViewUI.cxx

.SUFFIXES: .o .cxx
```

```
%.o: %.c
    $(CXX) $(CXXFLAGS) $(DEBUG) -c $<

all: $(TARGET)
    $(LINK) -o $(TARGET) $(OBJS) $(LDSTATIC)

$(TARGET): $(OBJS)
CubeMain.o: CubeMain.cxx CubeViewUI.h
CubeView.o: CubeView.cxx CubeView.h CubeViewUI.h
CubeViewUI.o: CubeViewUI.cxx CubeView.h

clean: $(TARGET) $(OBJS)
    rm -f *.o 2> /dev/null
    rm -f $(TARGET) 2> /dev/null
```

4.4 Compiling Programs with Microsoft Visual C++

In Visual C++ you will need to tell the compiler where to find the FLTK header files. This can be done by selecting "Settings" from the "Project" menu and then changing the "Preprocessor" settings under the "C/C++" tab. You will also need to add the FLTK (FLTK.LIB or FLTKD.LIB), the Windows Common Controls (COMCTRL32.LIB), and WinSock2 (WS2_32.LIB) libraries to the "Link" settings.

You can build your Microsoft Windows applications as Console or WIN32 applications. If you want to use the standard C `main()` function as the entry point, FLTK includes a `WinMain()` function that will call your `main()` function for you.

Note: The Visual C++ 5.0 optimizer is known to cause problems with many programs. We only recommend using the "Favor Small Code" optimization setting. The Visual C++ 6.0 optimizer seems to be much better and can be used with the "optimized for speed" setting.

4.5 Naming

All public symbols in FLTK start with the characters 'F' and 'L':

- Functions are either `Fl::foo()` or `fl_foo()`.
- Class and type names are capitalized: `Fl_Foo`.
- Constants and enumerations are uppercase: `FL_FOO`.
- All header files start with `<FL/...>`.

4.6 Header Files

The proper way to include FLTK header files is:

```
#include <FL/Fl_xyz.H>
```

Note:

Case is significant on many operating systems, and the C standard uses the forward slash (/) to separate directories. *Do not use any of the following include lines:*


```
#include <FL\Fl_xyz.H>
#include <fl/fl_xyz.h>
#include <Fl/fl_xyz.h>
```


Chapter 5

3 - Common Widgets and Attributes

This chapter describes many of the widgets that are provided with FLTK and covers how to query and set the standard attributes.

5.1 Buttons

FLTK provides many types of buttons:

- [Fl_Button](#) - A standard push button.
- [Fl_Check_Button](#) - A button with a check box.
- [Fl_Light_Button](#) - A push button with a light.
- [Fl_Repeat_Button](#) - A push button that repeats when held.
- [Fl_Return_Button](#) - A push button that is activated by the `Enter` key.
- [Fl_Round_Button](#) - A button with a radio circle.

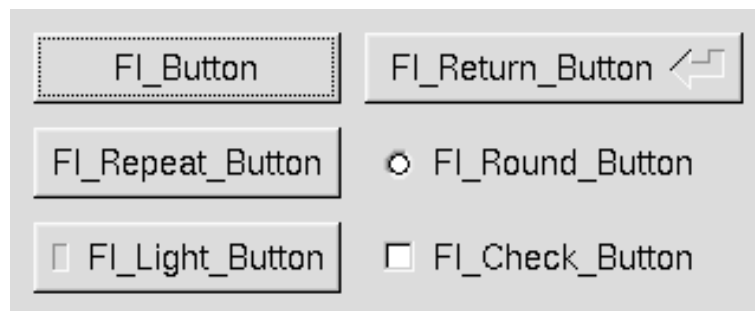


Figure 5.1: FLTK Button Widgets

All of these buttons just need the corresponding `<FL/Fl_xyz_Button.H>` header file. The constructor takes the bounding box of the button and optionally a label string:

```
Fl_Button *button = new Fl_Button(x, y, width, height, "label");
Fl_Light_Button *lbutton = new Fl_Light_Button(x, y, width, height);
Fl_Round_Button *rbutton = new Fl_Round_Button(x, y, width, height, "label");
```

Each button has an associated `type()` which allows it to behave as a push button, toggle button, or radio button:

```
button->type(FL_NORMAL_BUTTON);
lbutton->type(FL_TOGGLE_BUTTON);
rbutton->type(FL_RADIO_BUTTON);
```

For toggle and radio buttons, the `value()` method returns the current button state (0 = off, 1 = on). The `set()` and `clear()` methods can be used on toggle buttons to turn a toggle button on or off, respectively. Radio buttons can be turned on with the `setonly()` method; this will also turn off other radio buttons in the same group.

5.2 Text

FLTK provides several text widgets for displaying and receiving text:

- [Fl_Input](#) - A one-line text input field.
- [Fl_Output](#) - A one-line text output field.
- [Fl_Multiline_Input](#) - A multi-line text input field.
- [Fl_Multiline_Output](#) - A multi-line text output field.
- [Fl_Text_Display](#) - A multi-line text display widget.
- [Fl_Text_Editor](#) - A multi-line text editing widget.
- [Fl_Help_View](#) - A HTML text display widget.

The [Fl_Output](#) and [Fl_Multiline_Output](#) widgets allow the user to copy text from the output field but not change it.

The `value()` method is used to get or set the string that is displayed:

```
Fl_Input *input = new Fl_Input(x, y, width, height, "label");
input->value("Now is the time for all good men...");
```

The string is copied to the widget's own storage when you set the `value()` of the widget.

The [Fl_Text_Display](#) and [Fl_Text_Editor](#) widgets use an associated [Fl_Text_Buffer](#) class for the value, instead of a simple string.

5.3 Valuers

Unlike text widgets, valuers keep track of numbers instead of strings. FLTK provides the following valuers:

- [Fl_Counter](#) - A widget with arrow buttons that shows the current value.
- [Fl_Dial](#) - A round knob.
- [Fl_Roller](#) - An SGI-like dolly widget.
- [Fl_Scrollbar](#) - A standard scrollbar widget.
- [Fl_Slider](#) - A scrollbar with a knob.
- [Fl_Value_Slider](#) - A slider that shows the current value.

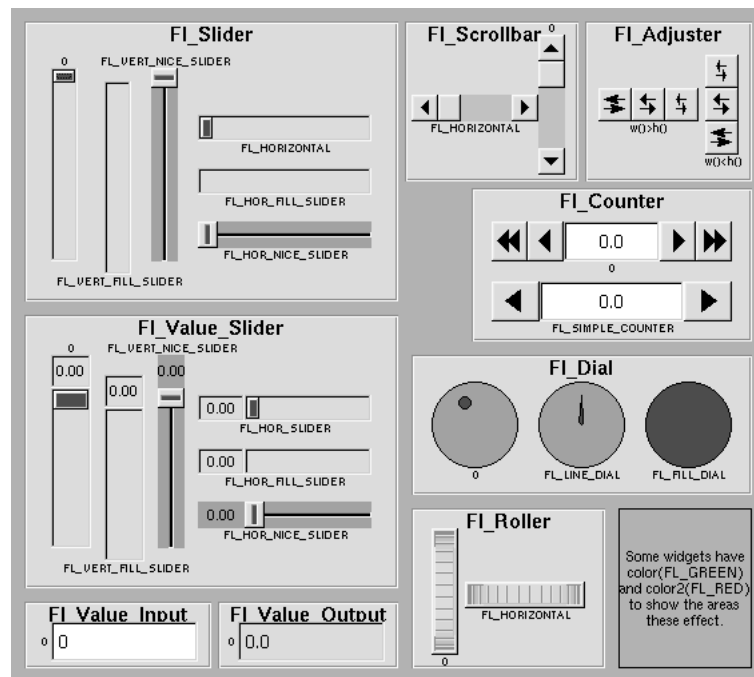


Figure 5.2: FLTK valuator widgets

The `value()` method gets and sets the current value of the widget. The `minimum()` and `maximum()` methods set the range of values that are reported by the widget.

5.4 Groups

The `FL_Group` widget class is used as a general purpose "container" widget. Besides grouping radio buttons, the groups are used to encapsulate windows, tabs, and scrolled windows. The following group classes are available with FLTK:

- `FL_Double_Window` - A double-buffered window on the screen.
- `FL_Gl_Window` - An OpenGL window on the screen.
- `FL_Group` - The base container class; can be used to group any widgets together.
- `FL_Pack` - A collection of widgets that are packed into the group area.
- `FL_Scroll` - A scrolled window area.
- `FL_Tabs` - Displays child widgets as tabs.
- `FL_Tile` - A tiled window area.
- `FL_Window` - A window on the screen.
- `FL_Wizard` - Displays one group of widgets at a time.

5.5 Setting the Size and Position of Widgets

The size and position of widgets is usually set when you create them. You can access them with the `x()`, `y()`, `w()`, and `h()` methods.

You can change the size and position by using the `position()`, `resize()`, and `size()` methods:

```
button->position(x, y);
group->resize(x, y, width, height);
window->size(width, height);
```

If you change a widget's size or position after it is displayed you will have to call `redraw()` on the widget's parent.

5.6 Colors

FLTK stores the colors of widgets as an 32-bit unsigned number that is either an index into a color palette of 256 colors or a 24-bit RGB color. The color palette is *not* the X or WIN32 colormap, but instead is an internal table with fixed contents.

There are symbols for naming some of the more common colors:

- `FL_BLACK`
- `FL_RED`
- `FL_GREEN`
- `FL_YELLOW`
- `FL_BLUE`
- `FL_MAGENTA`
- `FL_CYAN`
- `FL_WHITE`
- `FL_WHITE`

These symbols are the default colors for all FLTK widgets. They are explained in more detail in the chapter [Enumerations](#)

- `FL_FOREGROUND_COLOR`
- `FL_BACKGROUND_COLOR`
- `FL_INACTIVE_COLOR`
- `FL_SELECTION_COLOR`

RGB colors can be set using the `fl_rgb_color()` function:

```
Fl_Color c = fl_rgb_color(85, 170, 255);
```

The widget color is set using the `color()` method:

```
button->color (FL_RED);
```

Similarly, the label color is set using the `labelcolor()` method:

```
button->labelcolor (FL_WHITE);
```

5.7 Box Types

The type `FL_Boxtype` stored and returned in `FL_Widget::box()` is an enumeration defined in [Enumerations.H](#).

Figure 3-3 shows the standard box types included with FLTK.

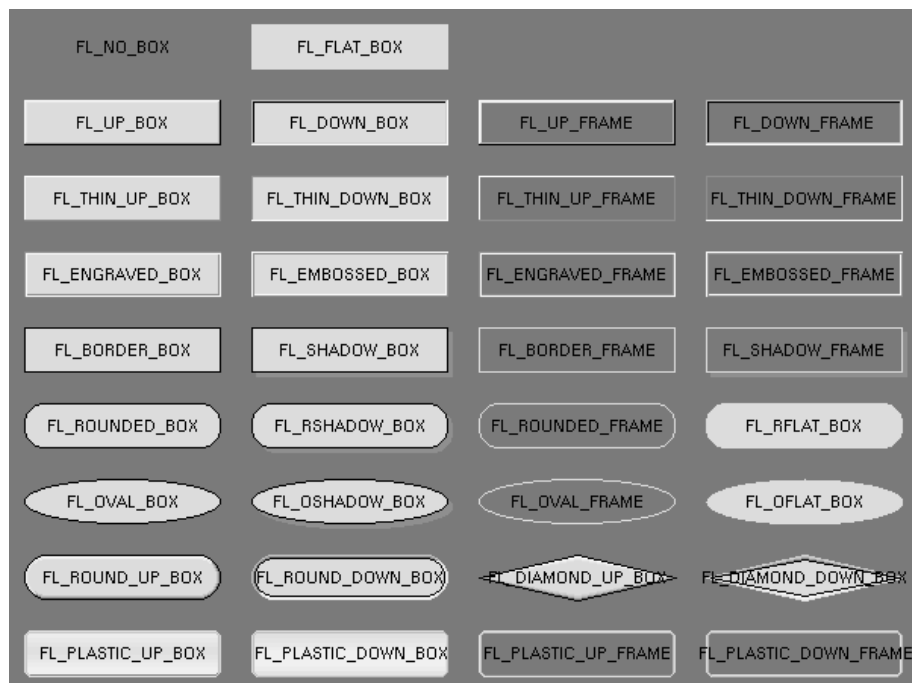


Figure 5.3: FLTK box types

`FL_NO_BOX` means nothing is drawn at all, so whatever is already on the screen remains. The `FL_..._FRAME` types only draw their edges, leaving the interior unchanged. The blue color in Figure 3-3 is the area that is not drawn by the frame types.

5.7.1 Box Types

You can define your own boxtypes by making a small function that draws the box and adding it to the table of boxtypes.

Note:

This interface has changed in FLTK 2.0!

The Drawing Function

The drawing function is passed the bounding box and background color for the widget:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
    ...
}
```

A simple drawing function might fill a rectangle with the given color and then draw a black outline:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
    fl_color(c);
    fl_rectf(x, y, w, h);
    fl_color(FL_BLACK);
    fl_rect(x, y, w, h);
}
```

Fl_Boxtype [fl_down\(Fl_Boxtype b\)](#)

[fl_down\(\)](#) returns the "pressed" or "down" version of a box. If no "down" version of a given box exists, the behavior of this function is undefined and some random box or frame is returned. See also: [fl_frame drawing](#).

Fl_Boxtype [fl_frame\(Fl_Boxtype b\)](#)

[fl_frame\(\)](#) returns the unfilled, frame-only version of a box. If no frame version of a given box exists, the behavior of this function is undefined and some random box or frame is returned. See also: [fl_frame drawing](#).

Fl_Boxtype [fl_box\(Fl_Boxtype b\)](#)

[fl_box\(\)](#) returns the filled version of a frame. If no filled version of a given frame exists, the behavior of this function is undefined and some random box or frame is returned. See also: [fl_frame](#).

Adding Your Box Type

The [Fl::set_boxtype\(\)](#) method adds or replaces the specified box type:

```
#define XYZ_BOX FL_FREE_BOXTYPE

Fl::set_boxtype(XYZ_BOX, xyz_draw, 1, 1, 2, 2);
```

The last 4 arguments to [Fl::set_boxtype\(\)](#) are the offsets for the x, y, width, and height values that should be subtracted when drawing the label inside the box.

A complete box design contains four box types in this order: a filled, neutral box ([UP_BOX](#)), a filled, depressed box ([DOWN_BOX](#)), and the same as outlines only ([UP_FRAME](#) and [DOWN_FRAME](#)). The function [fl_down\(Fl_Boxtype\)](#) expects the neutral design on a boxtype with a numerical value evenly dividable by two. [fl_frame\(Fl_Boxtype\)](#) expects the [UP_BOX](#) design at a value dividable by four.

5.8 Labels and Label Types

Todo

correct signatures for [label\(\)](#) and friends to produce links in [common.dox](#)

The `label()`, `align()`, `labelfont()`, `labelsize()`, `labeltype()`, `image()`, and `deimage()` methods control the labeling of widgets.

label()

The `label()` method sets the string that is displayed for the label. Symbols can be included with the label string by escaping them using the "@" symbol - "@@" displays a single at sign. Figure 3-4 shows the available symbols.

The @ sign may also be followed by the following optional "formatting" characters, in this order:

- '#' forces square scaling, rather than distortion to the widget's shape.
- +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
- '\$' flips the symbol horizontally, ' ' flips it vertically.
- [0-9] - rotates by a multiple of 45 degrees. '5' and '6' do no rotation while the others point in the direction of that key on a numeric keypad. '0', followed by four more digits rotates the symbol by that amount in degrees.

Thus, to show a very large arrow pointing downward you would use the label string "@+92 →".

align()

The `align()` method positions the label. The following constants are defined and may be OR'd together as needed:

- `FL_ALIGN_CENTER` - center the label in the widget.
- `FL_ALIGN_TOP` - align the label at the top of the widget.
- `FL_ALIGN_BOTTOM` - align the label at the bottom of the widget.
- `FL_ALIGN_LEFT` - align the label to the left of the widget.
- `FL_ALIGN_RIGHT` - align the label to the right of the widget.
- `FL_ALIGN_INSIDE` - align the label inside the widget.
- `FL_ALIGN_CLIP` - clip the label to the widget's bounding box.
- `FL_ALIGN_WRAP` - wrap the label text as needed.
- `FL_TEXT_OVER_IMAGE` - show the label text over the image.
- `FL_IMAGE_OVER_TEXT` - show the label image over the text (default).

labeltype()

The `labeltype()` method sets the type of the label. The following standard label types are included:

- `FL_NORMAL_LABEL` - draws the text.
- `FL_NO_LABEL` - does nothing.
- `FL_SHADOW_LABEL` - draws a drop shadow under the text.
- `FL_ENGRAVED_LABEL` - draws edges as though the text is engraved.
- `FL_EMBOSED_LABEL` - draws edges as though the text is raised.
- `FL_ICON_LABEL` - draws the icon associated with the text.

image() and deimage()

The `image()` and `deimage()` methods set an image that will be displayed with the widget. The `deimage()` method sets the image that is shown when the widget is inactive, while the `image()` method sets the image that is shown when the widget is active.

To make an image you use a subclass of `Fl_Image`.

Making Your Own Label Types

Label types are actually indexes into a table of functions that draw them. The primary purpose of this is to use this to draw the labels in ways inaccessible through the `fl_font()` mechanism (e.g. `FL_ENGRAVED_LABEL`) or with program-generated letters or symbology.

Note:

This interface has changed in FLTK 2.0!

Label Type Functions

To setup your own label type you will need to write two functions: one to draw and one to measure the label. The draw function is called with a pointer to a `Fl_Label` structure containing the label information, the bounding box for the label, and the label alignment:

```
void xyz_draw(const Fl_Label *label, int x, int y, int w, int h, Fl_Align align) {  
    ...  
}
```

The label should be drawn *inside* this bounding box, even if `FL_ALIGN_INSIDE` is not enabled. The function is not called if the label value is `NULL`.

The measure function is called with a pointer to a `Fl_Label` structure and references to the width and height:

```
void xyz_measure(const Fl_Label *label, int &w, int &h) {  
    ...  
}
```

The function should measure the size of the label and set `w` and `h` to the size it will occupy.

Adding Your Label Type

The `Fl::set_labeltype` method creates a label type using your draw and measure functions:

```
#define XYZ_LABEL FL_FREE_LABELTYPE

Fl::set_labeltype(XYZ_LABEL, xyz_draw, xyz_measure);
```

The label type number `n` can be any integer value starting at the constant `FL_FREE_LABELTYPE`. Once you have added the label type you can use the `labeltype()` method to select your label type.

The `Fl::set_labeltype` method can also be used to overload an existing label type such as `FL_NORMAL_LABEL`.

Making your own symbols

It is also possible to define your own drawings and add them to the symbol list, so they can be rendered as part of any label.

To create a new symbol, you implement a drawing function `void drawit(Fl_Color c)` which typically uses the [complex drawing functions](#) to generate a vector shape inside a two-by-two units sized box around the origin. This function is then linked into the symbols table using `fl_add_symbol()`:

```
int fl_add_symbol(const char *name, void (*drawit)(Fl_Color), int scalable)
```

`name` is the name of the symbol without the "@"; `scalable` must be set to 1 if the symbol is generated using scalable vector drawing functions.

```
int fl_draw_symbol(const char *name, int x, int y, int w, int h, Fl_Color col)
```

This function draws a named symbol fitting the given rectangle.

5.9 Callbacks

Callbacks are functions that are called when the value of a widget changes. A callback function is sent a [Fl_Widget](#) pointer of the widget that changed and a pointer to data that you provide:

```
void xyz_callback(Fl_Widget *w, void *data) {
    ...
}
```

The `callback()` method sets the callback function for a widget. You can optionally pass a pointer to some data needed for the callback:

```
int xyz_data;

button->callback(xyz_callback, &xyz_data);
```

Normally callbacks are performed only when the value of the widget changes. You can change this using the `Fl_Widget::when()` method:

```
button->when(FL_WHEN_NEVER);
button->when(FL_WHEN_CHANGED);
button->when(FL_WHEN_RELEASE);
button->when(FL_WHEN_RELEASE_ALWAYS);
button->when(FL_WHEN_ENTER_KEY);
button->when(FL_WHEN_ENTER_KEY_ALWAYS);
button->when(FL_WHEN_CHANGED | FL_WHEN_NOT_CHANGED);
```

Note:

You cannot delete a widget inside a callback, as the widget may still be accessed by FLTK after your callback is completed. Instead, use the `Fl::delete_widget()` method to mark your widget for deletion when it is safe to do so.

Hint:

Many programmers new to FLTK or C++ try to use a non-static class method instead of a static class method or function for their callback. Since callbacks are done outside a C++ class, the `this` pointer is not initialized for class methods.

To work around this problem, define a static method in your class that accepts a pointer to the class, and then have the static method call the class method(s) as needed. The data pointer you provide to the `callback()` method of the widget can be a pointer to the instance of your class.

```
class Foo {
    void my_callback(Fl_Widget *w);
    static void my_static_callback(Fl_Widget *w, void *f) { ((Foo *)f)->my_callback(w); }
    ...
}

...

w->callback(my_static_callback, (void *)this);
```

5.10 Shortcuts

Shortcuts are key sequences that activate widgets such as buttons or menu items. The `shortcut()` method sets the shortcut for a widget:

```
button->shortcut(FL_Enter);
button->shortcut(FL_SHIFT + 'b');
button->shortcut(FL_CTRL + 'b');
button->shortcut(FL_ALT + 'b');
button->shortcut(FL_CTRL + FL_ALT + 'b');
button->shortcut(0); // no shortcut
```

The shortcut value is the key event value - the ASCII value or one of the special keys like `FL_Enter` - combined with any modifiers like `Shift`, `Alt`, and `Control`.

Chapter 6

5 - Drawing Things in FLTK

This chapter covers the drawing functions that are provided with FLTK.

6.1 When Can You Draw Things in FLTK?

There are only certain places you can execute drawing code in FLTK. Calling these functions at other places will result in undefined behavior!

- The most common place is inside the virtual `Fl_Widget::draw()` method. To write code here, you must subclass one of the existing `Fl_Widget` classes and implement your own version of `draw()`.
- You can also write `boxtypes` and `labeltypes`. These are small procedures that can be called by existing `Fl_Widget::draw()` methods. These "types" are identified by an 8-bit index that is stored in the widget's `box()`, `labeltype()`, and possibly other properties.
- You can call `Fl_Window::make_current()` to do incremental update of a widget. Use `Fl_Widget::window()` to find the window.

6.2 Drawing Functions

To use the drawing functions you must first include the `<FL/fl_draw.H>` header file. FLTK provides the following types of drawing functions:

- [Boxes](#)
- [Clipping](#)
- [Colors](#)
- [Line Dashes and Thickness](#)
- [Drawing Fast Shapes](#)
- [Drawing Complex Shapes](#)
- [Drawing Text](#)
- [Fonts](#)
- [Character Encoding](#)
- [Drawing Overlays](#)
- [Drawing Images](#)
- [Direct Image Drawing](#)
- [Direct Image Reading](#)
- [Image Classes](#)
- [Offscreen Drawing](#)

6.2.1 Boxes

FLTK provides three functions that can be used to draw boxes for buttons and other UI controls. Each function uses the supplied upper-lefthand corner and width and height to determine where to draw the box.

```
void fl_draw_box(Fl_Boxtype b, int x, int y, int w, int h, Fl_Color c);
```

The first box drawing function is `fl_draw_box()` which draws a standard boxtype `b` in the specified color `c`.

```
void fl_frame(const char *s, int x, int y, int w, int h)
```

The `fl_frame()` function draws a series of line segments around the given box. The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: top, left, bottom, right. The results of calling `fl_frame()` with a string that is not a multiple of 4 characters in length are undefined.

The only difference between this function and `fl_frame2()` is the order of the line segments.

See also: `fl_frame boxtype`.

```
void fl_frame2(const char *s, int x, int y, int w, int h);
```

The `fl_frame2()` function draws a series of line segments around the given box. The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: bottom, right, top, left. The results of calling `fl_frame2()` with a string that is not a multiple of 4 characters in length are undefined.

The only difference between this function and `fl_frame()` is the order of the line segments.

6.2.2 Clipping

You can limit all your drawing to a rectangular region by calling `fl_push_clip()`, and put the drawings back by using `fl_pop_clip()`. This rectangle is measured in pixels and is unaffected by the current transformation matrix.

In addition, the system may provide clipping when updating windows which may be more complex than a simple rectangle.

```
void fl_clip(int x, int y, int w, int h)
```

```
void fl_push_clip(int x, int y, int w, int h)
```

Intersect the current clip region with a rectangle and push this new region onto the stack. The `fl_clip()` name is deprecated and will be removed from future releases.

```
void fl_push_no_clip()
```

Pushes an empty clip region on the stack so nothing will be clipped.

`void fl_pop_clip()`

Restore the previous clip region.

Note: You must call `fl_pop_clip()` once for every time you call `fl_push_clip()`. If you return to FLTK with the clip stack not empty unpredictable results occur.

`int fl_not_clipped(int x, int y, int w, int h)`

Returns non-zero if any of the rectangle intersects the current clip region. If this returns 0 you don't have to draw the object.

Note: Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

`int fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H)`

Intersect the rectangle `x, y, w, h` with the current clip region and returns the bounding box of the result in `X, Y, W, H`. Returns non-zero if the resulting rectangle is different than the original. This can be used to limit the necessary drawing to a rectangle. `W` and `H` are set to zero if the rectangle is completely outside the region.

`void fl_clip_region(Fl_Region r)`

`Fl_Region fl_clip_region()`

Replace the top of the clip stack with a clipping region of any shape. `Fl_Region` is an operating system specific type. The second form returns the current clipping region.

6.3 Colors

FLTK manages colors as 32-bit unsigned integers. Values from 0 to 255 represent colors from the FLTK 1.0.x standard colormap and are allocated as needed on screens without TrueColor support. The `Fl_Color` enumeration type defines the standard colors and color cube for the first 256 colors. All of these are named with symbols in `<FL/Enumerations.H>`.

Color values greater than 255 are treated as 24-bit RGB values. These are mapped to the closest color supported by the screen, either from one of the 256 colors in the FLTK 1.0.x colormap or a direct RGB value on TrueColor screens. You can generate 24-bit RGB color values using the `fl_rgb_color()` function.

`void fl_color(Fl_Color)`

Sets the color for all subsequent drawing operations.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color.

FL_Color `fl_color()`

Returns the last `fl_color()` that was set. This can be used for state save/restore.

void `fl_color(uchar r, uchar g, uchar b)`

Set the color for all subsequent drawing operations. The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used.

6.3.1 Line Dashes and Thickness

FLTK supports drawing of lines with different styles and widths. Full functionality is not available under Windows 95, 98, and Me due to the reduced drawing functionality these operating systems provide.

void `fl_line_style(int style, int width=0, char* dashes=0)`

Set how to draw lines (the "pen"). If you change this it is your responsibility to set it back to the default with `fl_line_style(0)`.

Note: Because of how line styles are implemented on WIN32 systems, you *must* set the line style *after* setting the drawing color. If you set the color after the line style you will lose the line style settings!

style is a bitmask which is a bitwise-OR of the following values. If you don't specify a dash type you will get a solid line. If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.

- FL_SOLID - - - - -
- FL_DASH - - - - -
- FL_DOT
- FL_DASHDOT - . - .
- FL_DASHDOTDOT - . . -
- FL_CAP_FLAT
- FL_CAP_ROUND
- FL_CAP_SQUARE (extends past end point 1/2 line width)
- FL_JOIN_MITER (pointed)

- FL_JOIN_ROUND
- FL_JOIN_BEVEL (flat)

width is the number of pixels thick to draw the lines. Zero results in the system-defined default, which on both X and Windows is somewhat different and nicer than 1.

dashes is a pointer to an array of dash lengths, measured in pixels. The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry. A NULL pointer or a zero-length array results in a solid line. Odd array sizes are not supported and result in undefined behavior.

Note:

The dashes array does not work under Windows 95, 98, or Me, since those operating systems do not support complex line styles.

6.3.2 Drawing Fast Shapes

These functions are used to draw almost all the FLTK widgets. They draw on exact pixel boundaries and are as fast as possible. Their behavior is duplicated exactly on all platforms FLTK is ported. It is undefined whether these are affected by the [transformation matrix](#), so you should only call these while the matrix is set to the identity matrix (the default).

void [fl_point](#)(int x, int y)

Draw a single pixel at the given coordinates.

void [fl_rectf](#)(int x, int y, int w, int h)

void [fl_rectf](#)(int x, int y, int w, int h)

Color a rectangle that exactly fills the given bounding box.

void [fl_rectf](#)(int x, int y, int w, int h, uchar r, uchar g, uchar b)

Color a rectangle with "exactly" the passed *r*, *g*, *b* color. On screens with less than 24 bits of color this is done by drawing a solid-colored block using [fl_draw_image\(\)](#) so that the correct color shade is produced.

void [fl_rect](#)(int x, int y, int w, int h)

void [fl_rect](#)(int x, int y, int w, int h, FL_Color c)

Draw a 1-pixel border *inside* this bounding box.

void [fl_line](#)(int x, int y, int x1, int y1)

void [fl_line](#)(int x, int y, int x1, int y1, int x2, int y2)

Draw one or two lines between the given points.

```
void fl_loop(int x, int y, int x1, int y1, int x2, int y2)
void fl_loop(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
```

Outline a 3 or 4-sided polygon with lines.

```
void fl_polygon(int x, int y, int x1, int y1, int x2, int y2)
void fl_polygon(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
```

Fill a 3 or 4-sided polygon. The polygon must be convex.

```
void fl_xyline(int x, int y, int x1)
void fl_xyline(int x, int y, int x1, int y2)
void fl_xyline(int x, int y, int x1, int y2, int x3)
```

Draw horizontal and vertical lines. A horizontal line is drawn first, then a vertical, then a horizontal.

```
void fl_yxline(int x, int y, int y1)
void fl_yxline(int x, int y, int y1, int x2)
void fl_yxline(int x, int y, int y1, int x2, int y3)
```

Draw vertical and horizontal lines. A vertical line is drawn first, then a horizontal, then a vertical.

```
void fl_arc(int x, int y, int w, int h, double a1, double a2)
void fl_pie(int x, int y, int w, int h, double a1, double a2)
```

Draw ellipse sections using integer coordinates. These functions match the rather limited circle drawing code provided by X and WIN32. The advantage over using `fl_arc` with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counterclockwise from 3’oclock and are the starting and ending angle of the arc, `a2` must be greater or equal to `a1`.

`fl_arc()` draws a series of lines to approximate the arc. Notice that the integer version of `fl_arc()` has a different number of arguments than the `fl_arc()` function described later in this chapter.

`fl_pie()` draws a filled-in pie slice. This slice may extend outside the line drawn by `fl_arc()`; to avoid this use `w - 1` and `h - 1`.

```
void fl_scroll(int X, int Y, int W, int H, int dx, int dy, void (*draw_area)(void*, int,int,int,int), void* data)
```

Scroll a rectangle and draw the newly exposed portions. The contents of the rectangular area is first shifted by `dx` and `dy` pixels. The callback is then called for every newly exposed rectangular area,

6.3.3 Drawing Complex Shapes

The complex drawing functions let you draw arbitrary shapes with 2-D linear transformations. The functionality matches that found in the Adobe®PostScript™ language. The exact pixels that are filled are less defined than for the fast drawing functions so that FLTK can take advantage of drawing hardware. On both X and WIN32 the transformed vertices are rounded to integers before drawing the line segments: this severely limits the accuracy of these functions for complex graphics, so use OpenGL when greater accuracy and/or performance is required.

`void fl_push_matrix()`

`void fl_pop_matrix()`

Save and restore the current transformation. The maximum depth of the stack is 4.

`void fl_scale(float x, float y)`

`void fl_scale(float x)`

`void fl_translate(float x, float y)`

`void fl_rotate(float d)`

`void fl_mult_matrix(float a, float b, float c, float d, float x, float y)`

Concatenate another transformation onto the current one. The rotation angle is in degrees (not radians) and is counter-clockwise.

`double fl_transform_x(double x, double y)`

`double fl_transform_y(double x, double y)`

`double fl_transform_dx(double x, double y)`

`double fl_transform_dy(double x, double y)`

`void fl_transformed_vertex(double xf, double yf)`

Transform a coordinate or a distance through the current transformation matrix. After transforming a coordinate pair, it can be added to the vertex list without any further translations using `fl_transformed_vertex`.

`void fl_begin_points()`

`void fl_end_points()`

Start and end drawing a list of points. Points are added to the list with `fl_vertex`.

`void fl_begin_line()`

`void fl_end_line()`

Start and end drawing lines.

```
void fl_begin_loop()
```

```
void fl_end_loop()
```

Start and end drawing a closed sequence of lines.

```
void fl_begin_polygon()
```

```
void fl_end_polygon()
```

Start and end drawing a convex filled polygon.

```
void fl_begin_complex_polygon()
```

```
void fl_gap()
```

```
void fl_end_complex_polygon()
```

Start and end drawing a complex filled polygon. This polygon may be concave, may have holes in it, or may be several disconnected pieces. Call `fl_gap()` to separate loops of the path. It is unnecessary but harmless to call `fl_gap()` before the first vertex, after the last one, or several times in a row.

Note: For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. Holes should be drawn in the opposite direction of the outside loop.

`fl_gap()` should only be called between `fl_begin_complex_polygon()` and `fl_end_complex_polygon()`. To outline the polygon, use `fl_begin_loop()` and replace each `fl_gap()` with `fl_end_loop();fl_begin_loop()`.

```
void fl_vertex(float x, float y)
```

Add a single vertex to the current path.

```
void fl_curve(float x, float y, float x1, float y1, float x2, float y2, float x3, float y3)
```

Add a series of points on a Bezier curve to the path. The curve ends (and two of the points) are at `x, y` and `x3, y3`.

```
void fl_arc(float x, float y, float r, float start, float end)
```

Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using `scale` and `rotate` before calling `fl_arc()`. `x, y` are the center of the circle, and `r` is its radius. `fl_arc()` takes `start` and `end` angles that are measured in degrees counter-clockwise from 3 o'clock. If `end` is less than `start` then it draws the arc in a clockwise direction.

```
void fl_circle(float x, float y, float r)
```

`fl_circle()` is equivalent to `fl_arc(...,0,360)` but may be faster. It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use `fl_arc()`.

Note: `fl_circle()` draws incorrectly if the transformation is both rotated and non-square scaled.

6.3.4 Drawing Text

All text is drawn in the `current font`. It is undefined whether this location or the characters are modified by the current transformation.

```
void fl_draw(const char *, int x, int y)
```

```
void fl_draw(const char *, int n, int x, int y)
```

Draw a nul-terminated string or an array of `n` characters starting at the given location. Text is aligned to the left and to the baseline of the font. To align to the bottom, subtract `fl_descent()` from `y`. To align to the top, subtract `fl_descent()` and add `fl_height()`. This version of `fl_draw` provides direct access to the text drawing function of the underlying OS. It does not apply any special handling to control characters.

```
void fl_draw(const char *, int x, int y, int w, int h, FL_Align align, FL_Image *img = 0, int draw_symbols = 1)
```

Fancy string drawing function which is used to draw all the labels. The string is formatted and aligned inside the passed box. Handles `'\t'` and `'\n'`, expands all other control characters to `^X`, and aligns inside or against the edges of the box described by `x`, `y`, `w` and `h`. See `FL_Widget::align()` for values for `align`. The value `FL_ALIGN_INSIDE` is ignored, as this function always prints inside the box.

If `img` is provided and is not `NULL`, the image is drawn above or below the text as specified by the `align` value.

The `draw_symbols` argument specifies whether or not to look for symbol names starting with the `"@"` character.

The text length is limited to 1024 characters per line.

```
void fl_measure(const char *, int &w, int &h, int draw_symbols = 1)
```

Measure how wide and tall the string will be when printed by the `fl_draw(...align)` function. If the incoming `w` is non-zero it will wrap to that width.

```
int fl_height()
```

Recommended minimum line spacing for the current font. You can also just use the value of `size` passed to `fl_font()`.

int [fl_descent\(\)](#)

Recommended distance above the bottom of a [fl_height\(\)](#) tall box to draw the text at so it looks centered vertically in that box.

float [fl_width\(const char*\)](#)

float [fl_width\(const char*, int n\)](#)

float [fl_width\(uchar\)](#)

Return the pixel width of a nul-terminated string, a sequence of *n* characters, or a single character in the current font.

const char* [fl_shortcut_label\(int shortcut\)](#)

Unparse a shortcut value as used by [Fl_Button](#) or [Fl_Menu_Item](#) into a human-readable string like "Alt+N". This only works if the shortcut is a character key or a numbered function key. If the shortcut is zero an empty string is returned. The return value points at a static buffer that is overwritten with each call.

6.3.5 Fonts

FLTK supports a set of standard fonts based on the Times, Helvetica/Arial, Courier, and Symbol typefaces, as well as custom fonts that your application may load. Each font is accessed by an index into a font table.

Initially only the first 16 faces are filled in. There are symbolic names for them: `FL_HELVETICA`, `FL_TIMES`, `FL_COURIER`, and modifier values `FL_BOLD` and `FL_ITALIC` which can be added to these, and `FL_SYMBOL` and `FL_ZAPF_DINGBATS`. Faces greater than 255 cannot be used in [Fl_Widget](#) labels, since [Fl_Widget](#) stores the index as a byte.

void [fl_font\(int face, int size\)](#)

Set the current font, which is then used by the routines described above. You may call this outside a draw context if necessary to call [fl_width\(\)](#), but on X this will open the display.

The font is identified by a *face* and a *size*. The size of the font is measured in `pixels` and not "points". Lines should be spaced *size* pixels apart or more.

int [fl_font\(\)](#)

int [fl_size\(\)](#)

Returns the face and size set by the most recent call to [fl_font\(a,b\)](#). This can be used to save/restore the font.

6.3.6 Character Encoding

FLTK 1 supports western character sets using the eight bit encoding of the user-selected global code page. For MS Windows and X11, the code page is assumed to be Windows-1252/Latin1, a superset to ISO 8859-1. On Mac OS X, we assume MacRoman.

FLTK provides the functions `fl_latin1_to_local()`, `fl_local_to_latin1()`, `fl_mac_roman_to_local()`, and `fl_local_to_mac_roman()` to convert strings between both encodings. These functions are only required if your source code contains "C"-strings with international characters and if this source will be compiled on multiple platforms.

Assuming that the following source code was written on MS Windows, this example will output the correct label on OS X and X11 as well. Without the conversion call, the label on OS X would read `Fahrvergn,gen` with a deformed umlaut u ("cedille", html "¸").

```
btn = new Fl_Button(10, 10, 300, 25);
btn->copy_label(fl_latin1_to_local("Fahrvergnügen"));
```

Note:

If your application uses characters that are not part of both encodings, or it will be used in areas that commonly use different code pages, you might consider upgrading to FLTK 2 which supports UTF-8 encoding.

Todo

`drawing.dox`: I fixed the above encoding problem of these ¸ and umlaut characters, but this text is obsoleted by FLTK 1.3 with utf-8 encoding, or must be rewritten accordingly: How to use native (e.g. Windows "ANSI", or ISO-8859-x) encoding in embedded strings for labels, error messages and more. Please check this (utf-8) encoding on different OS'es and with different language and font environments.

For more information about character encoding, unicode and utf-8 see chapter [11 - Unicode and utf-8 Support](#).

6.3.7 Drawing Overlays

These functions allow you to draw interactive selection rectangles without using the overlay hardware. FLTK will XOR a single rectangle outline over a window.

```
void fl_overlay_rect(int x, int y, int w, int h);
```

```
void fl_overlay_clear();
```

`fl_overlay_rect()` draws a selection rectangle, erasing any previous rectangle by XOR'ing it first. `fl_overlay_clear()` will erase the rectangle without drawing a new one.

Using these functions is tricky. You should make a widget with both a `handle()` and `draw()` method. `draw()` should call `fl_overlay_clear()` before doing anything else. Your `handle()` method should call `window()->make_current()` and then `fl_overlay_rect()` after `FL_DRAG` events, and should call `fl_overlay_clear()` after a `FL_RELEASE` event.

6.4 Drawing Images

To draw images, you can either do it directly from data in your memory, or you can create a [FL_Image](#) object. The advantage of drawing directly is that it is more intuitive, and it is faster if the image data changes more often than it is redrawn. The advantage of using the object is that FLTK will cache translated forms of the image (on X it uses a server pixmap) and thus redrawing is *much* faster.

6.4.1 Direct Image Drawing

The behavior when drawing images when the current transformation matrix is not the identity is not defined, so you should only draw images when the matrix is set to the identity.

```
void fl_draw_image(const uchar *buf,int X,int Y,int W,int H,int D,int L)
```

```
void fl_draw_image_mono(const uchar *buf,int X,int Y,int W,int H,int D,int L)
```

Draw an 8-bit per color RGB or luminance image. The pointer points at the "r" data of the top-left pixel. Color data must be in *r, g, b* order. *X, Y* are where to put the top-left corner. *W* and *H* define the size of the image. *D* is the delta to add to the pointer between pixels, it may be any value greater or equal to 3, or it can be negative to flip the image horizontally. *L* is the delta to add to the pointer between lines (if 0 is passed it uses $W * D$), and may be larger than $W * D$ to crop data, or negative to flip the image vertically.

It is highly recommended that you put the following code before the first `show()` of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling [fl_draw_image_mono\(\)](#). Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting *D* greater than 1 will let you display one channel of a color image.

Note: The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

```
typedef void (*Fl_Draw_Image_Cb)(void *data,int x,int y,int w,uchar *buf)
```

```
void fl_draw_image(Fl_Draw_Image_Cb cb,void *data,int X,int Y,int W,int H,int D)
```

```
void fl_draw_image_mono(Fl_Draw_Image_Cb cb,void *data,int X,int Y,int W,int H,int D)
```

Call the passed function to provide each scan line of the image. This lets you generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

The callback is called with the `void *` user data pointer which can be used to point at a structure of information about the image, and the *x, y*, and *w* of the scan line desired from the image. 0,0 is the

upper-left corner of the image, *not* X, Y . A pointer to a buffer to put the data into is passed. You must copy w pixels from scanline y , starting at pixel x , to this buffer.

Due to cropping, less than the whole image may be requested. So x may be greater than zero, the first y may be greater than zero, and w may be less than W . The buffer is long enough to store the entire $W * D$ pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if x is not zero, copy the data over so the x 'th pixel is at the start of the buffer.

You can assume the y 's will be consecutive, except the first one may be greater than zero.

If D is 4 or more, you must fill in the unused bytes with zero.

```
int fl_draw_pixmap(char* const* data, int x, int y, Fl_Color bg)
```

```
int fl_draw_pixmap(const char* const* cdata, int x, int y, Fl_Color bg)
```

Draws XPM image data, with the top-left corner at the given position. The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps. This function returns zero if there was any error decoding the XPM data.

To use an XPM, do:

```
#include "foo.xpm"
...
fl_draw_pixmap(foo, X, Y);
```

Transparent colors are replaced by the optional `Fl_Color` argument. To draw with true transparency you must use the `Fl_Pixmap` class.

```
int fl_measure_pixmap(char* const* data, int &w, int &h)
```

```
int fl_measure_pixmap(const char* const* cdata, int &w, int &h)
```

An XPM image contains the dimensions in its data. This function finds and returns the width and height. The return value is non-zero if the dimensions were parsed ok and zero if there was any problem.

6.4.2 Direct Image Reading

FLTK provides a single function for reading from the current window or off-screen buffer into a RGB(A) image buffer.

```
uchar* fl_read_image(uchar *p, int X, int Y, int W, int H, int alpha)
```

Read a RGB(A) image from the current window or off-screen buffer. The p argument points to a buffer that can hold the image and must be at least $W * H * 3$ bytes when reading RGB images and $W * H * 4$ bytes when reading RGBA images. If `NULL`, `fl_read_image()` will create an array of the proper size which can be freed using `delete[]`.

The `alpha` parameter controls whether an alpha channel is created and the value that is placed in the alpha channel. If 0, no alpha channel is generated.

6.4.3 Image Classes

FLTK provides a base image class called `Fl_Image` which supports creating, copying, and drawing images of various kinds, along with some basic color operations. Images can be used as labels for widgets using the `image()` and `deimage()` methods or drawn directly.

The `Fl_Image` class does almost nothing by itself, but is instead supported by three basic image types:

- `Fl_Bitmap`
- `Fl_Pixmap`
- `Fl_RGB_Image`

The `Fl_Bitmap` class encapsulates a mono-color bitmap image. The `draw()` method draws the image using the current drawing color.

The `Fl_Pixmap` class encapsulates a colormapped image. The `draw()` method draws the image using the colors in the file, and masks off any transparent colors automatically.

The `Fl_RGB_Image` class encapsulates a full-color (or grayscale) image with 1 to 4 color components. Images with an even number of components are assumed to contain an alpha channel that is used for transparency. The transparency provided by the `draw()` method is either a 24-bit blend against the existing window contents or a "screen door" transparency mask, depending on the platform and screen color depth.

char `fl_can_do_alpha_blending()`

`fl_can_do_alpha_blending()` will return 1, if your platform supports true alpha blending for RGBA images, or 0, if FLTK will use screen door transparency.

FLTK also provides several image classes based on the three standard image types for common file formats:

- `Fl_GIF_Image`
- `Fl_JPEG_Image`
- `Fl_PNG_Image`
- `Fl_PNM_Image`
- `Fl_XBM_Image`
- `Fl_XPM_Image`

Each of these image classes load a named file of the corresponding format. The `Fl_Shared_Image` class can be used to load any type of image file - the class examines the file and constructs an image of the appropriate type.

Finally, FLTK provides a special image class called `Fl_Tiled_Image` to tile another image object in the specified area. This class can be used to tile a background image in a `Fl_Group` widget, for example.

virtual void `copy()`;

virtual void `copy(int w, int h)`;

The `copy()` method creates a copy of the image. The second form specifies the new size of the image - the image is resized using the nearest-neighbor algorithm.

```
void draw(int x, int y, int w, int h, int ox = 0, int oy = 0);
```

The `draw()` method draws the image object. `x, y, w, h` indicates a destination rectangle. `ox, oy, w, h` is a source rectangle. This source rectangle is copied to the destination. The source rectangle may extend outside the image, i.e. `ox` and `oy` may be negative and `w` and `h` may be bigger than the image, and this area is left unchanged.

```
void draw(int x, int y)
```

Draws the image with the upper-left corner at `x, y`. This is the same as doing `draw(x,y,img->w(),img->h(),0,0)`.

6.4.4 Offscreen Drawing

Sometimes it can be very useful to generate a complex drawing in memory first and copy it to the screen at a later point in time. This technique can significantly reduce the amount of repeated drawing. [FL_Double_Window](#) uses offscreen rendering to avoid flickering on systems that don't support double-buffering natively.

```
Fl_Offscreen fl_create_offscreen(int w, int h)
```

Create an RGB offscreen buffer with `w*h` pixels.

```
void fl_delete_offscreen(Fl_Offscreen)
```

Delete a previously created offscreen buffer. All drawings are lost.

```
void fl_begin_offscreen(Fl_Offscreen)
```

Send all subsequent drawing commands to this offscreen buffer. FLTK can draw into a buffer at any time. There is no need to wait for an [Fl_Widget::draw\(\)](#) to occur.

```
void fl_end_offscreen()
```

Quit sending drawing commands to this offscreen buffer.

```
void fl_copy_offscreen(int x, int y, int w, int h, Fl_Offscreen osrc, int srcx, int srcy)
```

Copy a rectangular area of the size `w*h` from `srcx, srcy` in the offscreen buffer into the current buffer at `x, y`.

Chapter 7

4 - Designing a Simple Text Editor

This chapter takes you through the design of a simple FLTK-based text editor.

7.1 Determining the Goals of the Text Editor

Since this will be the first big project you'll be doing with FLTK, let's define what we want our text editor to do:

1. Provide a menubar/menus for all functions.
2. Edit a single text file, possibly with multiple views.
3. Load from a file.
4. Save to a file.
5. Cut/copy/delete/paste functions.
6. Search and replace functions.
7. Keep track of when the file has been changed.

7.2 Designing the Main Window

Now that we've outlined the goals for our editor, we can begin with the design of our GUI. Obviously the first thing that we need is a window, which we'll place inside a class called `EditorWindow`:

```
class EditorWindow : public Fl_Double_Window {
public:
    EditorWindow(int w, int h, const char* t);
    ~EditorWindow();

    Fl_Window      *replace_dlg;
    Fl_Input       *replace_find;
    Fl_Input       *replace_with;
    Fl_Button      *replace_all;
    Fl_Return_Button *replace_next;
    Fl_Button      *replace_cancel;

    Fl_Text_Editor *editor;
    char           search[256];
};
```

7.3 Variables

Our text editor will need some global variables to keep track of things:

```
int          changed = 0;
char         filename[256] = "";
Fl_Text_Buffer *textbuf;
```

The `textbuf` variable is the text editor buffer for our window class described previously. We'll cover the other variables as we build the application.

7.4 Menubars and Menus

The first goal requires us to use a menubar and menus that define each function the editor needs to perform. The `Fl_Menu_Item` structure is used to define the menus and items in a menubar:

```
Fl_Menu_Item menuitems[] = {
    { "&File", 0, 0, 0, FL_SUBMENU },
    { "&New File", 0, (Fl_Callback *)new_cb },
    { "&Open File...", FL_CTRL + 'o', (Fl_Callback *)open_cb },
    { "&Insert File...", FL_CTRL + 'i', (Fl_Callback *)insert_cb, 0, FL_MENU_DIVIDER },
    { "&Save File", FL_CTRL + 's', (Fl_Callback *)save_cb },
    { "Save File &As...", FL_CTRL + FL_SHIFT + 's', (Fl_Callback *)saveas_cb, 0, FL_MENU_DIVIDER },
    { "New &View", FL_ALT + 'v', (Fl_Callback *)view_cb, 0 },
    { "&Close View", FL_CTRL + 'w', (Fl_Callback *)close_cb, 0, FL_MENU_DIVIDER },
    { "E&xit", FL_CTRL + 'q', (Fl_Callback *)quit_cb, 0 },
    { 0 },

    { "&Edit", 0, 0, 0, FL_SUBMENU },
    { "&Undo", FL_CTRL + 'z', (Fl_Callback *)undo_cb, 0, FL_MENU_DIVIDER },
    { "Cu&t", FL_CTRL + 'x', (Fl_Callback *)cut_cb },
    { "&Copy", FL_CTRL + 'c', (Fl_Callback *)copy_cb },
    { "&Paste", FL_CTRL + 'v', (Fl_Callback *)paste_cb },
    { "&Delete", 0, (Fl_Callback *)delete_cb },
    { 0 },

    { "&Search", 0, 0, 0, FL_SUBMENU },
    { "&Find...", FL_CTRL + 'f', (Fl_Callback *)find_cb },
    { "F&ind Again", FL_CTRL + 'g', find2_cb },
    { "&Replace...", FL_CTRL + 'r', replace_cb },
    { "Re&place Again", FL_CTRL + 't', replace2_cb },
    { 0 },

    { 0 }
};
```

Once we have the menus defined we can create the `Fl_Menu_Bar` widget and assign the menus to it with:

```
Fl_Menu_Bar *m = new Fl_Menu_Bar(0, 0, 640, 30);
m->copy(menuitems);
```

We'll define the callback functions later.

7.5 Editing the Text

To keep things simple our text editor will use the `Fl_Text_Editor` widget to edit the text:

```
w->editor = new Fl_Text_Editor(0, 30, 640, 370);
w->editor->buffer(textbuf);
```

So that we can keep track of changes to the file, we also want to add a "modify" callback:

```
textbuf->add_modify_callback(changed_cb, w);
textbuf->call_modify_callbacks();
```

Finally, we want to use a mono-spaced font like `FL_COURIER`:

```
w->editor->textfont(FL_COURIER);
```

7.6 The Replace Dialog

We can use the FLTK convenience functions for many of the editor's dialogs, however the replace dialog needs its own custom window. To keep things simple we will have a "find" string, a "replace" string, and "replace all", "replace next", and "cancel" buttons. The strings are just `Fl_Input` widgets, the "replace all" and "cancel" buttons are `Fl_Button` widgets, and the "replace next" button is a `Fl_Return_Button` widget:

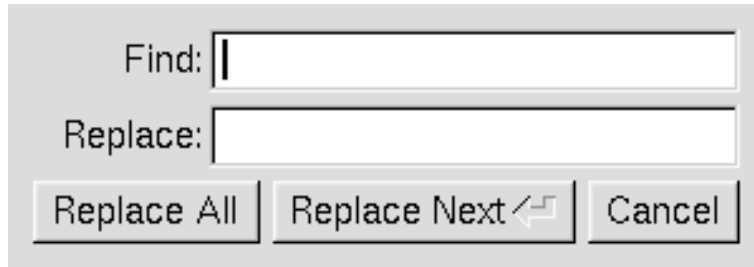


Figure 7.1: The search and replace dialog

```
Fl_Window *replace_dlg = new Fl_Window(300, 105, "Replace");
Fl_Input *replace_find = new Fl_Input(70, 10, 200, 25, "Find:");
Fl_Input *replace_with = new Fl_Input(70, 40, 200, 25, "Replace:");
Fl_Button *replace_all = new Fl_Button(10, 70, 90, 25, "Replace All");
Fl_Button *replace_next = new Fl_Button(105, 70, 120, 25, "Replace Next");
Fl_Button *replace_cancel = new Fl_Button(230, 70, 60, 25, "Cancel");
```

7.7 Callbacks

Now that we've defined the GUI components of our editor, we need to define our callback functions.

7.7.1 changed_cb()

This function will be called whenever the user changes any text in the `editor` widget:

```
void changed_cb(int, int nInserted, int nDeleted, int, const char*, void* v) {
    if ((nInserted || nDeleted) && !loading) changed = 1;
    EditorWindow *w = (EditorWindow *)v;
    set_title(w);
    if (loading) w->editor->show_insert_position();
}
```

The `set_title()` function is one that we will write to set the changed status on the current file. We're doing it this way because we want to show the changed status in the window's title bar.

7.7.2 copy_cb()

This callback function will call `Fl_Text_Editor::kf_copy()` to copy the currently selected text to the clipboard:

```
void copy_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    Fl_Text_Editor::kf_copy(0, e->editor);
}
```

7.7.3 cut_cb()

This callback function will call [Fl_Text_Editor::kf_cut\(\)](#) to cut the currently selected text to the clipboard:

```
void cut_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    Fl_Text_Editor::kf_cut(0, e->editor);
}
```

7.7.4 delete_cb()

This callback function will call [Fl_Text_Buffer::remove_selection\(\)](#) to delete the currently selected text to the clipboard:

```
void delete_cb(Fl_Widget*, void* v) {
    textbuf->remove_selection();
}
```

7.7.5 find_cb()

This callback function asks for a search string using the [fl_input\(\)](#) convenience function and then calls the [find2_cb\(\)](#) function to find the string:

```
void find_cb(Fl_Widget* w, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    const char *val;

    val = fl_input("Search String:", e->search);
    if (val != NULL) {
        // User entered a string - go find it!
        strcpy(e->search, val);
        find2_cb(w, v);
    }
}
```

7.7.6 find2_cb()

This function will find the next occurrence of the search string. If the search string is blank then we want to pop up the search dialog:

```
void find2_cb(Fl_Widget* w, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    if (e->search[0] == '\0') {
        // Search string is blank; get a new one...
        find_cb(w, v);
        return;
    }

    int pos = e->editor->insert_position();
    int found = textbuf->search_forward(pos, e->search, &pos);
    if (found) {
        // Found a match; select and update the position...
        textbuf->select(pos, pos+strlen(e->search));
        e->editor->insert_position(pos+strlen(e->search));
        e->editor->show_insert_position();
    }
    else fl_alert("No occurrences of \'%s\' found!", e->search);
}
```

If the search string cannot be found we use the `fl_alert()` convenience function to display a message to that effect.

7.7.7 new_cb()

This callback function will clear the editor widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void new_cb(Fl_Widget*, void*) {
    if (!check_save()) return;

    filename[0] = '\0';
    textbuf->select(0, textbuf->length());
    textbuf->remove_selection();
    changed = 0;
    textbuf->call_modify_callbacks();
}
```

7.7.8 open_cb()

This callback function will ask the user for a filename and then load the specified file into the input widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void open_cb(Fl_Widget*, void*) {
    if (!check_save()) return;

    char *newfile = fl_file_chooser("Open File?", "*", filename);
    if (newfile != NULL) load_file(newfile, -1);
}
```

We call the `load_file()` function to actually load the file.

7.7.9 paste_cb()

This callback function will call `Fl_Text_Editor::kf_paste()` to paste the clipboard at the current position:

```
void paste_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    Fl_Text_Editor::kf_paste(0, e->editor);
}
```

7.7.10 quit_cb()

The quit callback will first see if the current file has been modified, and if so give the user a chance to save it. It then exits from the program:

```
void quit_cb(Fl_Widget*, void*) {
    if (changed && !check_save())
        return;

    exit(0);
}
```

7.7.11 replace_cb()

The replace callback just shows the replace dialog:

```
void replace_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    e->replace_dlg->show();
}
```

7.7.12 replace2_cb()

This callback will replace the next occurrence of the replacement string. If nothing has been entered for the replacement string, then the replace dialog is displayed instead:

```
void replace2_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    const char *find = e->replace_find->value();
    const char *replace = e->replace_with->value();

    if (find[0] == '\0') {
        // Search string is blank; get a new one...
        e->replace_dlg->show();
        return;
    }

    e->replace_dlg->hide();

    int pos = e->editor->insert_position();
    int found = textbuf->search_forward(pos, find, &pos);

    if (found) {
        // Found a match; update the position and replace text...
        textbuf->select(pos, pos+strlen(find));
        textbuf->remove_selection();
        textbuf->insert(pos, replace);
        textbuf->select(pos, pos+strlen(replace));
        e->editor->insert_position(pos+strlen(replace));
        e->editor->show_insert_position();
    }
    else fl_alert("No occurrences of \'%s\' found!", find);
}
```

7.7.13 replall_cb()

This callback will replace all occurrences of the search string in the file:

```
void replall_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    const char *find = e->replace_find->value();
    const char *replace = e->replace_with->value();

    find = e->replace_find->value();
    if (find[0] == '\0') {
        // Search string is blank; get a new one...
        e->replace_dlg->show();
        return;
    }

    e->replace_dlg->hide();

    e->editor->insert_position(0);
```

```

int times = 0;

// Loop through the whole string
for (int found = 1; found;) {
    int pos = e->editor->insert_position();
    found = textbuf->search_forward(pos, find, &pos);

    if (found) {
        // Found a match; update the position and replace text...
        textbuf->select(pos, pos+strlen(find));
        textbuf->remove_selection();
        textbuf->insert(pos, replace);
        e->editor->insert_position(pos+strlen(replace));
        e->editor->show_insert_position();
        times++;
    }
}

if (times) fl_message("Replaced %d occurrences.", times);
else fl_alert("No occurrences of \'%s\' found!", find);
}

```

7.7.14 replcan_cb()

This callback just hides the replace dialog:

```

void replcan_cb(Fl_Widget*, void* v) {
    EditorWindow* e = (EditorWindow*)v;
    e->replace_dlg->hide();
}

```

7.7.15 save_cb()

This callback saves the current file. If the current filename is blank it calls the "save as" callback:

```

void save_cb(void) {
    if (filename[0] == '\0') {
        // No filename - get one!
        saveas_cb();
        return;
    }
    else save_file(filename);
}

```

The `save_file()` function saves the current file to the specified filename.

7.7.16 saveas_cb()

This callback asks the user for a filename and saves the current file:

```

void saveas_cb(void) {
    char *newfile;

    newfile = fl_file_chooser("Save File As?", "*", filename);
    if (newfile != NULL) save_file(newfile);
}

```

The `save_file()` function saves the current file to the specified filename.

7.8 Other Functions

Now that we've defined the callback functions, we need our support functions to make it all work:

7.8.1 `check_save()`

This function checks to see if the current file needs to be saved. If so, it asks the user if they want to save it:

```
int check_save(void) {
    if (!changed) return 1;

    int r = fl_choice("The current file has not been saved.\n"
                     "Would you like to save it now?",
                     "Cancel", "Save", "Discard");

    if (r == 1) {
        save_cb(); // Save the file...
        return !changed;
    }

    return (r == 2) ? 1 : 0;
}
```

7.8.2 `load_file()`

This function loads the specified file into the `textbuf` variable:

```
int loading = 0;
void load_file(char *newfile, int ipos) {
    loading = 1;
    int insert = (ipos != -1);
    changed = insert;
    if (!insert) strcpy(filename, "");
    int r;
    if (!insert) r = textbuf->loadfile(newfile);
    else r = textbuf->insertfile(newfile, ipos);
    if (r)
        fl_alert("Error reading from file '%s':\n%s.", newfile, strerror(errno));
    else
        if (!insert) strcpy(filename, newfile);
    loading = 0;
    textbuf->call_modify_callbacks();
}
```

When loading the file we use the [FL_Text_Buffer::loadfile\(\)](#) method to "replace" the text in the buffer, or the [FL_Text_Buffer::insertfile\(\)](#) method to insert text in the buffer from the named file.

7.8.3 `save_file()`

This function saves the current buffer to the specified file:

```
void save_file(char *newfile) {
    if (textbuf->savefile(newfile))
        fl_alert("Error writing to file '%s':\n%s.", newfile, strerror(errno));
    else
        strcpy(filename, newfile);
}
```

```
changed = 0;
textbuf->call_modify_callbacks();
}
```

7.8.4 set_title()

This function checks the `changed` variable and updates the window label accordingly:

```
void set_title(Fl_Window* w) {
    if (filename[0] == '\\0') strcpy(title, "Untitled");
    else {
        char *slash;
        slash = strrchr(filename, '/');
#ifdef WIN32
        if (slash == NULL) slash = strrchr(filename, '\\');
#endif
        if (slash != NULL) strcpy(title, slash + 1);
        else strcpy(title, filename);
    }

    if (changed) strcat(title, " (modified)");

    w->label(title);
}
```

7.9 The main() Function

Once we've created all of the support functions, the only thing left is to tie them all together with the `main()` function. The `main()` function creates a new text buffer, creates a new view (window) for the text, shows the window, loads the file on the command-line (if any), and then enters the FLTK event loop:

```
int main(int argc, char **argv) {
    textbuf = new Fl_Text_Buffer;

    Fl_Window* window = new_view();

    window->show(1, argv);

    if (argc > 1) load_file(argv[1], -1);

    return Fl::run();
}
```

7.10 Compiling the Editor

The complete source for our text editor can be found in the `test/editor.cxx` source file. Both the Makefile and Visual C++ workspace include the necessary rules to build the editor. You can also compile it using a standard compiler with:

```
CC -o editor editor.cxx -lfltk -lXext -lX11 -lm
```

or by using the `fltk-config` script with:

```
fltk-config --compile editor.cxx
```


As noted in [FLTK Basics](#), you may need to include compiler and linker options to tell them where to find the FLTK library. Also, the `CC` command may also be called `gcc` or `c++` on your system.

Congratulations, you've just built your own text editor!

7.11 The Final Product

The final editor window should look like the image in Figure 4-2.

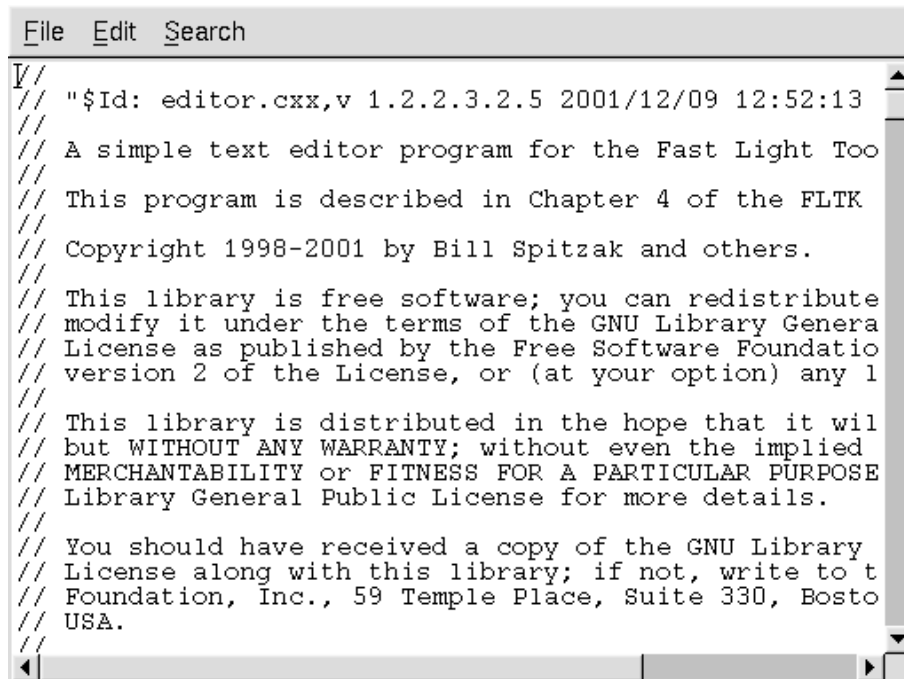


Figure 7.2: The completed editor window

7.12 Advanced Features

Now that we've implemented the basic functionality, it is time to show off some of the advanced features of the [Fl_Text_Editor](#) widget.

7.12.1 Syntax Highlighting

The [Fl_Text_Editor](#) widget supports highlighting of text with different fonts, colors, and sizes. The implementation is based on the excellent [NEdit](#) text editor core, which uses a parallel "style" buffer which tracks the font, color, and size of the text that is drawn.

Styles are defined using the [Fl_Text_Display::Style_Table_Entry](#) structure defined in `<FL/Fl_Text_Display.H>`:

```
struct Style_Table_Entry {  
    Fl_Color color;  
    Fl_Font font;
```

```

    int      size;
    unsigned attr;
};

```

The `color` member sets the color for the text, the `font` member sets the FLTK font index to use, and the `size` member sets the pixel size of the text. The `attr` member is currently not used.

For our text editor we'll define 7 styles for plain code, comments, keywords, and preprocessor directives:

```

Fl_Text_Display::Style_Table_Entry styletable[] = {      // Style table
{ FL_BLACK,      FL_COURIER,      FL_NORMAL_SIZE }, // A - Plain
{ FL_DARK_GREEN, FL_COURIER_ITALIC, FL_NORMAL_SIZE }, // B - Line comments
{ FL_DARK_GREEN, FL_COURIER_ITALIC, FL_NORMAL_SIZE }, // C - Block comments
{ FL_BLUE,       FL_COURIER,       FL_NORMAL_SIZE }, // D - Strings
{ FL_DARK_RED,   FL_COURIER,       FL_NORMAL_SIZE }, // E - Directives
{ FL_DARK_RED,   FL_COURIER_BOLD,  FL_NORMAL_SIZE }, // F - Types
{ FL_BLUE,       FL_COURIER_BOLD,  FL_NORMAL_SIZE }, // G - Keywords
};

```

You'll notice that the comments show a letter next to each style - each style in the style buffer is referenced using a character starting with the letter 'A'.

You call the `highlight_data()` method to associate the style data and buffer with the text editor widget:

```

Fl_Text_Buffer *stylebuf;

w->editor->highlight_data(stylebuf, styletable,
                          sizeof(styletable) / sizeof(styletable[0]),
                          'A', style_unfinished_cb, 0);

```

Finally, you need to add a callback to the main text buffer so that changes to the text buffer are mirrored in the style buffer:

```

textbuf->add_modify_callback(style_update, w->editor);

```

The `style_update()` function, like the `change_cb()` function described earlier, is called whenever text is added or removed from the text buffer. It mirrors the changes in the style buffer and then updates the style data as necessary:

```

//
// 'style_update()' - Update the style buffer...
//

void
style_update(int      pos,          // I - Position of update
             int      nInserted,    // I - Number of inserted chars
             int      nDeleted,     // I - Number of deleted chars
             int      nRestyled,    // I - Number of restyled chars
             const char *deletedText, // I - Text that was deleted
             void      *cbArg) {    // I - Callback data
    int  start,                      // Start of text
        end;                        // End of text
    char last,                      // Last style on line
        *style,                     // Style data
        *text;                      // Text data

    // If this is just a selection change, just unselect the style buffer...
    if (nInserted == 0 && nDeleted == 0) {
        stylebuf->unselect();
        return;
    }
}

```

```

    }

    // Track changes in the text buffer...
    if (nInserted > 0) {
        // Insert characters into the style buffer...
        style = new char[nInserted + 1];
        memset(style, 'A', nInserted);
        style[nInserted] = '\0';

        stylebuf->replace(pos, pos + nDeleted, style);
        delete[] style;
    } else {
        // Just delete characters in the style buffer...
        stylebuf->remove(pos, pos + nDeleted);
    }

    // Select the area that was just updated to avoid unnecessary
    // callbacks...
    stylebuf->select(pos, pos + nInserted - nDeleted);

    // Re-parse the changed region; we do this by parsing from the
    // beginning of the line of the changed region to the end of
    // the line of the changed region... Then we check the last
    // style character and keep updating if we have a multi-line
    // comment character...
    start = textbuf->line_start(pos);
    end   = textbuf->line_end(pos + nInserted - nDeleted);
    text  = textbuf->text_range(start, end);
    style = stylebuf->text_range(start, end);
    last  = style[end - start - 1];

    style_parse(text, style, end - start);

    stylebuf->replace(start, end, style);
    ((Fl_Text_Editor *)cbArg)->redisplay_range(start, end);

    if (last != style[end - start - 1]) {
        // The last character on the line changed styles, so reparse the
        // remainder of the buffer...
        free(text);
        free(style);

        end   = textbuf->length();
        text  = textbuf->text_range(start, end);
        style = stylebuf->text_range(start, end);

        style_parse(text, style, end - start);

        stylebuf->replace(start, end, style);
        ((Fl_Text_Editor *)cbArg)->redisplay_range(start, end);
    }

    free(text);
    free(style);
}

```

The `style_parse()` function scans a copy of the text in the buffer and generates the necessary style characters for display. It assumes that parsing begins at the start of a line:

```

//
// 'style_parse()' - Parse text and produce style data.
//

void
style_parse(const char *text,
            char       *style,

```

```

        int          length) {
char          current;
int           col;
int           last;
char          buf[255],
              *bufptr;
const char *temp;

for (current = *style, col = 0, last = 0; length > 0; length --, text ++) {
    if (current == 'A') {
        // Check for directives, comments, strings, and keywords...
        if (col == 0 && *text == '#') {
            // Set style to directive
            current = 'E';
        } else if (strncmp(text, "//", 2) == 0) {
            current = 'B';
        } else if (strncmp(text, "/*", 2) == 0) {
            current = 'C';
        } else if (strncmp(text, "\\\"", 2) == 0) {
            // Quoted quote...
            *style++ = current;
            *style++ = current;
            text ++;
            length --;
            col += 2;
            continue;
        } else if (*text == '\\') {
            current = 'D';
        } else if (!last && islower(*text)) {
            // Might be a keyword...
            for (temp = text, bufptr = buf;
                 islower(*temp) && bufptr < (buf + sizeof(buf) - 1);
                 *bufptr++ = *temp++);

            if (!islower(*temp)) {
                *bufptr = '\\0';

                bufptr = buf;

                if (bsearch(&bufptr, code_types,
                           sizeof(code_types) / sizeof(code_types[0]),
                           sizeof(code_types[0]), compare_keywords)) {
                    while (text < temp) {
                        *style++ = 'F';
                        text ++;
                        length --;
                        col ++;
                    }

                    text --;
                    length ++;
                    last = 1;
                    continue;
                } else if (bsearch(&bufptr, code_keywords,
                                   sizeof(code_keywords) / sizeof(code_keywords[0]),
                                   sizeof(code_keywords[0]), compare_keywords)) {
                    while (text < temp) {
                        *style++ = 'G';
                        text ++;
                        length --;
                        col ++;
                    }

                    text --;
                    length ++;
                    last = 1;
                    continue;
                }
            }
        }
    }
}

```

```
    }  
  }  
}  
} else if (current == 'C' && strncmp(text, "*/", 2) == 0) {  
  // Close a C comment...  
  *style++ = current;  
  *style++ = current;  
  text ++;  
  length --;  
  current = 'A';  
  col += 2;  
  continue;  
}  
} else if (current == 'D') {  
  // Continuing in string...  
  if (strncmp(text, "\\\"", 2) == 0) {  
    // Quoted end quote...  
    *style++ = current;  
    *style++ = current;  
    text ++;  
    length --;  
    col += 2;  
    continue;  
  } else if (*text == '\\') {  
    // End quote...  
    *style++ = current;  
    col ++;  
    current = 'A';  
    continue;  
  }  
}  
}  
  
// Copy style info...  
if (current == 'A' && (*text == '{' || *text == '}')) *style++ = 'G';  
else *style++ = current;  
col ++;  
  
last = isalnum(*text) || *text == '.';  
  
if (*text == '\\n') {  
  // Reset column and possibly reset the style  
  col = 0;  
  if (current == 'B' || current == 'E') current = 'A';  
}  
}  
}
```


Chapter 8

6 - Handling Events

This chapter discusses the FLTK event model and how to handle events in your program or widget.

8.1 The FLTK Event Model

Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application. Events can also come from other programs like the window manager.

Events are identified by the integer argument passed to a `handle()` method that overrides the `Fl_Widget::handle()` virtual method. Other information about the most recent event is stored in static locations and acquired by calling the `Fl::event_*` methods. This static information remains valid until the next event is read from the window system, so it is ok to look at it outside of the `handle()` method.

In the next chapter, the `MyClass::handle()` example shows how to override the `Fl_Widget::handle()` method to accept and process specific events.

8.2 Mouse Events

8.2.1 FL_PUSH

A mouse button has gone down with the mouse pointing at this widget. You can find out what button by calling `Fl::event_button()`. You find out the mouse position by calling `Fl::event_x()` and `Fl::event_y()`.

A widget indicates that it "wants" the mouse click by returning non-zero from its `handle()` method, as in the `MyClass::handle()` example. It will then become the `Fl::pushed()` widget and will get `FL_DRAG` and the matching `FL_RELEASE` events. If `handle()` returns zero then FLTK will try sending the `FL_PUSH` to another widget.

8.2.2 FL_DRAG

The mouse has moved with a button held down. The current button state is in `Fl::event_state()`. The mouse position is in `Fl::event_x()` and `Fl::event_y()`.

In order to receive `FL_DRAG` events, the widget must return non-zero when handling `FL_PUSH`.

8.2.3 FL_RELEASE

A mouse button has been released. You can find out what button by calling `Fl::event_button()`.

In order to receive the `FL_RELEASE` event, the widget must return non-zero when handling `FL_PUSH`.

8.2.4 FL_MOVE

The mouse has moved without any mouse buttons held down. This event is sent to the `Fl::belowmouse()` widget.

In order to receive `FL_MOVE` events, the widget must return non-zero when handling `FL_ENTER`.

8.2.5 FL_MOUSEWHEEL

The user has moved the mouse wheel. The `Fl::event_dx()` and `Fl::event_dy()` methods can be used to find the amount to scroll horizontally and vertically.

8.3 Focus Events

8.3.1 FL_ENTER

The mouse has been moved to point at this widget. This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its `handle()` method. It then becomes the `Fl::belowmouse()` widget and will receive `FL_MOVE` and `FL_LEAVE` events.

8.3.2 FL_LEAVE

The mouse has moved out of the widget.

In order to receive the `FL_LEAVE` event, the widget must return non-zero when handling `FL_ENTER`.

8.3.3 FL_FOCUS

This indicates an *attempt* to give a widget the keyboard focus.

If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its `handle()` method. It then becomes the `Fl::focus()` widget and gets `FL_KEYDOWN`, `FL_KEYUP`, and `FL_UNFOCUS` events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check `Fl::event_key()` to figure out why it moved. For navigation it will be the key pressed and interaction with the window manager it will be zero.

8.3.4 FL_UNFOCUS

This event is sent to the previous `Fl::focus()` widget when another widget gets the focus or the window loses focus.

8.4 Keyboard Events

8.4.1 FL_KEYDOWN, FL_KEYUP

A key was pressed or released. The key can be found in `Fl::event_key()`. The text that the key should insert can be found with `Fl::event_text()` and its length is in `Fl::event_length()`. If you use the key, then `handle()` should return 1. If you return zero then FLTK assumes you ignored the key and will then attempt to send it to a parent widget. If none of them want it, it will change the event into a `FL_SHORTCUT` event.

To receive `FL_KEYBOARD` events you must also respond to the `FL_FOCUS` and `FL_UNFOCUS` events.

If you are writing a text-editing widget you may also want to call the [Fl::compose\(\)](#) function to translate individual keystrokes into foreign characters.

`FL_KEYUP` events are sent to the widget that currently has focus. This is not necessarily the same widget that received the corresponding `FL_KEYDOWN` event because focus may have changed between events.

8.4.2 FL_SHORTCUT

If the [Fl::focus\(\)](#) widget is zero or ignores an `FL_KEYBOARD` event then FLTK tries sending this event to every widget it can, until one of them returns non-zero. `FL_SHORTCUT` is first sent to the [Fl::belowmouse\(\)](#) widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero. FLTK tries really hard to not to ignore any keystrokes!

You can also make "global" shortcuts by using [Fl::add_handler\(\)](#). A global shortcut will work no matter what windows are displayed or which one has the focus.

8.5 Widget Events

8.5.1 FL_DEACTIVATE

This widget is no longer active, due to [deactivate\(\)](#) being called on it or one of its parents. Please note that although [active\(\)](#) may still return true for this widget after receiving this event, it is only truly active if [active\(\)](#) is true for both it and all of its parents. (You can use [active_r\(\)](#) to check this).

8.5.2 FL_ACTIVATE

This widget is now active, due to [activate\(\)](#) being called on it or one of its parents.

8.5.3 FL_HIDE

This widget is no longer visible, due to [hide\(\)](#) being called on it or one of its parents, or due to a parent window being minimized. Please note that although [visible\(\)](#) may still return true for this widget after receiving this event, it is only truly visible if [visible\(\)](#) is true for both it and all of its parents. (You can use [visible_r\(\)](#) to check this).

8.5.4 FL_SHOW

This widget is visible again, due to [show\(\)](#) being called on it or one of its parents, or due to a parent window being restored. *A child [Fl_Window](#) will respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass `FL_SHOW` to the base class `handle()` method!*

8.6 Clipboard Events

8.6.1 FL_PASTE

You should get this event some time after you call [Fl::paste\(\)](#). The contents of [Fl::event_text\(\)](#) is the text to insert and the number of characters is in [Fl::event_length\(\)](#).

8.6.2 FL_SELECTIONCLEAR

The `Fl::selection_owner()` will get this event before the selection is moved to another widget. This indicates that some other widget or program has claimed the selection. Motif programs used this to clear the selection indication. Most modern programs ignore this.

8.7 Drag and Drop Events

FLTK supports drag and drop of text and files from any application on the desktop. Text is transferred using the current code page. Files are received as a list of full path and file names, separated by newline. On some platforms, path names are prepended with `file://`.

The drag and drop data is available in `Fl::event_text()` at the concluding `FL_PASTE`. On some platforms, the event text is also available for the `FL_DND_*` events, however application must not depend on that behavior because it depends on the protocol used on each platform.

`FL_DND_*` events cannot be used in widgets derived from `Fl_Group` or `Fl_Window`.

8.7.1 FL_DND_ENTER

The mouse has been moved to point at this widget. A widget that is interested in receiving drag'n'drop data must return 1 to receive `FL_DND_DRAG`, `FL_DND_LEAVE` and `FL_DND_RELEASE` events.

8.7.2 FL_DND_DRAG

The mouse has been moved inside a widget while dragging data. A widget that is interested in receiving drag'n'drop data should indicate the possible drop position.

8.7.3 FL_DND_LEAVE

The mouse has moved out of the widget.

8.7.4 FL_DND_RELEASE

The user has released the mouse button dropping data into the widget. If the widget returns 1, it will receive the data in the immediately following `FL_PASTE` event.

8.8 Fl::event_*() methods

FLTK keeps the information about the most recent event in static storage. This information is good until the next event is processed. Thus it is valid inside `handle()` and `callback()` methods.

These are all trivial inline functions and thus very fast and small:

- `Fl::event_button()`
- `Fl::event_clicks()`
- `Fl::event_dx()`

- [Fl::event_dy\(\)](#)
- [Fl::event_inside\(\)](#)
- [Fl::event_is_click\(\)](#)
- [Fl::event_key\(\)](#)
- [Fl::event_length\(\)](#)
- [Fl::event_state\(\)](#)
- [Fl::event_text\(\)](#)
- [Fl::event_x\(\)](#)
- [Fl::event_x_root\(\)](#)
- [Fl::event_y\(\)](#)
- [Fl::event_y_root\(\)](#)
- [Fl::get_key\(\)](#)
- [Fl::get_mouse\(\)](#)
- [Fl::test_shortcut\(\)](#)

8.9 Event Propagation

FLTK follows very simple and unchangeable rules for sending events. The major innovation is that widgets can indicate (by returning 0 from the `handle()` method) that they are not interested in an event, and FLTK can then send that event elsewhere. This eliminates the need for *"interests"* (event masks or tables), and this is probably the main reason FLTK is much smaller than other toolkits.

Most events are sent directly to the `handle()` method of the [Fl_Window](#) that the window system says they belong to. The window (actually the [Fl_Group](#) that [Fl_Window](#) is a subclass of) is responsible for sending the events on to any child widgets. To make the [Fl_Group](#) code somewhat easier, FLTK sends some events (`FL_DRAG`, `FL_RELEASE`, `FL_KEYBOARD`, `FL_SHORTCUT`, `FL_UNFOCUS`, and `FL_LEAVE`) directly to leaf widgets. These procedures control those leaf widgets:

- [Fl::add_handler\(\)](#)
- [Fl::belowmouse\(\)](#)
- [Fl::focus\(\)](#)
- [Fl::grab\(\)](#)
- [Fl::modal\(\)](#)
- [Fl::pushed\(\)](#)
- [Fl::release\(\)](#)
- [Fl_Widget::take_focus\(\)](#)

8.10 FLTK Compose-Character Sequences

Todo

Does Fltk Compose Character Sequences text need updating after the addition of UTF-8 handling to FLTK-1.3.x ?

The foreign-letter compose processing done by the [Fl_Input](#) widget's [compose](#) key handler is provided in a function that you can call if you are writing your own text editor widget.

FLTK uses its own compose processing to allow "preview" of the partially composed sequence, which is impossible with the usual "dead key" processing.

Although currently only characters in the ISO-8859-1 character set are handled, you should call this in case any enhancements to the processing are done in the future. The interface has been designed to handle arbitrary UTF-8 encoded text.

The following methods are provided for character composition:

- [Fl::compose\(\)](#)
- [Fl::compose_reset\(\)](#)

Chapter 9

7 - Adding and Extending Widgets

This chapter describes how to add your own widgets or extend existing widgets in FLTK.

9.1 Subclassing

New widgets are created by *subclassing* an existing FLTK widget, typically `Fl_Widget` for controls and `Fl_Group` for composite widgets.

A control widget typically interacts with the user to receive and/or display a value of some sort.

A composite widget holds a list of child widgets and handles moving, sizing, showing, or hiding them as needed. `Fl_Group` is the main composite widget class in FLTK, and all of the other composite widgets (`Fl_Pack`, `Fl_Scroll`, `Fl_Tabs`, `Fl_Tile`, and `Fl_Window`) are subclasses of it.

You can also subclass other existing widgets to provide a different look or user-interface. For example, the button widgets are all subclasses of `Fl_Button` since they all interact with the user via a mouse button click. The only difference is the code that draws the face of the button.

9.2 Making a Subclass of Fl_Widget

Your subclasses can directly descend from `Fl_Widget` or any subclass of `Fl_Widget`. `Fl_Widget` has only four virtual methods, and overriding some or all of these may be necessary.

9.3 The Constructor

The constructor should have the following arguments:

```
MyClass(int x, int y, int w, int h, const char *label = 0);
```

This will allow the class to be used in `FLUID` without problems.

The constructor must call the constructor for the base class and pass the same arguments:

```
MyClass::MyClass(int x, int y, int w, int h, const char *label)
: Fl_Widget(x, y, w, h, label) {
// do initialization stuff...
}
```

`Fl_Widget`'s protected constructor sets `x()`, `y()`, `w()`, `h()`, and `label()` to the passed values and initializes the other instance variables to:

```
type(0);
box(FL_NO_BOX);
color(FL_BACKGROUND_COLOR);
selection_color(FL_BACKGROUND_COLOR);
labeltype(FL_NORMAL_LABEL);
labelstyle(FL_NORMAL_STYLE);
labelsize(FL_NORMAL_SIZE);
labelcolor(FL_FOREGROUND_COLOR);
align(FL_ALIGN_CENTER);
callback(default_callback, 0);
flags(ACTIVE|VISIBLE);
image(0);
deimage(0);
```


9.4 Protected Methods of Fl_Widget

The following methods are provided for subclasses to use:

- `clear_visible()`
- `damage()`
- `draw_box()`
- `draw_focus()`
- `draw_label()`
- `set_flag()`
- `set_visible()`
- `test_shortcut()`
- `type()`

`void Fl_Widget::damage(uchar mask)`

`void Fl_Widget::damage(uchar mask, int x, int y, int w, int h)`

`uchar Fl_Widget::damage()`

The first form indicates that a partial update of the object is needed. The bits in `mask` are OR'd into `damage()`. Your `draw()` routine can examine these bits to limit what it is drawing. The public method `Fl_Widget::redraw()` simply does `Fl_Widget::damage(FL_DAMAGE_ALL)`, but the implementation of your widget can call the public `damage(n)`.

The second form indicates that a region is damaged. If only these calls are done in a window (no calls to `damage(n)`) then FLTK will clip to the union of all these calls before drawing anything. This can greatly speed up incremental displays. The mask bits are OR'd into `damage()` unless this is a `Fl_Window` widget.

The third form returns the bitwise-OR of all `damage(n)` calls done since the last `draw()`.

When redrawing your widgets you should look at the damage bits to see what parts of your widget need redrawing. The `handle()` method can then set individual damage bits to limit the amount of drawing that needs to be done:

```
MyClass::handle(int event) {
    ...
    if (change_to_part1) damage(1);
    if (change_to_part2) damage(2);
    if (change_to_part3) damage(4);
}

MyClass::draw() {
    if (damage() & FL_DAMAGE_ALL) {
        ... draw frame/box and other static stuff ...
    }
}
```

```

    if (damage() & (FL_DAMAGE_ALL | 1)) draw_part1();
    if (damage() & (FL_DAMAGE_ALL | 2)) draw_part2();
    if (damage() & (FL_DAMAGE_ALL | 4)) draw_part3();
}

```

Todo

Clarify `Fl_Window::damage(n)` handling - seems confused/wrong? ORing value doesn't match setting behaviour in `Fl_Widget.H`!

```
void Fl_Widget::draw_box() const
```

```
void Fl_Widget::draw_box(Fl_Boxtype t, Fl_Color c) const
```

The first form draws this widget's `box()`, using the dimensions of the widget. The second form uses `t` as the box type and `c` as the color for the box.

```
void Fl_Widget::draw_focus()
```

```
void Fl_Widget::draw_focus(Fl_Boxtype t, int x, int y, int w, int h) const
```

Draws a focus box inside the widget's bounding box. The second form allows you to specify a different bounding box.

```
void Fl_Widget::draw_label() const
```

```
void Fl_Widget::draw_label(int x, int y, int w, int h) const
```

```
void Fl_Widget::draw_label(int x, int y, int w, int h, Fl_Align align) const
```

The first form is the usual function for a `draw()` method to call to draw the widget's label. It does not draw the label if it is supposed to be outside the box (on the assumption that the enclosing group will draw those labels).

The second form uses the passed bounding box instead of the widget's bounding box. This is useful so "centered" labels are aligned with some feature, like a moving slider.

The third form draws the label anywhere. It acts as though `FL_ALIGN_INSIDE` has been forced on so the label will appear inside the passed bounding box. This is designed for parent groups to draw labels with.

```
void Fl_Widget::set_flag(int c)
```

Calling `set_flag(SHORTCUT_LABEL)` modifies the behavior of `draw_label()` so that `'&'` characters cause an underscore to be printed under the next letter.

```
void Fl_Widget::set_visible()
```

```
void Fl_Widget::clear_visible()
```

Fast inline versions of `Fl_Widget::hide()` and `Fl_Widget::show()`. These do not send the `FL_HIDE` and `FL_SHOW` events to the widget.

```
int Fl_Widget::test_shortcut()
static int Fl_Widget::test_shortcut(const char *s)
```

The first version tests `Fl_Widget::label()` against the current event (which should be a `FL_SHORTCUT` event). If the label contains a `'&'` character and the character after it matches the keypress, this returns true. This returns false if the `SHORTCUT_LABEL` flag is off, if the label is `NULL`, or does not have a `'&'` character in it, or if the keypress does not match the character.

The second version lets you do this test against an arbitrary string.

Todo

Clarify `Fl_Widget::test_shortcut()` explanations. `Fl_Widget.h` says Internal Use only, but subclassing chapter gives details!

```
uchar Fl_Widget::type() const
void Fl_Widget::type(uchar t)
```

The property `Fl_Widget::type()` can return an arbitrary 8-bit identifier, and can be set with the protected method `type(uchar t)`. This value had to be provided for Forms compatibility, but you can use it for any purpose you want. Try to keep the value less than 100 to not interfere with reserved values.

FLTK does not use RTTI (Run Time Typing Information), to enhance portability. But this may change in the near future if RTTI becomes standard everywhere.

If you don't have RTTI you can use the clumsy FLTK mechanism, by having `type()` use a unique value. These unique values must be greater than the symbol `FL_RESERVED_TYPE` (which is 100) and less than `FL_WINDOW` (unless you make a subclass of `Fl_Window`). Look through the header files for `FL_RESERVED_TYPE` to find an unused number. If you make a subclass of `Fl_Window` you must use `FL_WINDOW+n` (where `n` must be in the range 1 to 7).

9.5 Handling Events

The virtual method `Fl_Widget::handle(int event)` is called to handle each event passed to the widget. It can:

- Change the state of the widget.
- Call `Fl_Widget::redraw()` if the widget needs to be redisplayed.
- Call `Fl_Widget::damage(uchar c)` if the widget needs a partial-update (assuming you provide support for this in your `draw()` method).
- Call `Fl_Widget::do_callback()` if a callback should be generated.

- Call `Fl_Widget::handle()` on child widgets.

Events are identified by the integer argument. Other information about the most recent event is stored in static locations and acquired by calling the `Fl::event_*` methods. This information remains valid until another event is handled.

Here is a sample `handle()` method for a widget that acts as a pushbutton and also accepts the keystroke 'x' to cause the callback:

```
int MyClass::handle(int event) {
    switch(event) {
        case FL_PUSH:
            highlight = 1;
            redraw();
            return 1;
        case FL_DRAG: {
            int t = Fl::event_inside(this);
            if (t != highlight) {
                highlight = t;
                redraw();
            }
        }
        return 1;
        case FL_RELEASE:
            if (highlight) {
                highlight = 0;
                redraw();
                do_callback();
                // never do anything after a callback, as the callback
                // may delete the widget!
            }
            return 1;
        case FL_SHORTCUT:
            if (Fl::event_key() == 'x') {
                do_callback();
                return 1;
            }
            return 0;
        default:
            return Fl_Widget::handle(event);
    }
}
```

You must return non-zero if your `handle()` method uses the event. If you return zero, the parent widget will try sending the event to another widget.

9.6 Drawing the Widget

The `draw()` virtual method is called when FLTK wants you to redraw your widget. It will be called if and only if `damage()` is non-zero, and `damage()` will be cleared to zero after it returns. The `draw()` method should be declared protected so that it can't be called from non-drawing code.

The `damage()` value contains the bitwise-OR of all the `damage(n)` calls to this widget since it was last drawn. This can be used for minimal update, by only redrawing the parts whose bits are set. FLTK will turn on the `FL_DAMAGE_ALL` bit if it thinks the entire widget must be redrawn, e.g. for an expose event.

Expose events (and the `damage(mask,x,y,w,h)` function described above) will cause `draw()` to be called with FLTK's `clipping` turned on. You can greatly speed up redrawing in some cases by testing `fl_not_clipped(x,y,w,h)` or `fl_clip_box(...)` and skipping invisible parts.

Besides the protected methods described above, FLTK provides a large number of basic drawing functions, which are described in the [5 - Drawing Things in FLTK](#) chapter.

9.7 Resizing the Widget

The `resize(x, y, w, h)` method is called when the widget is being resized or moved. The arguments are the new position, width, and height. `x()`, `y()`, `w()`, and `h()` still remain the old size. You must call `resize()` on your base class with the same arguments to get the widget size to actually change.

This should *not* call `redraw()`, at least if only the `x()` and `y()` change. This is because composite widgets like [Fl_Scroll](#) may have a more efficient way of drawing the new position.

9.8 Making a Composite Widget

A "composite" widget contains one or more "child" widgets. To make a composite widget you should subclass [Fl_Group](#). It is possible to make a composite object that is not a subclass of [Fl_Group](#), but you'll have to duplicate the code in [Fl_Group](#) anyways.

Instances of the child widgets may be included in the parent:

```
class MyClass : public Fl_Group {
    Fl_Button the_button;
    Fl_Slider the_slider;
    ...
};
```

The constructor has to initialize these instances. They are automatically added to the group, since the [Fl_Group](#) constructor does [Fl_Group::begin\(\)](#). *Don't forget to call [Fl_Group::end\(\)](#) or use the [Fl_End](#) pseudo-class:*

```
MyClass::MyClass(int x, int y, int w, int h) :
    Fl_Group(x, y, w, h),
    the_button(x + 5, y + 5, 100, 20),
    the_slider(x, y + 50, w, 20)
{
    ... (you could add dynamically created child widgets here) ...
    end(); // don't forget to do this!
}
```

The child widgets need callbacks. These will be called with a pointer to the children, but the widget itself may be found in the `parent()` pointer of the child. Usually these callbacks can be static private methods, with a matching private method:

```
void MyClass::static_slider_cb(Fl_Widget* v, void *) { // static method
    ((MyClass*) (v->parent()))->slider_cb();
}
void MyClass::slider_cb() { // normal method
    use(the_slider->value());
}
```

If you make the `handle()` method, you can quickly pass all the events to the children using the [Fl_Group::handle\(\)](#) method. You don't need to override `handle()` if your composite widget does nothing other than pass events to the children:

```
int MyClass::handle(int event) {
    if (Fl_Group::handle(event)) return 1;
    ... handle events that children don't want ...
}
```

If you override `draw()` you need to draw all the children. If `redraw()` or `damage()` is called on a child, `damage(FL_DAMAGE_CHILD)` is done to the group, so this bit of `damage()` can be used to indicate that a child needs to be drawn. It is fastest if you avoid drawing anything else in this case:

```
int MyClass::draw() {
    Fl_Widget *const*a = array();
    if (damage() == FL_DAMAGE_CHILD) { // only redraw some children
        for (int i = children(); i --; a++) update_child(**a);
    } else { // total redraw
        ... draw background graphics ...
        // now draw all the children atop the background:
        for (int i = children_; i --; a++) {
            draw_child(**a);
            draw_outside_label(**a); // you may not need to do this
        }
    }
}
```

Todo

The new method `draw_children` should be documented below.

`Fl_Group` provides some protected methods to make drawing easier:

- `draw_child()`
- `draw_outside_label()`
- `update_child()`

```
void Fl_Group::draw_child(Fl_Widget &widget) const
```

This will force the child's `damage()` bits all to one and call `draw()` on it, then clear the `damage()`. You should call this on all children if a total redraw of your widget is requested, or if you draw something (like a background box) that damages the child. Nothing is done if the child is not `visible()` or if it is clipped.

```
void Fl_Group::draw_outside_label(Fl_Widget &widget) const
```

Draw the labels that are *not* drawn by `draw_label()`. If you want more control over the label positions you might want to call `child->draw_label(x, y, w, h, a)`.

```
void Fl_Group::update_child(Fl_Widget &widget)
```

Draws the child only if its `damage()` is non-zero. You should call this on all the children if your own damage is equal to `FL_DAMAGE_CHILD`. Nothing is done if the child is not `visible()` or if it is clipped.

9.9 Cut and Paste Support

FLTK provides routines to cut and paste 8-bit text (in the future this may be UTF-8) between applications:

- `Fl::paste()`
- `Fl::selection()`
- `Fl::selection_owner()`

It may be possible to cut/paste non-text data by using `Fl::add_handler()`.

9.10 Drag And Drop Support

FLTK provides routines to drag and drop 8-bit text between applications:

Drag'n'drop operations are initiated by copying data to the clipboard and calling the function `Fl::dnd()`.

Drop attempts are handled via the following events, already described under [Drag and Drop Events](#) in a previous chapter:

- `FL_DND_ENTER`
- `FL_DND_DRAG`
- `FL_DND_LEAVE`
- `FL_DND_RELEASE`
- `FL_PASTE`

9.11 Making a subclass of `Fl_Window`

You may want your widget to be a subclass of `Fl_Window`, `Fl_Double_Window`, or `Fl_Gl_Window`. This can be useful if your widget wants to occupy an entire window, and can also be used to take advantage of system-provided clipping, or to work with a library that expects a system window ID to indicate where to draw.

Subclassing `Fl_Window` is almost exactly like subclassing `Fl_Group`, and in fact you can easily switch a subclass back and forth. Watch out for the following differences:

1. `Fl_Window` is a subclass of `Fl_Group` so *make sure your constructor calls* `end()` unless you actually want children added to your window.
1. When handling events and drawing, the upper-left corner is at 0,0, not `x()`, `y()` as in other `Fl_Widget`'s. For instance, to draw a box around the widget, call `draw_box(0,0,w(),h())`, rather than `draw_box(x(),y(),w(),h())`.

You may also want to subclass `Fl_Window` in order to get access to different visuals or to change other attributes of the windows. See the [F - Operating System Issues](#) chapter for more information.

Chapter 10

8 - Using OpenGL

This chapter discusses using FLTK for your OpenGL applications.

10.1 Using OpenGL in FLTK

The easiest way to make an OpenGL display is to subclass `Fl_Gl_Window`. Your subclass must implement a `draw()` method which uses OpenGL calls to draw the display. Your main program should call `redraw()` when the display needs to change, and (somewhat later) FLTK will call `draw()`.

With a bit of care you can also use OpenGL to draw into normal FLTK windows. This allows you to use Gouraud shading for drawing your widgets. To do this you use the `gl_start()` and `gl_finish()` functions around your OpenGL code.

`gl_start()` and `gl_finish()` functions around your OpenGL code.

You must include FLTK's `<FL/gl.h>` header file. It will include the file `<GL/gl.h>`, define some extra drawing functions provided by FLTK, and include the `<windows.h>` header file needed by WIN32 applications.

10.2 Making a Subclass of Fl_Gl_Window

To make a subclass of `Fl_Gl_Window`, you must provide:

- A class definition.
- A `draw()` method.
- A `handle()` method if you need to receive input from the user.

If your subclass provides static controls in the window, they must be redrawn whenever the `FL_DAMAGE_ALL` bit is set in the value returned by `damage()`. For double-buffered windows you will need to surround the drawing code with the following code to make sure that both buffers are redrawn:

```
#ifndef MESA
glDrawBuffer(GL_FRONT_AND_BACK);
#endif // !MESA
... draw stuff here ...
#ifndef MESA
glDrawBuffer(GL_BACK);
#endif // !MESA
```

Note:

If you are using the Mesa graphics library, the call to `glDrawBuffer()` is not required and will slow down drawing considerably. The preprocessor instructions shown above will optimize your code based upon the graphics library used.

10.2.1 Defining the Subclass

To define the subclass you just subclass the `Fl_Gl_Window` class:

```
class MyWindow : public Fl_Gl_Window {
    void draw();
    int handle(int);
};
```

```
public:
    MyWindow(int X, int Y, int W, int H, const char *L)
        : Fl_Gl_Window(X, Y, W, H, L) {}
};
```

The `draw()` and `handle()` methods are described below. Like any widget, you can include additional private and public data in your class (such as scene graph information, etc.)

10.2.2 The `draw()` Method

The `draw()` method is where you actually do your OpenGL drawing:

```
void MyWindow::draw() {
    if (!valid()) {
        ... set up projection, viewport, etc ...
        ... window size is in w() and h().
        ... valid() is turned on by FLTK after draw() returns
    }
    ... draw ...
}
```

10.2.3 The `handle()` Method

The `handle()` method handles mouse and keyboard events for the window:

```
int MyWindow::handle(int event) {
    switch(event) {
        case FL_PUSH:
            ... mouse down event ...
            ... position in Fl::event_x() and Fl::event_y()
            return 1;
        case FL_DRAG:
            ... mouse moved while down event ...
            return 1;
        case FL_RELEASE:
            ... mouse up event ...
            return 1;
        case FL_FOCUS :
        case FL_UNFOCUS :
            ... Return 1 if you want keyboard events, 0 otherwise
            return 1;
        case FL_KEYBOARD:
            ... keypress, key is in Fl::event_key(), ascii in Fl::event_text()
            ... Return 1 if you understand/use the keyboard event, 0 otherwise...
            return 1;
        case FL_SHORTCUT:
            ... shortcut, key is in Fl::event_key(), ascii in Fl::event_text()
            ... Return 1 if you understand/use the shortcut event, 0 otherwise...
            return 1;
        default:
            // pass other events to the base class...
            return Fl_Gl_Window::handle(event);
    }
}
```

When `handle()` is called, the OpenGL context is not set up! If your display changes, you should call `redraw()` and let `draw()` do the work. Don't call any OpenGL drawing functions from inside `handle()`!

You can call *some* OpenGL stuff like hit detection and texture loading functions by doing:

```

case FL_PUSH:
    make_current();          // make OpenGL context current
    if (!valid()) {

        ... set up projection exactly the same as draw ...

        valid(1);           // stop it from doing this next time
    }
    ... ok to call NON-DRAWING OpenGL code here, such as hit
    detection, loading textures, etc...

```

Your main program can now create one of your windows by doing `new MyWindow(...)`.

You can also use your new window class in **FLUID** by:

1. Putting your class definition in a `MyWindow.H` file.
2. Creating a **FL_Box** widget in FLUID.
3. In the widget panel fill in the "class" field with `MyWindow`. This will make FLUID produce constructors for your new class.
4. In the "Extra Code" field put `#include "MyWindow.H"`, so that the FLUID output file will compile.

You must put `glwindow->show()` in your main code after calling `show()` on the window containing the OpenGL window.

10.3 Using OpenGL in Normal FLTK Windows

You can put OpenGL code into the `draw()` method, as described in [Drawing the Widget](#) in the previous chapter, or into the code for a **boxtype** or other places with some care.

Fl_Widget::draw() method or into the code for a **boxtype** or other places with some care.

Most importantly, before you show *any* windows, including those that don't have OpenGL drawing, you **must** initialize FLTK so that it knows it is going to use OpenGL. You may use any of the symbols described for `Fl_Gl_Window::mode()` to describe how you intend to use OpenGL:

```
Fl::gl_visual(FL_RGB);
```

You can then put OpenGL drawing code anywhere you can draw normally by surrounding it with:

```

gl_start();
... put your OpenGL code here ...
gl_finish();

```

`gl_start()` and `gl_finish()` set up an OpenGL context with an orthographic projection so that 0,0 is the lower-left corner of the window and each pixel is one unit. The current clipping is reproduced with OpenGL `glScissor()` commands. These functions also synchronize the OpenGL graphics stream with the drawing done by other X, WIN32, or FLTK functions.

The same context is reused each time. If your code changes the projection transformation or anything else you should use `glPushMatrix()` and `glPopMatrix()` functions to put the state back before calling `gl_finish()`.

You may want to use `Fl_Window::current() -> h()` to get the drawable height so that you can flip the Y coordinates.

Unfortunately, there are a bunch of limitations you must adhere to for maximum portability:

- You must choose a default visual with `Fl::gl_visual()`.
- You cannot pass `FL_DOUBLE` to `Fl::gl_visual()`.
- You cannot use `Fl_Double_Window` or `Fl_Overlay_Window`.

Do *not* call `gl_start()` or `gl_finish()` when drawing into an `Fl_Gl_Window` !

10.4 OpenGL Drawing Functions

FLTK provides some useful OpenGL drawing functions. They can be freely mixed with any OpenGL calls, and are defined by including `<FL/gl.h>` which you should include instead of the OpenGL header `<GL/gl.h>`.

```
void gl_color(Fl_Color)
```

Sets the current OpenGL color to a FLTK color. *For color-index modes it will use `fl_xpixel(c)`, which is only right if this window uses the default colormap!*

```
void gl_rect(int x, int y, int w, int h)
```

```
void gl_rectf(int x, int y, int w, int h)
```

Outlines or fills a rectangle with the current color. If `Fl_Gl_Window::ortho()` has been called, then the rectangle will exactly fill the pixel rectangle passed.

```
void gl_font(Fl_Font fontid, int size)
```

Sets the current OpenGL font to the same font you get by calling `fl_font()`.

```
int gl_height()
```

```
int gl_descent()
```

```
float gl_width(const char *)
```

```
float gl_width(const char *, int n)
```

```
float gl_width(uchar)
```

Returns information about the current OpenGL font.

```
void gl_draw(const char *)
```

```
void gl_draw(const char *, int n)
```

Draws a nul-terminated string or an array of $n <$ characters in the current OpenGL font at the current raster position.

```
void gl_draw(const char *, int x, int y)
void gl_draw(const char *, int n, int x, int y)
void gl_draw(const char *, float x, float y)
void gl_draw(const char *, int n, float x, float y)
```

Draws a nul-terminated string or an array of n characters in the current OpenGL font at the given position.

```
void gl_draw(const char *, int x, int y, int w, int h, Fl_Align)
```

Draws a string formatted into a box, with newlines and tabs expanded, other control characters changed to ^X, and aligned with the edges or center. Exactly the same output as `fl_draw()`.

10.5 Speeding up OpenGL

Performance of [Fl_Gl_Window](#) may be improved on some types of OpenGL implementations, in particular MESA and other software emulators, by setting the `GL_SWAP_TYPE` environment variable. This variable declares what is in the backbuffer after you do a swapbuffers.

- `setenv GL_SWAP_TYPE COPY`

This indicates that the back buffer is copied to the front buffer, and still contains it's old data. This is true of many hardware implementations. Setting this will speed up emulation of overlays, and widgets that can do partial update can take advantage of this as `damage()` will not be cleared to -1.

- `setenv GL_SWAP_TYPE NODAMAGE`

This indicates that nothing changes the back buffer except drawing into it. This is true of MESA and Win32 software emulation and perhaps some hardware emulation on systems with lots of memory.

- All other values for `GL_SWAP_TYPE`, and not setting the variable, cause FLTK to assume that the back buffer must be completely redrawn after a swap.

This is easily tested by running the `gl_overlay` demo program and seeing if the display is correct when you drag another window over it or if you drag the window off the screen and back on. You have to exit and run the program again for it to see any changes to the environment variable.

10.6 Using OpenGL Optimizer with FLTK

[OpenGL Optimizer](#) is a scene graph toolkit for OpenGL available from Silicon Graphics for IRIX and Microsoft Windows. It allows you to view large scenes without writing a lot of OpenGL code.

OptimizerWindow Class Definition

To use `OpenGL Optimizer` with FLTK you'll need to create a subclass of `Fl_Gl_Widget` that includes several state variables:

```
class OptimizerWindow : public Fl_Gl_Window {
    csContext *context_; // Initialized to 0 and set by draw()...
    csDrawAction *draw_action_; // Draw action...
    csGroup *scene_; // Scene to draw...
    csCamara *camera_; // Viewport for scene...

    void draw();

public:
    OptimizerWindow(int X, int Y, int W, int H, const char *L)
        : Fl_Gl_Window(X, Y, W, H, L) {
        context_ = (csContext *)0;
        draw_action_ = (csDrawAction *)0;
        scene_ = (csGroup *)0;
        camera_ = (csCamera *)0;
    }

    void scene(csGroup *g) { scene_ = g; redraw(); }

    void camera(csCamera *c) {
        camera_ = c;
        if (context_) {
            draw_action_>setCamera(camera_);
            camera_>draw(draw_action_);
            redraw();
        }
    }
};
```

The camera() Method

The `camera()` method sets the camera (projection and viewpoint) to use when drawing the scene. The scene is redrawn after this call.

The draw() Method

The `draw()` method performs the needed initialization and does the actual drawing:

```
void OptimizerWindow::draw() {
    if (!context_) {
        // This is the first time we've been asked to draw; create the
        // Optimizer context for the scene...

#ifdef WIN32
        context_ = new csContext((HDC)fl_getHDC());
        context_>ref();
        context_>makeCurrent((HDC)fl_getHDC());
#else
        context_ = new csContext(fl_display, fl_visual);
        context_>ref();
        context_>makeCurrent(fl_display, fl_window);
#endif // WIN32

        ... perform other context setup as desired ...
    }
}
```

```

    // Then create the draw action to handle drawing things...

    draw_action_ = new csDrawAction;
    if (camera_) {
        draw_action_>setCamera(camera_);
        camera_>draw(draw_action_);
    }
    } else {
#ifdef WIN32
        context_>makeCurrent((HDC)fl_getHDC());
    #else
        context_>makeCurrent(fl_display, fl_window);
    #endif // WIN32
    }

    if (!valid()) {
        // Update the viewport for this context...
        context_>setViewport(0, 0, w(), h());
    }

    // Clear the window...
    context_>clear(csContext::COLOR_CLEAR | csContext::DEPTH_CLEAR,
                  0.0f,          // Red
                  0.0f,          // Green
                  0.0f,          // Blue
                  1.0f);         // Alpha

    // Then draw the scene (if any)...
    if (scene_)
        draw_action_>apply(scene_);
}

```

The scene() Method

The `>scene()` method sets the scene to be drawn. The scene is a collection of 3D objects in a `csGroup`. The scene is redrawn after this call.

Chapter 11

9 - Programming with FLUID

This chapter shows how to use the Fast Light User-Interface Designer ("FLUID") to create your GUIs.

Subchapters:

- [What is FLUID?](#)
- [Running FLUID Under UNIX](#)
- [Running FLUID Under Microsoft Windows](#)
- [Compiling .fl files](#)
- [A Short Tutorial](#)
- [FLUID Reference](#)
- [Internationalization with FLUID](#)
- [Known limitations](#)

11.1 What is FLUID?

The Fast Light User Interface Designer, or FLUID, is a graphical editor that is used to produce FLTK source code. FLUID edits and saves its state in `.fl` files. These files are text, and you can (with care) edit them in a text editor, perhaps to get some special effects.

FLUID can "compile" the `.fl` file into a `.cxx` and a `.h` file. The `.cxx` file defines all the objects from the `.fl` file and the `.h` file declares all the global ones. FLUID also supports localization ([Internationalization](#)) of label strings using message files and the GNU gettext or POSIX catgets interfaces.

A simple program can be made by putting all your code (including a `main()` function) into the `.fl` file and thus making the `.cxx` file a single source file to compile. Most programs are more complex than this, so you write other `.cxx` files that call the FLUID functions. These `.cxx` files must `#include` the `.h` file or they can `#include` the `.cxx` file so it still appears to be a single source file.



Figure 11.1: FLUID organization

Normally the FLUID file defines one or more functions or classes which output C++ code. Each function defines a one or more FLTK windows, and all the widgets that go inside those windows.

Widgets created by FLUID are either "named", "complex named" or "unnamed". A named widget has a legal C++ variable identifier as its name (i.e. only alphanumeric and underscore). In this case FLUID defines a global variable or class member that will point at the widget after the function defining it is called. A complex named object has punctuation such as '`.`' or '`->`' or any other symbols in its name. In this case FLUID assigns a pointer to the widget to the name, but does not attempt to declare it. This can be used to get the widgets into structures. An unnamed widget has a blank name and no pointer is stored.

Widgets may either call a named callback function that you write in another source file, or you can supply a small piece of C++ source and FLUID will write a private callback function into the `.cxx` file.

11.2 Running FLUID Under UNIX

To run FLUID under UNIX, type:

```
fluid filename.fl &
```

to edit the `.fl` file `filename.fl`. If the file does not exist you will get an error pop-up, but if you dismiss it you will be editing a blank file of that name. You can run FLUID without any name, in which case you will be editing an unnamed blank setup (but you can use save-as to write it to a file).

You can provide any of the standard FLTK switches before the filename:

```
-display host:n.n
-geometry WxH+X+Y
-title windowtitle
-name classname
-iconic
-fg color
-bg color
-bg2 color
-scheme schemename
```

Changing the colors may be useful to see what your interface will look at if the user calls it with the same switches. Similarly, using `"-scheme plastic"` will show how the interface will look using the "plastic" scheme.

In the current version, if you don't put FLUID into the background with `' & '` then you will be able to abort FLUID by typing `CTRL-C` on the terminal. It will exit immediately, losing any changes.

11.3 Running FLUID Under Microsoft Windows

To run FLUID under WIN32, double-click on the *FLUID.exe* file. You can also run FLUID from the Command Prompt window. FLUID always runs in the background under WIN32.

11.4 Compiling .fl files

FLUID can also be called as a command-line "compiler" to create the `.cxx` and `.h` file from a `.fl` file. To do this type:

```
fluid -c filename.fl
```

This will read the `filename.fl` file and write `filename.cxx` and `filename.h`. Any leading directory on `filename.fl` will be stripped, so they are always written to the current directory. If there are any errors reading or writing the files, FLUID will print the error and exit with a non-zero code. You can use the following lines in a makefile to automate the creation of the source and header files:

```
my_panels.h my_panels.cxx: my_panels.fl
    fluid -c my_panels.fl
```

Most versions of make support rules that cause `.fl` files to be compiled:

```
.SUFFIXES: .fl .cxx .h
.fl.h .fl.cxx:
    fluid -c $<
```

11.5 A Short Tutorial

FLUID is an amazingly powerful little program. However, this power comes at a price as it is not always obvious how to accomplish seemingly simple tasks with it. This tutorial will show you how to generate a complete user interface class with FLUID that is used for the CubeView program provided with FLTK.

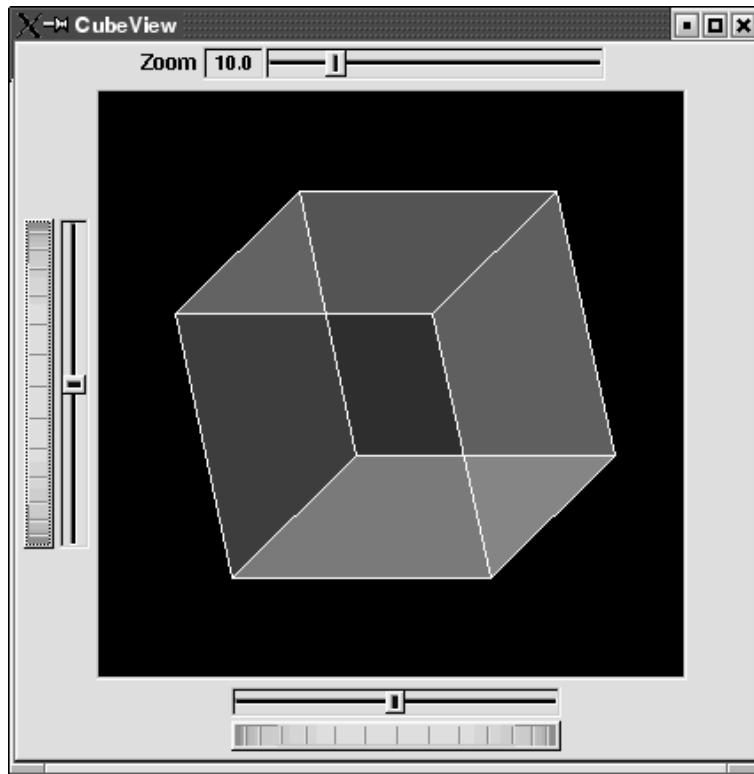


Figure 11.2: CubeView demo

The window is of class `CubeViewUI`, and is completely generated by FLUID, including class member functions. The central display of the cube is a separate subclass of `FL_GL_Window` called `CubeView`. `CubeViewUI` manages `CubeView` using callbacks from the various sliders and rollers to manipulate the viewing angle and zoom of `CubeView`.

At the completion of this tutorial you will (hopefully) understand how to:

1. Use FLUID to create a complete user interface class, including constructor and any member functions necessary.
2. Use FLUID to set callbacks member functions of a custom widget classes.
3. Subclass an `FL_GL_Window` to suit your purposes.

11.5.1 The CubeView Class

The `CubeView` class is a subclass of `FL_GL_Window`. It has methods for setting the zoom, the x and y pan, and the rotation angle about the x and y axes.

You can safely skip this section as long as you realize the CubeView is a subclass of [Fl_Gl_Window](#) and will respond to calls from CubeViewUI, generated by FLUID.

The CubeView Class Definition

Here is the CubeView class definition, as given by its header file "test/CubeView.h":

```
class CubeView : public Fl_Gl_Window {
public:
    CubeView(int x,int y,int w,int h,const char *l=0);
    // this value determines the scaling factor used to draw the cube.
    double size;
    /* Set the rotation about the vertical (y ) axis.

This function is called by the horizontal roller in CubeViewUI
and the initialize button in CubeViewUI.
    */
    void v_angle(float angle){vAng=angle;};
    // Return the rotation about the vertical (y ) axis.
    float v_angle(){return vAng;};
    /* Set the rotation about the horizontal (x ) axis.

This function is called by the vertical roller in CubeViewUI
and the
initialize button in CubeViewUI.
    */
    void h_angle(float angle){hAng=angle;};
    // the rotation about the horizontal (x ) axis.
    float h_angle(){return hAng;};
    /* Sets the x shift of the cube view camera.

This function is called by the slider in CubeViewUI and the
initialize button in CubeViewUI.
    */
    void panx(float x){xshift=x;};
    /* Sets the y shift of the cube view camera.

This function is called by the slider in CubeViewUI and the
initialize button in CubeViewUI.
    */
    void pany(float y){yshift=y;};
    /* The widget class draw() override.
The draw() function initialize Gl for another round of
drawing then calls specialized functions for drawing each
of the entities displayed in the cube view.
    */
    void draw();

private:
    /* Draw the cube boundaries
Draw the faces of the cube using the boxv[] vertices, using
GL_LINE_LOOP for the faces. The color is #defined by
CUBECOLOR.
    */
    void drawCube();

    float vAng,hAng; float xshift,yshift;

    float boxv0[3];float boxv1[3]; float boxv2[3];float boxv3[3];
    float boxv4[3];float boxv5[3]; float boxv6[3];float boxv7[3];
};
```

The CubeView Class Implementation

Here is the CubeView implementation. It is very similar to the "cube" demo included with FLTK.

```
#include "CubeView.h"
#include <math.h>

CubeView::CubeView(int x,int y,int w,int h,const char *l)
    : Fl_Gl_Window(x,y,w,h,l)
{
    vAng = 0.0; hAng=0.0; size=10.0;
    /* The cube definition. These are the vertices of a unit cube
    centered on the origin.*/
    boxv0[0] = -0.5; boxv0[1] = -0.5; boxv0[2] = -0.5; boxv1[0] = 0.5;
    boxv1[1] = -0.5; boxv1[2] = -0.5; boxv2[0] = 0.5; boxv2[1] = 0.5;
    boxv2[2] = -0.5; boxv3[0] = -0.5; boxv3[1] = 0.5; boxv3[2] = -0.5;
    boxv4[0] = -0.5; boxv4[1] = -0.5; boxv4[2] = 0.5; boxv5[0] = 0.5;
    boxv5[1] = -0.5; boxv5[2] = 0.5; boxv6[0] = 0.5; boxv6[1] = 0.5;
    boxv6[2] = 0.5; boxv7[0] = -0.5; boxv7[1] = 0.5; boxv7[2] = 0.5;
};

// The color used for the edges of the bounding cube.
#define CUBECOLOR 255,255,255,255

void CubeView::drawCube() {
    /* Draw a colored cube */
    #define ALPHA 0.5
    glShadeModel(GL_FLAT);

    glBegin(GL_QUADS);
    glColor4f(0.0, 0.0, 1.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv1);
    glVertex3fv(boxv2);
    glVertex3fv(boxv3);

    glColor4f(1.0, 1.0, 0.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv4);
    glVertex3fv(boxv5);
    glVertex3fv(boxv1);

    glColor4f(0.0, 1.0, 1.0, ALPHA);
    glVertex3fv(boxv2);
    glVertex3fv(boxv6);
    glVertex3fv(boxv7);
    glVertex3fv(boxv3);

    glColor4f(1.0, 0.0, 0.0, ALPHA);
    glVertex3fv(boxv4);
    glVertex3fv(boxv5);
    glVertex3fv(boxv6);
    glVertex3fv(boxv7);

    glColor4f(1.0, 0.0, 1.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv3);
    glVertex3fv(boxv7);
    glVertex3fv(boxv4);

    glColor4f(0.0, 1.0, 0.0, ALPHA);
    glVertex3fv(boxv1);
    glVertex3fv(boxv5);
    glVertex3fv(boxv6);
    glVertex3fv(boxv2);
}
```

```

    glEnd();

    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINES);
        glVertex3fv(boxv0);
        glVertex3fv(boxv1);

        glVertex3fv(boxv1);
        glVertex3fv(boxv2);

        glVertex3fv(boxv2);
        glVertex3fv(boxv3);

        glVertex3fv(boxv3);
        glVertex3fv(boxv0);

        glVertex3fv(boxv4);
        glVertex3fv(boxv5);

        glVertex3fv(boxv5);
        glVertex3fv(boxv6);

        glVertex3fv(boxv6);
        glVertex3fv(boxv7);

        glVertex3fv(boxv7);
        glVertex3fv(boxv4);

        glVertex3fv(boxv0);
        glVertex3fv(boxv4);

        glVertex3fv(boxv1);
        glVertex3fv(boxv5);

        glVertex3fv(boxv2);
        glVertex3fv(boxv6);

        glVertex3fv(boxv3);
        glVertex3fv(boxv7);
    glEnd();
}; //drawCube

void CubeView::draw() {
    if (!valid()) {
        glLoadIdentity(); glViewport(0,0,w(),h());
        glOrtho(-10,10,-10,10,-20000,10000); glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix(); glTranslatef(xshift, yshift, 0);
    glRotatef(hAng,0,1,0); glRotatef(vAng,1,0,0);
    glScalef(float(size),float(size),float(size)); drawCube();
    glPopMatrix();
};

```

11.5.2 The CubeViewUI Class

We will completely construct a window to display and control the CubeView defined in the previous section using FLUID.

Defining the CubeViewUI Class

Once you have started FLUID, the first step in defining a class is to create a new class within FLUID using the **New->Code->Class** menu item. Name the class "CubeViewUI" and leave the subclass blank. We do not need any inheritance for this window. You should see the new class declaration in the FLUID browser window.

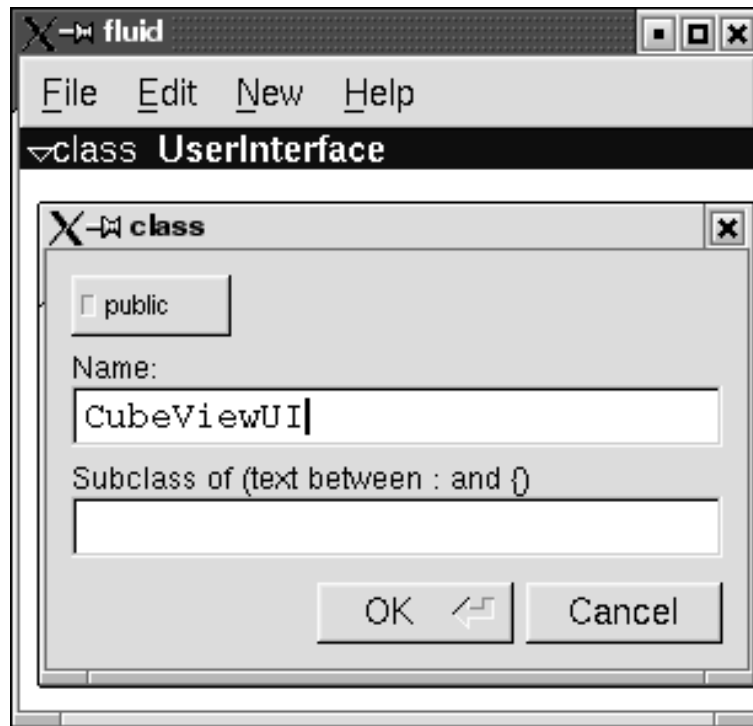


Figure 11.3: FLUID file for CubeView

Adding the Class Constructor

Click on the CubeViewUI class in the FLUID window and add a new method by selecting **New->Code->Function/Method**. The name of the function will also be CubeViewUI. FLUID will understand that this will be the constructor for the class and will generate the appropriate code. Make sure you declare the constructor public.

Then add a window to the CubeViewUI class. Highlight the name of the constructor in the FLUID browser window and click on **New->Group->Window**. In a similar manner add the following to the CubeViewUI constructor:

- A horizontal roller named `hrot`
- A vertical roller named `vrot`
- A horizontal slider named `xpan`
- A vertical slider named `ypan`
- A horizontal value slider named `zoom`

None of these additions need be public. And they shouldn't be unless you plan to expose them as part of the interface for CubeViewUI.

When you are finished you should have something like this:

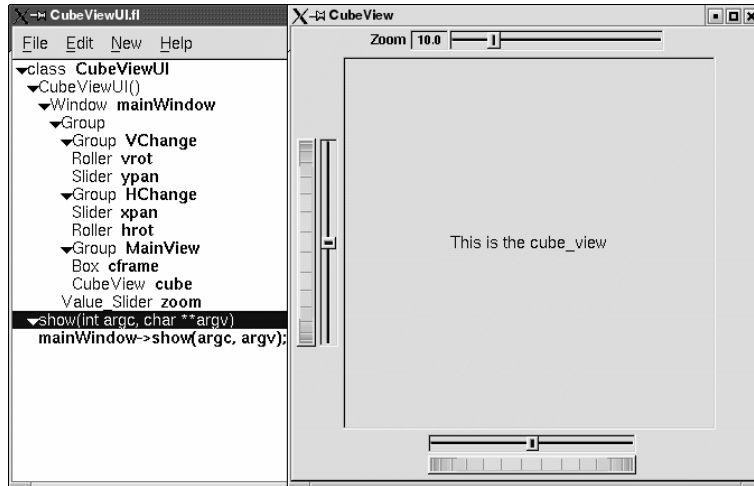


Figure 11.4: FLUID window containing CubeView demo

We will talk about the `show()` method that is highlighted shortly.

Adding the CubeView Widget

What we have is nice, but does little to show our cube. We have already defined the CubeView class and we would like to show it within the CubeViewUI.

The CubeView class inherits the [FL_GI_Window](#) class, which is created in the same way as a [FL_Box](#) widget. Use **New->Other->Box** to add a square box to the main window. This will be no ordinary box, however.

The Box properties window will appear. The key to letting CubeViewUI display CubeView is to enter CubeView in the **Class:** text entry box. This tells FLUID that it is not an [FL_Box](#), but a similar widget with the same constructor.

In the **Extra Code:** field enter `#include "CubeView.h"`

This `#include` is important, as we have just included CubeView as a member of CubeViewUI, so any public CubeView methods are now available to CubeViewUI.

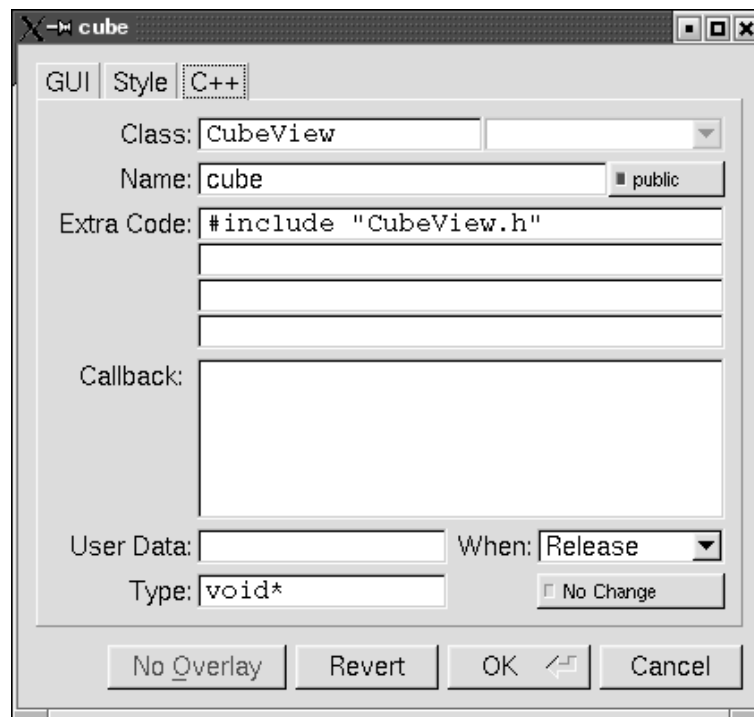


Figure 11.5: CubeView methods

Defining the Callbacks

Each of the widgets we defined before adding CubeView can have callbacks that call CubeView methods. You can call an external function or put in a short amount of code in the **Callback** field of the widget panel. For example, the callback for the ypan slider is:

```
cube->pany((Fl_Slider *)o)->value();
cube->redraw();
```

We call `cube->redraw()` after changing the value to update the CubeView window. CubeView could easily be modified to do this, but it is nice to keep this exposed in the case where you may want to do more than one view change only redrawing once saves a lot of time.

There is no reason no wait until after you have added CubeView to enter these callbacks. FLUID assumes you are smart enough not to refer to members or functions that don't exist.

Adding a Class Method

You can add class methods within FLUID that have nothing to do with the GUI. An an example add a show function so that CubeViewUI can actually appear on the screen.

Make sure the top level CubeViewUI is selected and select **New->Code->Function/Method**. Just use the name `show()`. We don't need a return value here, and since we will not be adding any widgets to this method FLUID will assign it a return type of `void`.

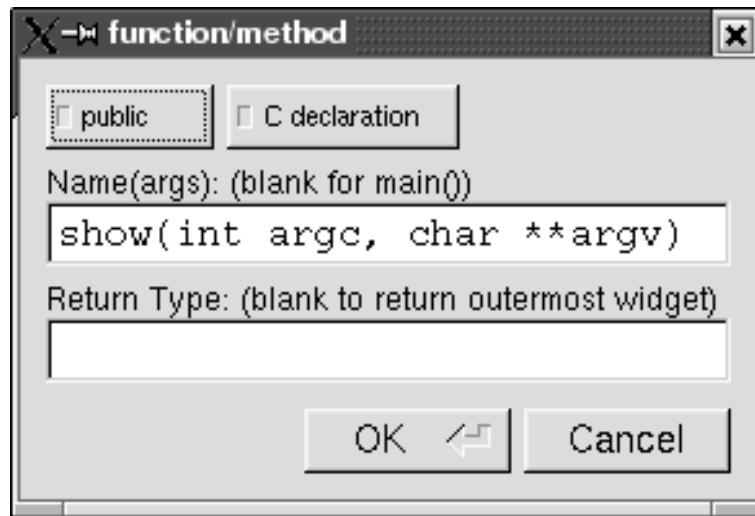


Figure 11.6: CubeView constructor

Once the new method has been added, highlight its name and select **New->Code->Code**. Enter the method's code in the code window.

11.5.3 Adding Constructor Initialization Code

If you need to add code to initialize class, for example setting initial values of the horizontal and vertical angles in the CubeView, you can simply highlight the Constructor and select **New->Code->Code**. Add any required code.

11.5.4 Generating the Code

Now that we have completely defined the CubeViewUI, we have to generate the code. There is one last trick to ensure this all works. Open the preferences dialog from **Edit->Preferences**.

At the bottom of the preferences dialog box is the key: "Include Header from Code". Select that option and set your desired file extensions and you are in business. You can include the CubeViewUI.h (or whatever extension you prefer) as you would any other C++ class.

11.6 FLUID Reference

The following sections describe each of the windows in FLUID.

11.6.1 The Widget Browser

The main window shows a menu bar and a scrolling browser of all the defined widgets. The name of the .fl file being edited is shown in the window title.

The widgets are stored in a hierarchy. You can open and close a level by clicking the "triangle" at the left of a widget. The leftmost widgets are the *parents*, and all the widgets listed below them are their *children*. Parents don't have to have any children.

The top level of the hierarchy is composed of *functions* and *classes*. Each of these will produce a single C++ public function or class in the output `.cxx` file. Calling the function or instantiating the class will create all of the child widgets.

The second level of the hierarchy contains the *windows*. Each of these produces an instance of class `FL_Window`.

Below that are either *widgets* (subclasses of `FL_Widget`) or *groups* of widgets (including other groups). Plain groups are for layout, navigation, and resize purposes. *Tab groups* provide the well-known file-card tab interface.

Widgets are shown in the browser by either their *name* (such as "main_panel" in the example), or by their *type* and *label* (such as "Button "the green"").

You *select* widgets by clicking on their names, which highlights them (you can also select widgets from any displayed window). You can select many widgets by dragging the mouse across them, or by using Shift+Click to toggle them on and off. To select no widgets, click in the blank area under the last widget. Note that hidden children may be selected even when there is no visual indication of this.

You *open* widgets by double-clicking on them, or (to open several widgets you have picked) by typing the F1 key. A control panel will appear so you can change the widget(s).

11.6.2 Menu Items

The menu bar at the top is duplicated as a pop-up menu on any displayed window. The shortcuts for all the menu items work in any window. The menu items are:

File/Open... (Ctrl+o)

Discards the current editing session and reads in a different `.fl` file. You are asked for confirmation if you have changed the current file.

FLUID can also read `.fd` files produced by the Forms and XForms "fdesign" programs. It is best to File/Merge them instead of opening them. FLUID does not understand everything in a `.fd` file, and will print a warning message on the controlling terminal for all data it does not understand. You will probably need to edit the resulting setup to fix these errors. Be careful not to save the file without changing the name, as FLUID will write over the `.fd` file with its own format, which fdesign cannot read!

File/Insert... (Ctrl+i)

Inserts the contents of another `.fl` file, without changing the name of the current `.fl` file. All the functions (even if they have the same names as the current ones) are added, and you will have to use cut/paste to put the widgets where you want.

File/Save (Ctrl+s)

Writes the current data to the `.fl` file. If the file is unnamed then FLUID will ask for a filename.

File/Save As... (Ctrl+Shift+S)

Asks for a new filename and saves the file.

File/Write Code (Ctrl+Shift+C)

"Compiles" the data into a `.cxx` and `.h` file. These are exactly the same as the files you get when you run FLUID with the `-c` switch.

The output file names are the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".h" or ".cxx" appended.

File/Write Strings (Ctrl+Shift+W)

Writes a message file for all of the text labels defined in the current file.

The output file name is the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".txt", ".po", or ".msg" appended depending on the [Internationalization Mode](#).

File/Quit (Ctrl+q)

Exits FLUID. You are asked for confirmation if you have changed the current file.

Edit/Undo (Ctrl+z)

This isn't implemented yet. You should do save often so you can recover from any mistakes you make.

Edit/Cut (Ctrl+x)

Deletes the selected widgets and all of their children. These are saved to a "clipboard" file and can be pasted back into any FLUID window.

Edit/Copy (Ctrl+c)

Copies the selected widgets and all of their children to the "clipboard" file.

Edit/Paste (Ctrl+v)

Pastes the widgets from the clipboard file.

If the widget is a window, it is added to whatever function is selected, or contained in the current selection.

If the widget is a normal widget, it is added to whatever window or group is selected. If none is, it is added to the window or group that is the parent of the current selection.

To avoid confusion, it is best to select exactly one widget before doing a paste.

Cut/paste is the only way to change the parent of a widget.

Edit/Select All (Ctrl+a)

Selects all widgets in the same group as the current selection.

If they are all selected already then this selects all widgets in that group's parent. Repeatedly typing Ctrl+a will select larger and larger groups of widgets until everything is selected.

Edit/Open... (F1 or double click)

Displays the current widget in the attributes panel. If the widget is a window and it is not visible then the window is shown instead.

Edit/Sort

Sorts the selected widgets into left to right, top to bottom order. You need to do this to make navigation keys in FLTK work correctly. You may then fine-tune the sorting with "Earlier" and "Later". This does not affect the positions of windows or functions.

Edit/Earlier (F2)

Moves all of the selected widgets one earlier in order among the children of their parent (if possible). This will affect navigation order, and if the widgets overlap it will affect how they draw, as the later widget is drawn on top of the earlier one. You can also use this to reorder functions, classes, and windows within functions.

Edit/Later (F3)

Moves all of the selected widgets one later in order among the children of their parent (if possible).

Edit/Group (F7)

Creates a new [FL_Group](#) and make all the currently selected widgets children of it.

Edit/Ungroup (F8)

Deletes the parent group if all the children of a group are selected.

Edit/Overlays on/off (Ctrl+Shift+O)

Toggles the display of the red overlays off, without changing the selection. This makes it easier to see box borders and how the layout looks. The overlays will be forced back on if you change the selection.

Edit/Project Settings... (Ctrl+p)

Displays the project settings panel. The output filenames control the extensions or names of the files the are generated by FLUID. If you check the "Include .h from .cxx" button the code file will include the header file automatically.

The [internationalization](#) options are described later in this chapter.

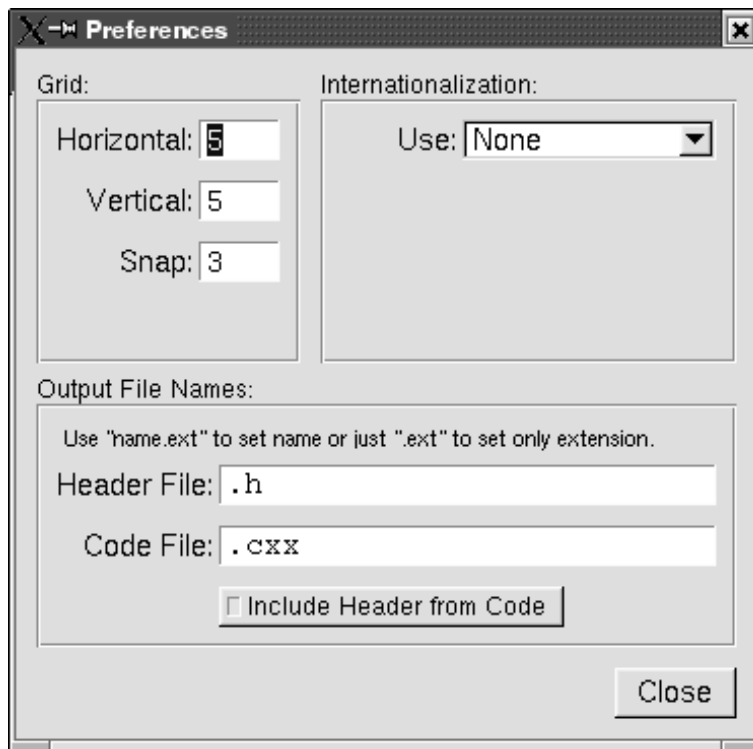


Figure 11.7: FLUID Preferences Window

Edit/GUI Settings... (Shift+Ctrl+p)

Displays the GUI settings panel. This panel is used to control the user interface settings.

New/Code/Function

Creates a new C function. You will be asked for a name for the function. This name should be a legal C++ function template, without the return type. You can pass arguments which can be referred to by code you type into the individual widgets.

If the function contains any unnamed windows, it will be declared as returning a `FL_Window` pointer. The unnamed window will be returned from it (more than one unnamed window is useless). If the function contains only named windows, it will be declared as returning nothing (`void`).

It is possible to make the `.cxx` output be a self-contained program that can be compiled and executed. This is done by deleting the function name so `main(argc, argv)` is used. The function will call `show()` on all the windows it creates and then call `Fl::run()`. This can also be used to test resize behavior or other parts of the user interface.

You can change the function name by double-clicking on the function.

New/Window

Creates a new `FL_Window` widget. The window is added to the currently selected function, or to the function containing the currently selected item. The window will appear, sized to 100x100. You can resize it to whatever size you require.

The widget panel will also appear and is described later in this chapter.

New/...

All other items on the New menu are subclasses of `FL_Widget`. Creating them will add them to the currently selected group or window, or the group or window containing the currently selected widget. The initial dimensions and position are chosen by copying the current widget, if possible.

When you create the widget you will get the widget's control panel, which is described later in this chapter.

Layout/Align/...

Align all selected widgets to the first widget in the selection.

Layout/Space Evenly/...

Space all selected widgets evenly inside the selected space. Widgets will be sorted from first to last.

Layout/Make Same Size/...

Make all selected widgets the same size as the first selected widget.

Layout/Center in Group/...

Center all selected widgets relative to their parent widget

Layout/Grid... (Ctrl+g)

Displays the grid settings panel. This panel controls the grid that all widgets snap to when you move and resize them, and for the "snap" which is how far a widget has to be dragged from its original position to actually change.

Shell/Execute Command... (Alt+x)

Displays the shell command panel. The shell command is commonly used to run a 'make' script to compile the FLTK output.

Shell/Execute Again (Alt+g)

Run the shell command again.

Help/About FLUID

Pops up a panel showing the version of FLUID.

Help/On FLUID

Shows this chapter of the manual.

Help/Manual

Shows the contents page of the manual

11.6.3 The Widget Panel

When you double-click on a widget or a set of widgets you will get the "widget attribute panel".

When you change attributes using this panel, the changes are reflected immediately in the window. It is useful to hit the "no overlay" button (or type Ctrl+Shift+O) to hide the red overlay so you can see the widgets more accurately, especially when setting the box type.

If you have several widgets selected, they may have different values for the fields. In this case the value for *one* of the widgets is shown. But if you change this value, *all* of the selected widgets are changed to the new value.

Hitting "OK" makes the changes permanent. Selecting a different widget also makes the changes permanent. FLUID checks for simple syntax errors such as mismatched parenthesis in any code before saving any text.

"Revert" or "Cancel" put everything back to when you last brought up the panel or hit OK. However in the current version of FLUID, changes to "visible" attributes (such as the color, label, box) are not undone by revert or cancel. Changes to code like the callbacks are undone, however.

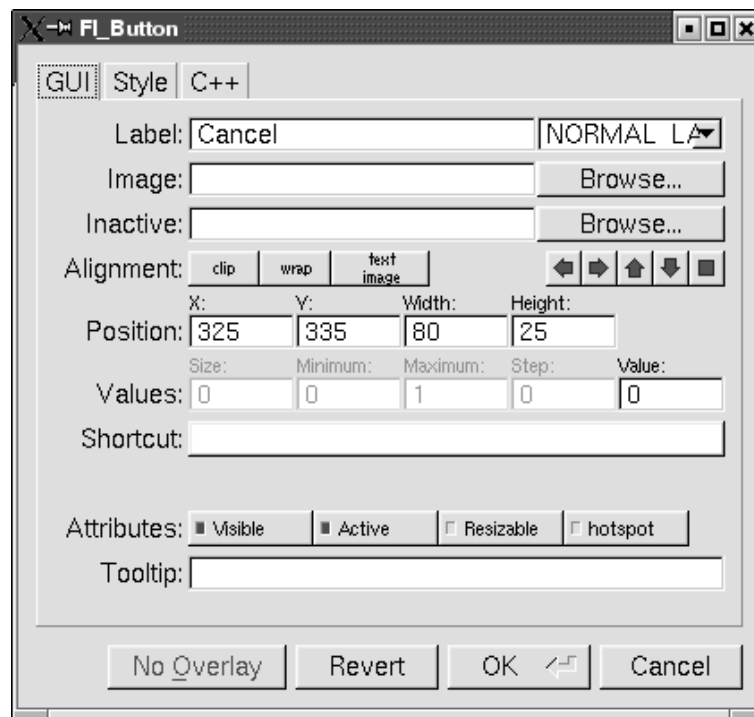


Figure 11.8: The FLUID widget GUI attributes

11.7 GUI Attributes

Label (text field)

String to print next to or inside the button. You can put newlines into the string to make multiple lines. The easiest way is by typing Ctrl+j.

Symbols can be added to the label using the at sign ("@").

Label (pull down menu)

How to draw the label. Normal, shadowed, engraved, and embossed change the appearance of the text.

Image

The active image for the widget. Click on the **Browse...** button to pick an image file using the file chooser.

Inactive

The inactive image for the widget. Click on the **Browse...** button to pick an image file using the file chooser.

Alignment (buttons)

Where to draw the label. The arrows put it on that side of the widget, you can combine the to put it in the corner. The "box" button puts the label inside the widget, rather than outside.

The **clip** button clips the label to the widget box, the **wrap** button wraps any text in the label, and the **text image** button puts the text over the image instead of under the image.

Position (text fields)

The position fields show the current position and size of the widget box. Enter new values to move and/or resize a widget.

Values (text fields)

The values and limits of the current widget. Depending on the type of widget, some or all of these fields may be inactive.

Shortcut

The shortcut key to activate the widget. Click on the shortcut button and press any key sequence to set the shortcut.

Attributes (buttons)

The **Visible** button controls whether the widget is visible (on) or hidden (off) initially. Don't change this for windows or for the immediate children of a Tabs group.

The **Active** button controls whether the widget is activated (on) or deactivated (off) initially. Most widgets appear greyed out when deactivated.

The **Resizable** button controls whether the window is resizable. In addition all the size changes of a window or group will go "into" the resizable child. If you have a large data display surrounded by buttons, you probably want that data area to be resizable. You can get more complex behavior by making invisible boxes the resizable widget, or by using hierarchies of groups. Unfortunately the only way to test it is to compile the program. Resizing the FLUID window is *not* the same as what will happen in the user program.

The **Hotspot** button causes the parent window to be positioned with that widget centered on the mouse. This position is determined *when the FLUID function is called*, so you should call it immediately before showing the window. If you want the window to hide and then reappear at a new position, you should have your program set the hotspot itself just before `show()`.

The **Border** button turns the window manager border on or off. On most window managers you will have to close the window and reopen it to see the effect.

X Class (text field)

The string typed into here is passed to the X window manager as the class. This can change the icon or window decorations. On most (all?) window managers you will have to close the window and reopen it to see the effect.

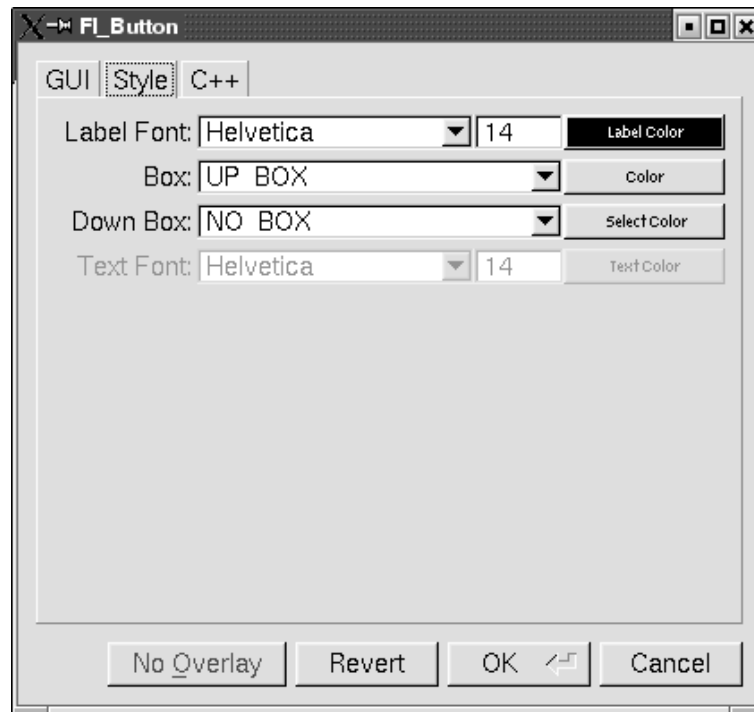


Figure 11.9: The FLUID widget Style attributes

11.7.1 Style Attributes

Label Font (pulldown menu)

Font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. Your program can change the actual font used by these "slots" in case you want some font other than the 16 provided.

Label Size (pulldown menu)

Pixel size (height) for the font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. To see the result without dismissing the panel, type the new number and then Tab.

Label Color (button)

Color to draw the label. Ignored by pixmaps (bitmaps, however, do use this color as the foreground color).

Box (pulldown menu)

The boxtype to draw as a background for the widget.

Many widgets will work, and draw faster, with a "frame" instead of a "box". A frame does not draw the colored interior, leaving whatever was already there visible. Be careful, as FLUID may draw this ok but the real program may leave unwanted stuff inside the widget.

If a window is filled with child widgets, you can speed up redrawing by changing the window's box type to "NO_BOX". FLUID will display a checkerboard for any areas that are not colored in by boxes. Note that this checkerboard is not drawn by the resulting program. Instead random garbage will be displayed.

Down Box (pulldown menu)

The boxtype to draw when a button is pressed or for some parts of other widgets like scrollbars and valuat-
tors.

Color (button)

The color to draw the box with.

Select Color (button)

Some widgets will use this color for certain parts. FLUID does not always show the result of this: this is the color buttons draw in when pushed down, and the color of input fields when they have the focus.

Text Font, Size, and Color

Some widgets display text, such as input fields, pull-down menus, and browsers.

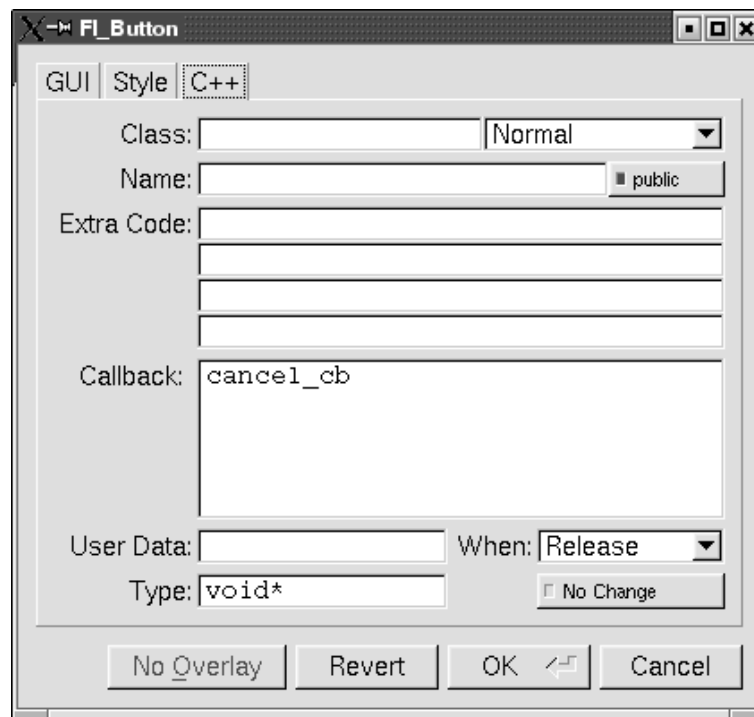


Figure 11.10: The FLUID widget C++ attributes

11.7.2 C++ Attributes

Class

This is how you use your own subclasses of [Fl_Widget](#). Whatever identifier you type in here will be the class that is instantiated.

In addition, no `#include` header file is put in the `.h` file. You must provide a `#include` line as the first line of the "Extra Code" which declares your subclass.

The class must be similar to the class you are spoofing. It does not have to be a subclass. It is sometimes useful to change this to another FLTK class. Currently the only way to get a double-buffered window is to change this field for the window to "Fl_Double_Window" and to add

```
\#include <FL/Fl_Double_Window.h>
```

to the extra code.

Type (upper-right pulldown menu)

Some classes have subtypes that modify their appearance or behavior. You pick the subtype off of this menu.

Name (text field)

Name of a variable to declare, and to store a pointer to this widget into. This variable will be of type "`<class>*`". If the name is blank then no variable is created.

You can name several widgets with "`name[0]`", "`name[1]`", "`name[2]`", etc. This will cause FLUID to declare an array of pointers. The array is big enough that the highest number found can be stored. All widgets that in the array must be the same type.

Public (button)

Controls whether the widget is publicly accessible. When embedding widgets in a C++ class, this controls whether the widget is `public` or `private` in the class. Otherwise it controls whether the widget is declared `static` or global (`extern`).

Extra Code (text fields)

These four fields let you type in literal lines of code to dump into the `.h` or `.cxx` files.

If the text starts with a `#` or the word `extern` then FLUID thinks this is an "include" line, and it is written to the `.h` file. If the same include line occurs several times then only one copy is written.

All other lines are "code" lines. The current widget is pointed to by the local variable `o`. The window being constructed is pointed to by the local variable `w`. You can also access any arguments passed to the function here, and any named widgets that are before this one.

FLUID will check for matching parenthesis, braces, and quotes, but does not do much other error checking. Be careful here, as it may be hard to figure out what widget is producing an error in the compiler. If you need more than four lines you probably should call a function in your own `.cxx` code.

Callback (text field)

This can either be the name of a function, or a small snippet of code. If you enter anything but letters, numbers, and the underscore then FLUID treats it as code.

A name names a function in your own code. It must be declared as `void name(<class>*, void*)`.

A code snippet is inserted into a static function in the `.cxx` output file. The function prototype is `void name(class *o, void *v)` so that you can refer to the widget as `o` and the `user_data()` as `v`. FLUID will check for matching parenthesis, braces, and quotes, but does not do much other error checking. Be careful here, as it may be hard to figure out what widget is producing an error in the compiler.

If the callback is blank then no callback is set.

User Data (text field)

This is a value for the `user_data()` of the widget. If blank the default value of zero is used. This can be any piece of C code that can be cast to a `void` pointer.

Type (text field)

The `void*` in the callback function prototypes is replaced with `this`. You may want to use `long` for old XForms code. Be warned that anything other than `void*` is not guaranteed to work! However on most architectures other pointer types are ok, and `long` is usually ok, too.

When (pulldown menu)

When to do the callback. This can be **Never**, **Changed**, **Release**, or **Enter Key**. The value of **Enter Key** is only useful for text input fields.

There are other rare but useful values for the `when()` field that are not in the menu. You should use the extra code fields to put these values in.

No Change (button)

The **No Change** button means the callback is done on the matching event even if the data is not changed.

11.8 Selecting and Moving Widgets

Double-clicking a window name in the browser will display it, if not displayed yet. From this display you can select widgets, sets of widgets, and move or resize them. To close a window either double-click it or type `ESC`.

To select a widget, click it. To select several widgets drag a rectangle around them. Holding down shift will toggle the selection of the widgets instead.

You cannot pick hidden widgets. You also cannot choose some widgets if they are completely overlapped by later widgets. Use the browser to select these widgets.

The selected widgets are shown with a red "overlay" line around them. You can move the widgets by dragging this box. Or you can resize them by dragging the outer edges and corners. Hold down the `Alt` key while dragging the mouse to defeat the snap-to-grid effect for fine positioning.

If there is a tab box displayed you can change which child is visible by clicking on the file tabs. The child you pick is selected.

The arrow, tab, and shift+tab keys "navigate" the selection. Left, right, tab, or shift+tab move to the next or previous widgets in the hierarchy. Hit the right arrow enough and you will select every widget in the window. Up/down widgets move to the previous/next widgets that overlap horizontally. If the navigation does not seem to work you probably need to "Sort" the widgets. This is important if you have input fields, as FLTK uses the same rules when using arrow keys to move between input fields.

To "open" a widget, double click it. To open several widgets select them and then type `F1` or pick "Edit/Open" off the pop-up menu.

Type `Ctrl+o` to temporarily toggle the overlay off without changing the selection, so you can see the widget borders.

You can resize the window by using the window manager border controls. FLTK will attempt to round the window size to the nearest multiple of the grid size and makes it big enough to contain all the widgets (it does this using illegal X methods, so it is possible it will barf with some window managers!). Notice that the actual window in your program may not be resizable, and if it is, the effect on child widgets may be different.

The panel for the window (which you get by double-clicking it) is almost identical to the panel for any other [Fl_Widget](#). There are three extra items:

11.9 Image Labels

The *contents* of the image files in the **Image** and **Inactive** text fields are written to the `.cxx` file. If many widgets share the same image then only one copy is written. Since the image data is embedded in the generated source code, you need only distribute the C++ code and not the image files themselves.

However, the *filenames* are stored in the `.fl` file so you will need the image files as well to read the `.fl` file. Filenames are relative to the location of the `.fl` file and not necessarily the current directory. We recommend you either put the images in the same directory as the `.fl` file, or use absolute path names.

Notes for All Image Types

FLUID runs using the default visual of your X server. This may be 8 bits, which will give you dithered images. You may get better results in your actual program by adding the code `Fl::visual(FL_RGB)` to your code right before the first window is displayed.

All widgets with the same image on them share the same code and source X pixmap. Thus once you have put an image on a widget, it is nearly free to put the same image on many other widgets.

If you edit an image at the same time you are using it in FLUID, the only way to convince FLUID to read the image file again is to remove the image from all widgets that are using it or re-load the `.fl` file.

Don't rely on how FLTK crops images that are outside the widget, as this may change in future versions! The cropping of inside labels will probably be unchanged.

To more accurately place images, make a new "box" widget and put the image in that as the label.

XBM (X Bitmap) Files

FLUID reads X bitmap files which use C source code to define a bitmap. Sometimes they are stored with the ".h" or ".bm" extension rather than the standard ".xpm" extension.

FLUID writes code to construct an `Fl_Bitmap` image and use it to label the widget. The '1' bits in the bitmap are drawn using the label color of the widget. You can change this color in the FLUID widget attributes panel. The '0' bits are transparent.

The program "bitmap" on the X distribution does an adequate job of editing bitmaps.

XPM (X Pixmap) Files

FLUID reads X pixmap files as used by the `libxpm` library. These files use C source code to define a pixmap. The filenames usually have the ".xpm" extension.

FLUID writes code to construct an `Fl_Pixmap` image and use it to label the widget. The label color of the widget is ignored, even for 2-color images that could be a bitmap. XPM files can mark a single color as being transparent, and FLTK uses this information to generate a transparency mask for the image.

We have not found any good editors for small iconic pictures. For pixmaps we have used `XPaint` and the KDE icon editor.

BMP Files

FLUID reads Windows BMP image files which are often used in WIN32 applications for icons. FLUID converts BMP files into (modified) XPM format and uses a [Fl_BMP_Image](#) image to label the widget. Transparency is handled the same as for XPM files. All image data is uncompressed when written to the source file, so the code may be much bigger than the .bmp file.

GIF Files

FLUID reads GIF image files which are often used in HTML documents to make icons. FLUID converts GIF files into (modified) XPM format and uses a [Fl_GIF_Image](#) image to label the widget. Transparency is handled the same as for XPM files. All image data is uncompressed when written to the source file, so the code may be much bigger than the .gif file. Only the first image of an animated GIF file is used.

JPEG Files

If FLTK is compiled with JPEG support, FLUID can read JPEG image files which are often used for digital photos. FLUID uses a [Fl_JPEG_Image](#) image to label the widget, and writes uncompressed RGB or grayscale data to the source file.

PNG (Portable Network Graphics) Files

If FLTK is compiled with PNG support, FLUID can read PNG image files which are often used in HTML documents. FLUID uses a [Fl_PNG_Image](#) image to label the widget, and writes uncompressed RGB or grayscale data to the source file. PNG images can provide a full alpha channel for partial transparency, and FLTK supports this as best as possible on each platform.

11.10 Internationalization with FLUID

FLUID supports internationalization (I18N for short) of label strings used by widgets. The preferences window (`Ctrl+p`) provides access to the I18N options.

11.10.1 I18N Methods

FLUID supports three methods of I18N: use none, use GNU gettext, and use POSIX catgets. The "use none" method is the default and just passes the label strings as-is to the widget constructors.

The "GNU gettext" method uses GNU gettext (or a similar text-based I18N library) to retrieve a localized string before calling the widget constructor.

The "POSIX catgets" method uses the POSIX catgets function to retrieve a numbered message from a message catalog before calling the widget constructor.

11.10.2 Using GNU gettext for I18N

FLUID's code support for GNU gettext is limited to calling a function or macro to retrieve the localized label; you still need to call `setlocale()` and `textdomain()` or `bindtextdomain()` to select the appropriate language and message file.

To use GNU gettext for I18N, open the preferences window and choose "GNU gettext" from the "Use" chooser. Two new input fields will then appear to control the include file and function/macro name to use when retrieving the localized label strings.

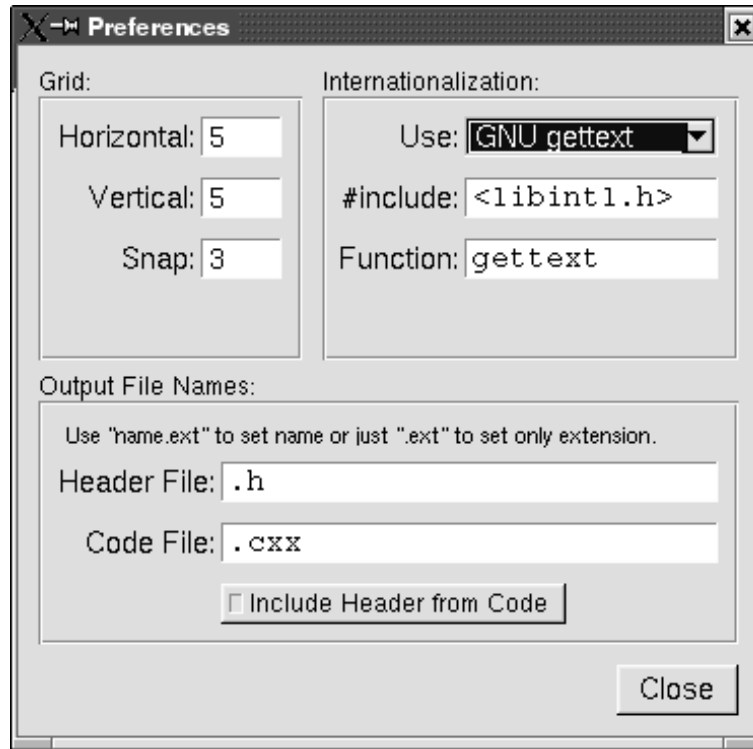


Figure 11.11: Internationalization using GNU gettext

The `#include` field controls the header file to include for I18N; by default this is `<libintl.h>`, the standard I18N file for GNU gettext.

The "Function" field controls the function (or macro) that will retrieve the localized message; by default the `gettext` function will be called.

11.10.3 Using POSIX catgets for I18N

FLUID's code support for POSIX catgets allows you to use a global message file for all interfaces or a file specific to each `.fl` file; you still need to call `setlocale()` to select the appropriate language.

To use POSIX catgets for I18N, open the preferences window and choose "POSIX catgets" from the "Use" chooser. Three new input fields will then appear to control the include file, catalog file, and set number for retrieving the localized label strings.

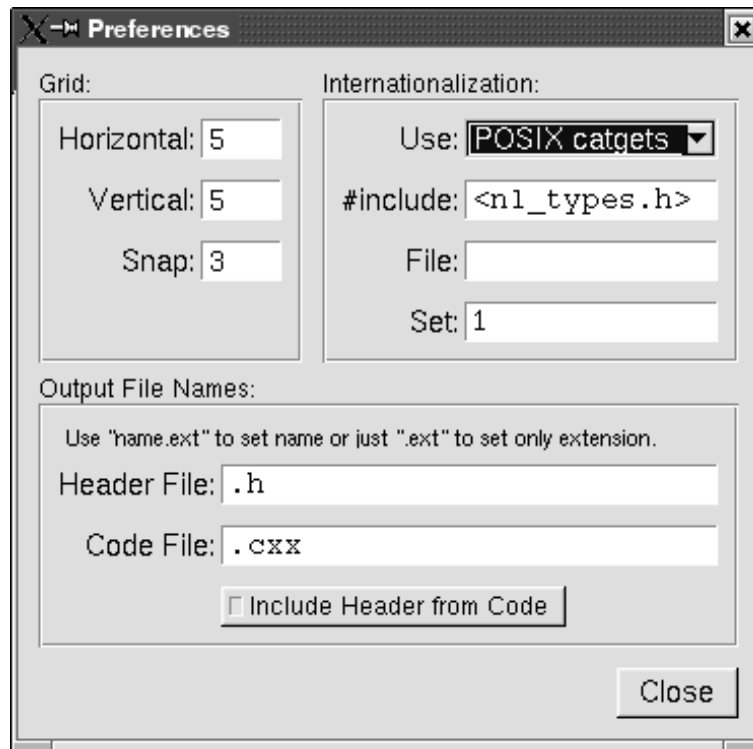


Figure 11.12: Internationalization using POSIX catgets

The `#include` field controls the header file to include for I18N; by default this is `<nl_types.h>`, the standard I18N file for POSIX catgets.

The "File" field controls the name of the catalog file variable to use when retrieving localized messages; by default the file field is empty which forces a local (static) catalog file to be used for all of the windows defined in your `.fl` file.

The "Set" field controls the set number in the catalog file. The default set is 1 and rarely needs to be changed.

11.11 Known limitations

Declaration Blocks can be used to temporarily block out already designed code using `#if 0` and `#endif` type construction. This will effectively avoid compilation of blocks of code. However, static code and data generated by this segment (menu items, images, include statements, etc.) will still be generated and likely cause compile-time warnings.

Chapter 12

10 - Advanced FLTK

This chapter explains advanced programming and design topics that will help you to get the most out of FLTK.

12.1 Multithreading

FLTK supports multithreaded application using a locking mechanism based on "threads". We do not provide a threading interface as part of the library. However a simple example how threads can be implemented for all supported platforms can be found in `test/threads.h` and `test/threads.cxx`.

To use the locking mechanism, FLTK must be compiled with `-enable-threads` set during the configure process. IDE-based versions of FLTK are automatically compiled with locking enabled if possible.

In `main()`, call `Fl::lock()` before `Fl::run()` or `Fl::wait()` to start the runtime multithreading support for your program. All callbacks and derived functions like `handle()` and `draw()` will now be properly locked:

```
int main() {
    Fl::lock();
    /* run thread */
    while (Fl::wait() > 0) {
        if (Fl::thread_message()) {
            /* process your data */
        }
    }
}
```

You can now start as many threads as you like. From within a thread (other than the main thread) FLTK calls must be wrapped with calls to `Fl::lock()` and `Fl::unlock()`:

```
Fl::lock();      // avoid conflicting calls
...             // your code here
Fl::unlock();    // allow other threads to access FLTK again
```

You can send messages from child threads to the main thread using `Fl::awake(void* message)`:

```
void *msg;       // "msg" is a pointer to your message
Fl::awake(msg);  // send "msg" to main thread
```

You can also tell the main thread to call a function for you as soon as possible by using `Fl::awake(Fl_Awake_Handler cb, void* userdata)`:

```
void do_something(void *userdata) {
    // running with the main thread
}

// running in another thread
void *data;          // "data" is a pointer to your user data
Fl::awake(do_something, data); // call something in main thread
```

FLTK supports multiple platforms, some of them which do not allow any other but the main thread to handle system events and open or close windows. The safe thing to do is to adhere to the following rules for threads on all operating systems:

- Don't `show()` or `hide()` anything that contains widgets derived from `Fl_Window`, including dialogs, file choosers, subwindows or `Fl_Gl_Window`'s

- Don't call `Fl::wait()`, `Fl::flush()` or any related methods that will handle system messages
- Don't start or cancel timers
- Don't change window decorations or titles
- The `make_current()` method may or may not work well for regular windows, but should always work for `Fl_Gl_Window`'s to allow for high speed rendering on graphics cards with multiple pipelines

See also: `Fl::awake(void* message)`, `Fl::lock()`, `Fl::thread_message()`, `Fl::unlock()`.

Chapter 13

11 - Unicode and utf-8 Support

This chapter explains how FLTK handles international text via Unicode and utf-8.

Unicode support was only recently added to FLTK and is still incomplete. This chapter is Work in Progress, reflecting the current state of Unicode support.

13.1 About Unicode and utf-8

The Unicode Standard is a worldwide accepted character encoding standard. Unicode provides access to over 100,000 characters used in all the major languages written today.

Utf-8 encodes all Unicode characters into variable length sequences of bytes. Unicode characters in the 7-bit ASCII range map to the same value in utf-8, making the transformation to Unicode quick and easy.

Moving from ASCII encoding to Unicode will allow all new FLTK applications to be easily internationalized and used all over the world. By choosing utf-8 encoding, FLTK remains largely source-code compatible to previous iteration of the library.

13.2 Unicode in FLTK

FLTK will be entirely converted to Unicode in utf-8 encoding. If a different encoding is required by the underlying operating system, FLTK will convert string as needed.

TODO:

- more doc on unicode, add links
- write something about filename encoding on OS X...
- explain the `fl_utf8_...` commands
- explain issues with [Fl_Preferences](#)
- why FLTK has no `Fl_String` class

DONE:

- initial transfer of the Ian/O'ksi'D patch
- adapted Makefiles and IDEs for available platforms
- hacked some Unicode keyboard entry for OS X

ISSUES:

- IDEs:
 - Makefile support: tested on Fedora Core 5 and OS X, but heaven knows on which platforms this may fail
 - Xcode: tested, seems to be working (but see comments below on OS X)

- VisualC (VC6): tested, test/utf8 works, but may have had some issues during merge. Some additional work needed (imm32.lib)
 - VisualStudio2005: tested, test/utf8 works, some addtl. work needed (imm32.lib)
 - VisualCNet: sorry, I have no longer access to that IDE
 - Borland and other compiler: sorry, I can't update those
- Platforms:
 - you will encounter problems on all platforms!
 - X11: many characters are missing, but that may be related to bad fonts on my machine. I also could not do any keyboard tests yet. Rendering seems to generally work ok.
 - Win32: US and German keyboard worked ok, but no compositing was tested. Rendering looks pretty good.
 - OS X: rendering looks good. Keyboard is completely messed up, even in US setting (with Alt key)
 - all: while merging I have seen plenty of places that are not entirely utf8-safe, particularly [FL_Input](#), [FL_Text_Editor](#), and [FL_Help_View](#). Keycodes from the keyboard conflict with Unicode characters. Right-to-left rendered text can not be marked or edited, and probably much more.

Chapter 14

C - FLTK Enumerations

Note:

This file is not actively maintained any more, but is left here as a reference, until the doxygen documentation is completed.

See also:

[FL/Enumerations.H](#).

This appendix lists the enumerations provided in the [<FL/Enumerations.H>](#) header file, organized by section. Constants whose value are zero are marked with "(0)", this is often useful to know when programming.

14.1 Version Numbers

The FLTK version number is stored in a number of compile-time constants:

- `FL_MAJOR_VERSION` - The major release number, currently 1.
- `FL_MINOR_VERSION` - The minor release number, currently 3.
- `FL_PATCH_VERSION` - The patch release number, currently 0.
- `FL_VERSION` - A combined floating-point version number for the major, minor, and patch release numbers, currently 1.0300.

14.2 Events

Events are identified by an `Fl_Event` enumeration value. The following events are currently defined:

- `FL_NO_EVENT` - No event (or an event fltk does not understand) occurred (0).
- `FL_PUSH` - A mouse button was pushed.
- `FL_RELEASE` - A mouse button was released.
- `FL_ENTER` - The mouse pointer entered a widget.
- `FL_LEAVE` - The mouse pointer left a widget.
- `FL_DRAG` - The mouse pointer was moved with a button pressed.
- `FL_FOCUS` - A widget should receive keyboard focus.
- `FL_UNFOCUS` - A widget loses keyboard focus.
- `FL_KEYBOARD` - A key was pressed.
- `FL_CLOSE` - A window was closed.
- `FL_MOVE` - The mouse pointer was moved with no buttons pressed.
- `FL_SHORTCUT` - The user pressed a shortcut key.
- `FL_DEACTIVATE` - The widget has been deactivated.

- FL_ACTIVATE - The widget has been activated.
- FL_HIDE - The widget has been hidden.
- FL_SHOW - The widget has been shown.
- FL_PASTE - The widget should paste the contents of the clipboard.
- FL_SELECTIONCLEAR - The widget should clear any selections made for the clipboard.
- FL_MOUSEWHEEL - The horizontal or vertical mousewheel was turned.
- FL_DND_ENTER - The mouse pointer entered a widget dragging data.
- FL_DND_DRAG - The mouse pointer was moved dragging data.
- FL_DND_LEAVE - The mouse pointer left a widget still dragging data.
- FL_DND_RELEASE - Dragged data is about to be dropped.

14.3 Callback "When" Conditions

The following constants determine when a callback is performed:

- FL_WHEN_NEVER - Never call the callback (0).
- FL_WHEN_CHANGED - Do the callback only when the widget value changes.
- FL_WHEN_NOT_CHANGED - Do the callback whenever the user interacts with the widget.
- FL_WHEN_RELEASE - Do the callback when the button or key is released and the value changes.
- FL_WHEN_ENTER_KEY - Do the callback when the user presses the ENTER key and the value changes.
- FL_WHEN_RELEASE_ALWAYS - Do the callback when the button or key is released, even if the value doesn't change.
- FL_WHEN_ENTER_KEY_ALWAYS - Do the callback when the user presses the ENTER key, even if the value doesn't change.

14.4 Fl::event_button() Values

The following constants define the button numbers for FL_PUSH and FL_RELEASE events:

- FL_LEFT_MOUSE - the left mouse button
- FL_MIDDLE_MOUSE - the middle mouse button
- FL_RIGHT_MOUSE - the right mouse button

14.5 Fl::event_key() Values

The following constants define the non-ASCII keys on the keyboard for FL_KEYBOARD and FL_SHORTCUT events:

- FL_Button - A mouse button; use `Fl_Button + n` for mouse button `n`.
- FL_BackSpace - The backspace key.
- FL_Tab - The tab key.
- FL_Enter - The enter key.
- FL_Pause - The pause key.
- FL_Scroll_Lock - The scroll lock key.
- FL_Escape - The escape key.
- FL_Home - The home key.
- FL_Left - The left arrow key.
- FL_Up - The up arrow key.
- FL_Right - The right arrow key.
- FL_Down - The down arrow key.
- FL_Page_Up - The page-up key.
- FL_Page_Down - The page-down key.
- FL_End - The end key.
- FL_Print - The print (or print-screen) key.
- FL_Insert - The insert key.
- FL_Menu - The menu key.
- FL_Num_Lock - The num lock key.
- FL_KP - One of the keypad numbers; use `FL_KP + n` for number `n`.
- FL_KP_Enter - The enter key on the keypad.
- FL_F - One of the function keys; use `FL_F + n` for function key `n`.
- FL_Shift_L - The lefthand shift key.
- FL_Shift_R - The righthand shift key.
- FL_Control_L - The lefthand control key.
- FL_Control_R - The righthand control key.
- FL_Caps_Lock - The caps lock key.
- FL_Meta_L - The left meta/Windows key.
- FL_Meta_R - The right meta/Windows key.
- FL_Alt_L - The left alt key.
- FL_Alt_R - The right alt key.
- FL_Delete - The delete key.

14.6 Fl::event_state() Values

The following constants define bits in the [Fl::event_state\(\)](#) value:

- FL_SHIFT - One of the shift keys is down.
- FL_CAPS_LOCK - The caps lock is on.
- FL_CTRL - One of the ctrl keys is down.
- FL_ALT - One of the alt keys is down.
- FL_NUM_LOCK - The num lock is on.
- FL_META - One of the meta/Windows keys is down.
- FL_COMMAND - An alias for FL_CTRL on WIN32 and X11, or FL_META on MacOS X.
- FL_SCROLL_LOCK - The scroll lock is on.
- FL_BUTTON1 - Mouse button 1 is pushed.
- FL_BUTTON2 - Mouse button 2 is pushed.
- FL_BUTTON3 - Mouse button 3 is pushed.
- FL_BUTTONS - Any mouse button is pushed.
- [FL_BUTTON\(n\)](#) - Mouse button n (where n > 0) is pushed.

14.7 Alignment Values

The following constants define bits that can be used with `Fl_Widget::alignn()` to control the positioning of the label:

- FL_ALIGN_CENTER - The label is centered (0).
- FL_ALIGN_TOP - The label is top-aligned.
- FL_ALIGN_BOTTOM - The label is bottom-aligned.
- FL_ALIGN_LEFT - The label is left-aligned.
- FL_ALIGN_RIGHT - The label is right-aligned.
- FL_ALIGN_CLIP - The label is clipped to the widget.
- FL_ALIGN_WRAP - The label text is wrapped as needed.
- FL_ALIGN_TOP_LEFT
- FL_ALIGN_TOP_RIGHT
- FL_ALIGN_BOTTOM_LEFT
- FL_ALIGN_BOTTOM_RIGHT
- FL_ALIGN_LEFT_TOP

- FL_ALIGN_RIGHT_TOP
- FL_ALIGN_LEFT_BOTTOM
- FL_ALIGN_RIGHT_BOTTOM
- FL_ALIGN_INSIDE - 'or' this with other values to put label inside the widget.

14.8 Fonts

The following constants define the standard FLTK fonts:

- FL_HELVETICA - Helvetica (or Arial) normal (0).
- FL_HELVETICA_BOLD - Helvetica (or Arial) bold.
- FL_HELVETICA_ITALIC - Helvetica (or Arial) oblique.
- FL_HELVETICA_BOLD_ITALIC - Helvetica (or Arial) bold-oblique.
- FL_COURIER - Courier normal.
- FL_COURIER_BOLD - Courier bold.
- FL_COURIER_ITALIC - Courier italic.
- FL_COURIER_BOLD_ITALIC - Courier bold-italic.
- FL_TIMES - Times roman.
- FL_TIMES_BOLD - Times bold.
- FL_TIMES_ITALIC - Times italic.
- FL_TIMES_BOLD_ITALIC - Times bold-italic.
- FL_SYMBOL - Standard symbol font.
- FL_SCREEN - Default monospaced screen font.
- FL_SCREEN_BOLD - Default monospaced bold screen font.
- FL_ZAPF_DINGBATS - Zapf-dingbats font.

14.9 Colors

The Fl_Color enumeration type holds a FLTK color value. Colors are either 8-bit indexes into a virtual colormap or 24-bit RGB color values. Color indices occupy the lower 8 bits of the value, while RGB colors occupy the upper 24 bits, for a byte organization of RGBI.

14.9.1 Color Constants

Constants are defined for the user-defined foreground and background colors, as well as specific colors and the start of the grayscale ramp and color cube in the virtual colormap. Inline functions are provided to retrieve specific grayscale, color cube, or RGB color values.

The following color constants can be used to access the user-defined colors:

- `FL_BACKGROUND_COLOR` - the default background color
- `FL_BACKGROUND2_COLOR` - the default background color for text, list, and valuator widgets
- `FL_FOREGROUND_COLOR` - the default foreground color (0) used for labels and text
- `FL_INACTIVE_COLOR` - the inactive foreground color
- `FL_SELECTION_COLOR` - the default selection/highlight color

The following color constants can be used to access the colors from the FLTK standard color cube:

- `FL_BLACK`
- `FL_BLUE`
- `FL_CYAN`
- `FL_DARK_BLUE`
- `FL_DARK_CYAN`
- `FL_DARK_GREEN`
- `FL_DARK_MAGENTA`
- `FL_DARK_RED`
- `FL_DARK_YELLOW`
- `FL_GREEN`
- `FL_MAGENTA`
- `FL_RED`
- `FL_WHITE`
- `FL_YELLOW`

The inline methods for getting a grayscale, color cube, or RGB color value are described in [Appendix B – Function Reference](#).

14.10 Cursors

The following constants define the mouse cursors that are available in FLTK. The double-headed arrows are bitmaps provided by FLTK on X, the others are provided by system-defined cursors.

- FL_CURSOR_DEFAULT - the default cursor, usually an arrow (0)
- FL_CURSOR_ARROW - an arrow pointer
- FL_CURSOR_CROSS - crosshair
- FL_CURSOR_WAIT - watch or hourglass
- FL_CURSOR_INSERT - I-beam
- FL_CURSOR_HAND - hand (uparrow on MSWindows)
- FL_CURSOR_HELP - question mark
- FL_CURSOR_MOVE - 4-pointed arrow
- FL_CURSOR_NS - up/down arrow
- FL_CURSOR_WE - left/right arrow
- FL_CURSOR_NWSE - diagonal arrow
- FL_CURSOR_NESW - diagonal arrow
- FL_CURSOR_NONE - invisible

14.11 FD "When" Conditions

- FL_READ - Call the callback when there is data to be read.
- FL_WRITE - Call the callback when data can be written without blocking.
- FL_EXCEPT - Call the callback if an exception occurs on the file.

14.12 Damage Masks

The following damage mask bits are used by the standard FLTK widgets:

- FL_DAMAGE_CHILD - A child needs to be redrawn.
- FL_DAMAGE_EXPOSE - The window was exposed.
- FL_DAMAGE_SCROLL - The [Fl_Scroll](#) widget was scrolled.
- FL_DAMAGE_OVERLAY - The overlay planes need to be redrawn.
- FL_DAMAGE_USER1 - First user-defined damage bit.
- FL_DAMAGE_USER2 - Second user-defined damage bit.
- FL_DAMAGE_ALL - Everything needs to be redrawn.

Chapter 15

D - GLUT Compatibility

This appendix describes the GLUT compatibility header file supplied with FLTK.

FLTK's GLUT compatibility is based on the original GLUT 3.7 and the follow-on FreeGLUT 2.4.0 libraries.

15.1 Using the GLUT Compatibility Header File

You should be able to compile existing GLUT source code by including `<FL/glut.H>` instead of `<GL/glut.h>`. This can be done by editing the source, by changing the `-I` switches to the compiler, or by providing a symbolic link from `GL/glut.h` to `FL/glut.H`.

All files calling GLUT procedures must be compiled with C++. You may have to alter them slightly to get them to compile without warnings, and you may have to rename them to get make to use the C++ compiler.

You must link with the FLTK library. Most of `FL/glut.H` is inline functions. You should take a look at it (and maybe at `test/glpuzzle.cxx` in the FLTK source) if you are having trouble porting your GLUT program.

This has been tested with most of the demo programs that come with the GLUT and FreeGLUT distributions.

15.2 Known Problems

The following functions and/or arguments to functions are missing, and you will have to replace them or comment them out for your code to compile:

- `glutGet (GLUT_ELAPSED_TIME)`
- `glutGet (GLUT_SCREEN_HEIGHT_MM)`
- `glutGet (GLUT_SCREEN_WIDTH_MM)`
- `glutGet (GLUT_WINDOW_NUM_CHILDREN)`
- `glutInitDisplayMode (GLUT_LUMINANCE)`
- `glutLayerGet (GLUT_HAS_OVERLAY)`
- `glutLayerGet (GLUT_LAYER_IN_USE)`
- `glutPushWindow ()`
- `glutSetColor (), glutGetColor (), glutCopyColormap ()`
- `glutVideoResize ()` missing.
- `glutWarpPointer ()`
- `glutWindowStatusFunc ()`
- Spaceball, buttonbox, dials, and tablet functions

Most of the symbols/enumerations have different values than GLUT uses. This will break code that relies on the actual values. The only symbols guaranteed to have the same values are true/false pairs like `GLUT_DOWN` and `GLUT_UP`, mouse buttons `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`, and `GLUT_KEY_F1` thru `GLUT_KEY_F12`.

The strings passed as menu labels are not copied.

`glutPostRedisplay()` does not work if called from inside a display function. You must use `glutIdleFunc()` if you want your display to update continuously.

`glutSwapBuffers()` does not work from inside a display function. This is on purpose, because FLTK swaps the buffers for you.

`glutUseLayer()` does not work well, and should only be used to initialize transformations inside a resize callback. You should redraw overlays by using `glutOverlayDisplayFunc()`.

Overlays are cleared before the overlay display function is called. `glutLayerGet(GLUT_OVERLAY_DAMAGED)` always returns true for compatibility with some GLUT overlay programs. You must rewrite your code so that `gl_color()` is used to choose colors in an overlay, or you will get random overlay colors.

`glutSetCursor(GLUT_CURSOR_FULL_CROSSHAIR)` just results in a small crosshair.

The fonts used by `glutBitmapCharacter()` and `glutBitmapWidth()` may be different.

`glutInit(argc, argv)` will consume different switches than GLUT does. It accepts the switches recognized by `Fl::args()`, and will accept any abbreviation of these switches (such as "-di" for "-display").

15.3 Mixing GLUT and FLTK Code

You can make your GLUT window a child of a `Fl_Window` with the following scheme. The biggest trick is that GLUT insists on a call to `show()` the window at the point it is created, which means the `Fl_Window` parent window must already be shown.

- Don't call `glutInit()`.
- Create your `Fl_Window`, and any FLTK widgets. Leave a blank area in the window for your GLUT window.
- `show()` the `Fl_Window`. Perhaps call `show(argc, argv)`.
- Call `window->begin()` so that the GLUT window will be automatically added to it.
- Use `glutInitWindowSize()` and `glutInitWindowPosition()` to set the location in the parent window to put the GLUT window.
- Put your GLUT code next. It probably does not need many changes. Call `window->end()` immediately after the `glutCreateWindow()` !
- You can call either `glutMainLoop()`, `Fl::run()`, or loop calling `Fl::wait()` to run the program.

15.4 class Fl_Glut_Window

15.4.1 Class Hierarchy

```
Fl_Gl_Window
|
+----Fl_Glut_Window
```

15.4.2 Include Files

```
#include <FL/glut.H>
```

15.4.3 Description

Each GLUT window is an instance of this class. You may find it useful to manipulate instances directly rather than use GLUT window id's. These may be created without opening the display, and thus can fit better into FLTK's method of creating windows.

The current GLUT window is available in the global variable `glut_window`.

`new Fl_Glut_Window(...)` is the same as `glutCreateWindow()` except it does not `show()` the window or make the window current.

`window->make_current()` is the same as `glutSetWindow(number)`. If the window has not had `show()` called on it yet, some functions that assume an OpenGL context will not work. If you do `show()` the window, call `make_current()` again to set the context.

`~Fl_Glut_Window()` is the same as `glutDestroyWindow()`.

15.4.4 Members

The `Fl_Glut_Window` class contains several public members that can be altered directly:

member	description
<code>display</code>	A pointer to the function to call to draw the normal planes.
<code>entry</code>	A pointer to the function to call when the mouse moves into or out of the window.
<code>keyboard</code>	A pointer to the function to call when a regular key is pressed.
<code>menu[3]</code>	The menu to post when one of the mouse buttons is pressed.
<code>mouse</code>	A pointer to the function to call when a button is pressed or released.
<code>motion</code>	A pointer to the function to call when the mouse is moved with a button down.
<code>overlaydisplay</code>	A pointer to the function to call to draw the overlay planes.
<code>passivemotion</code>	A pointer to the function to call when the mouse is moved with no buttons down.
<code>reshape</code>	A pointer to the function to call when the window is resized.
<code>special</code>	A pointer to the function to call when a special key is pressed.
<code>visibility</code>	A pointer to the function to call when the window is iconified or restored (made visible.)

15.4.5 Methods

- `Fl_Glut_Window()`
- `~Fl_Glut_Window()`

- `make_current()`

`Fl_Glut_Window::Fl_Glut_Window(int x, int y, int w, int h, const char *title = 0)`

`Fl_Glut_Window::Fl_Glut_Window(int w, int h, const char *title = 0)`

The first constructor takes 4 int arguments to create the window with a preset position and size. The second constructor with 2 arguments will create the window with a preset size, but the window manager will choose the position according to it's own whims.

`virtual Fl_Glut_Window::~~Fl_Glut_Window()`

Destroys the GLUT window.

`void Fl_Glut_Window::make_current()`

Switches all drawing functions to the GLUT window.

Chapter 16

E - Forms Compatibility

This appendix describes the Forms compatibility included with FLTK.

16.1 Importing Forms Layout Files

FLUID can read the `.fd` files put out by all versions of Forms and XForms `fdesign`. However, it will mangle them a bit, but it prints a warning message about anything it does not understand. **FLUID** cannot write `fdesign` files, so you should save to a new name so you don't write over the old one.

You will need to edit your main code considerably to get it to link with the output from **FLUID**. If you are not interested in this you may have more immediate luck with the forms compatibility header, `<FL/forms.H>`.

16.2 Using the Compatibility Header File

You should be able to compile existing Forms or XForms source code by changing the include directory switch to your compiler so that the `forms.h` file supplied with FLTK is included. Take a look at `forms.h` to see how it works, but the basic trick is lots of inline functions. Most of the XForms demo programs work without changes.

You will also have to compile your Forms or XForms program using a C++ compiler. The FLTK library does not provide C bindings or header files.

Although FLTK was designed to be compatible with the GL Forms library (version 0.3 or so), XForms has bloated severely and it's interface is X-specific. Therefore, XForms compatibility is no longer a goal of FLTK. Compatibility was limited to things that were free, or that would add code that would not be linked in if the feature is unused, or that was not X-specific.

To use any new features of FLTK, you should rewrite your code to not use the inline functions and instead use "pure" FLTK. This will make it a lot cleaner and make it easier to figure out how to call the FLTK functions. Unfortunately this conversion is harder than expected and even Digital Domain's inhouse code still uses `forms.H` a lot.

16.3 Problems You Will Encounter

Many parts of XForms use X-specific structures like `XEvent` in their interface. I did not emulate these! Unfortunately these features (such as the "canvas" widget) are needed by most large programs. You will need to rewrite these to use FLTK subclasses.

Fl_Free widgets emulate the *old* Forms "free" widget. It may be useful for porting programs that change the `handle()` function on widgets, but you will still need to rewrite things.

Fl_Timer widgets are provided to emulate the XForms timer. These work, but are quite inefficient and inaccurate compared to using `Fl::add_timeout()`.

All instance variables are hidden. If you directly refer to the `x`, `y`, `w`, `h`, `label`, or other fields of your Forms widgets you will have to add empty parenthesis after each reference. The easiest way to do this is to globally replace "`→ x`" with "`→ x()`", etc. Replace "boxtype" with "box()".

`const char *` arguments to most FLTK methods are simply stored, while Forms would `strdup()` the passed string. This is most noticable with the `label` of widgets. Your program must always pass static data such as a string constant or `malloc'd` buffer to `label()`. If you are using labels to display program output you may want to try the **Fl_Output** widget.

The default fonts and sizes are matched to the older GL version of Forms, so all labels will draw somewhat larger than an XForms program does.

fdesign outputs a setting of a "fdi" instance variable to the main window. I did not emulate this because I wanted all instance variables to be hidden. You can store the same information in the `user_data()` field of a window. To do this, search through the fdesign output for all occurrences of "`→ fdi`" and edit to use "`→ user_data()`" instead. This will require casts and is not trivial.

The prototype for the functions passed to `fl_add_timeout()` and `fl_set_idle_callback()` callback are different.

All the following XForms calls are missing:

- `FL_REVISION, fl_library_version()`
- `FL_RETURN_DBLCLICK` (use `Fl::event_clicks()`)
- `fl_add_signal_callback()`
- `fl_set_form_atactivate()` `fl_set_form_atdeactivate()`
- `fl_set_form_property()`
- `fl_set_app_mainform(), fl_get_app_mainform()`
- `fl_set_form_minsize(), fl_set_form_maxsize()`
- `fl_set_form_event_cmask(), fl_get_form_event_cmask()`
- `fl_set_form_dblbuffer(), fl_set_object_dblbuffer()` (use an `Fl_Double_Window` instead)
- `fl_adjust_form_size()`
- `fl_register_raw_callback()`
- `fl_set_object_bw(), fl_set_border_width()`
- `fl_set_object_resize(), fl_set_object_gravity()`
- `fl_set_object_shortcutkey()`
- `fl_set_object_automatic()`
- `fl_get_object_bbox()` (maybe FLTK should do this)
- `fl_set_object_prehandler(), fl_set_object_posthandler()`
- `fl_enumerate_fonts()`
- Most drawing functions
- `fl_set_coordunit()` (FLTK uses pixels all the time)
- `fl_ringbell()`
- `fl_gettime()`
- `fl_win*()` (all these functions)
- `fl_initialize(argc, argv, x, y, z)` ignores last 3 arguments
- `fl_read_bitmapfile(), fl_read_pixmapfile()`

- `fl_addto_browser_chars()`
- `FL_MENU_BUTTON` just draws normally
- `fl_set_bitmapbutton_file()`, `fl_set_pixmapbutton_file()`
- `FL_CANVAS` objects
- `FL_DIGITAL_CLOCK` (comes out analog)
- `fl_create_bitmap_cursor()`, `fl_set_cursor_color()`
- `fl_set_dial_angles()`
- `fl_show_oneliner()`
- `fl_set_choice_shortcut(a, b, c)`
- command log
- Only some of file selector is emulated
- `FL_DATE_INPUT`
- `fl_pup*()` (all these functions)
- textbox object (should be easy but I had no sample programs)
- xyplot object

16.4 Additional Notes

These notes were written for porting programs written with the older IRISGL version of Forms. Most of these problems are the same ones encountered when going from old Forms to XForms:

Does Not Run In Background

The IRISGL library always forked when you created the first window, unless "foreground()" was called. FLTK acts like "foreground()" is called all the time. If you really want the fork behavior do "if (fork()) exit(0)" right at the start of your program.

You Cannot Use IRISGL Windows or `fl_queue`

If a Forms (not XForms) program if you wanted your own window for displaying things you would create a IRISGL window and draw in it, periodically calling Forms to check if the user hit buttons on the panels. If the user did things to the IRISGL window, you would find this out by having the value `FL_EVENT` returned from the call to Forms.

None of this works with FLTK. Nor will it compile, the necessary calls are not in the interface.

You have to make a subclass of `Fl_Gl_Window` and write a `draw()` method and `handle()` method. This may require anywhere from a trivial to a major rewrite.

If you draw into the overlay planes you will have to also write a `draw_overlay()` method and call `redraw_overlay()` on the OpenGL window.

One easy way to hack your program so it works is to make the `draw()` and `handle()` methods on your window set some static variables, storing what event happened. Then in the main loop of your program, call `Fl::wait()` and then check these variables, acting on them as though they are events read from `fl_queue`.

You Must Use OpenGL to Draw Everything

The file `<FL/gl.h>` defines replacements for a lot of IRISGL calls, translating them to OpenGL. There are much better translators available that you might want to investigate.

You Cannot Make Forms Subclasses

Programs that call `fl_make_object` or directly setting the handle routine will not compile. You have to rewrite them to use a subclass of `Fl_Widget`. It is important to note that the `handle()` method is not exactly the same as the `handle()` function of Forms. Where a Forms `handle()` returned non-zero, your `handle()` must call `do_callback()`. And your `handle()` must return non-zero if it "understood" the event.

An attempt has been made to emulate the "free" widget. This appears to work quite well. It may be quicker to modify your subclass into a "free" widget, since the "handle" functions match.

If your subclass draws into the overlay you are in trouble and will have to rewrite things a lot.

You Cannot Use `<device.h>`

If you have written your own "free" widgets you will probably get a lot of errors about "getvaluator". You should substitute:

Forms	FLTK
MOUSE_X	Fl::event_x_root()
MOUSE_Y	Fl::event_y_root()
LEFTSHIFTKEY,RIGHTSHIFTKEY	Fl::event_shift()
CAPSLOCKKEY	Fl::event_capslock()
LEFTCTRLKEY,RIGHTCTRLKEY	Fl::event_ctrl()
LEFTALTKEY,RIGHTALTKEY	Fl::event_alt()
MOUSE1,RIGHTMOUSE	Fl::event_state()
MOUSE2,MIDDLEMOUSE	Fl::event_state()
MOUSE3,LEFTMOUSE	Fl::event_state()

Anything else in `getvaluator` and you are on your own...

Font Numbers Are Different

The "style" numbers have been changed because I wanted to insert bold-italic versions of the normal fonts. If you use Times, Courier, or Bookman to display any text you will get a different font out of FLTK. If you are really desperate to fix this use the following code:

```
fl_font_name(3, "*courier-medium-r-no*");  
fl_font_name(4, "*courier-bold-r-no*");  
fl_font_name(5, "*courier-medium-o-no*");  
fl_font_name(6, "*times-medium-r-no*");  
fl_font_name(7, "*times-bold-r-no*");  
fl_font_name(8, "*times-medium-i-no*");  
fl_font_name(9, "*bookman-light-r-no*");  
fl_font_name(10, "*bookman-demi-r-no*");  
fl_font_name(11, "*bookman-light-i-no*");
```

Chapter 17

F - Operating System Issues

This appendix describes the operating system specific interfaces in FLTK.

17.1 Accessing the OS Interfaces

All programs that need to access the operating system specific interfaces must include the following header file:

```
#include <FL/x.H>
```

Despite the name, this header file will define the appropriate interface for your environment. The pages that follow describe the functionality that is provided for each operating system.

WARNING:

The interfaces provided by this header file may change radically in new FLTK releases. Use them only when an existing generic FLTK interface is not sufficient.

17.2 The UNIX (X11) Interface

The UNIX interface provides access to the X Window System state information and data structures.

17.2.1 Handling Other X Events

```
void Fl::add_handler(int (*f)(int))
```

Installs a function to parse unrecognized events. If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non-zero then the event is ignored.

FLTK calls this for any X events it does not recognize, or X events with a window ID that FLTK does not recognize. You can look at the X event in the `fl_xevent` variable.

The argument is the FLTK event type that was not handled, or zero for unrecognized X events. These handlers are also called for global shortcuts and some other events that the widget they were passed to did not handle, for example `FL_SHORTCUT`.

```
extern XEvent *fl_xvent
```

This variable contains the most recent X event.

```
extern ulong fl_event_time
```

This variable contains the time stamp from the most recent X event that reported it; not all events do. Many X calls like cut and paste need this value.

```
Window fl_xid(const Fl_Window *)
```

Returns the XID for a window, or zero if not shown().

```
Fl_Window *fl_find(ulong xid)
```

Returns the `Fl_Window` that corresponds to the given XID, or `NULL` if not found. This function uses a cache so it is slightly faster than iterating through the windows yourself.

```
int fl_handle(const XEvent &)
```

This call allows you to supply the X events to FLTK, which may allow FLTK to cooperate with another toolkit or library. The return value is non-zero if FLTK understood the event. If the window does not belong to FLTK and the `add_handler()` functions all return 0, this function will return false.

Besides feeding events your code should call `Fl::flush()` periodically so that FLTK redraws its windows.

This function will call the callback functions. It will not return until they complete. In particular, if a callback pops up a modal window by calling `fl_ask()`, for instance, it will not return until the modal function returns.

17.2.2 Drawing using Xlib

The following global variables are set before `Fl_Widget::draw()` is called, or by `Fl_Window::make_current()`:

```
extern Display *fl_display;
extern Window fl_window;
extern GC fl_gc;
extern int fl_screen;
extern XVisualInfo *fl_visual;
extern Colormap fl_colormap;
```

You must use them to produce Xlib calls. Don't attempt to change them. A typical X drawing call is written like this:

```
XDrawSomething(fl_display, fl_window, fl_gc, ...);
```

Other information such as the position or size of the X window can be found by looking at `Fl_Window::current()`, which returns a pointer to the `Fl_Window` being drawn.

`unsigned long fl_xpixel(Fl_Color i)`

`unsigned long fl_xpixel(uchar r, uchar g, uchar b)`

Returns the X pixel number used to draw the given FLTK color index or RGB color. This is the X pixel that `fl_color()` would use.

`int fl_parse_color(const char* p, uchar& r, uchar& g, uchar& b)`

Convert a name into the red, green, and blue values of a color by parsing the X11 color names. On other systems, `fl_parse_color` can only convert names in hexadecimal encoding, for example `#ff8083`.

`extern XFontStruct *fl_xfont`

Points to the font selected by the most recent `fl_font()`. This is not necessarily the current font of `fl_gc`, which is not set until `fl_draw()` is called. If FLTK was compiled with Xft support, `fl_xfont` will usually be 0 and `fl_xftfont` will contain a pointer to the `XftFont` structure instead.

`extern void *fl_xftfont`

If FLTK was compiled with Xft support enabled, `fl_xftfont` Points to the xft font selected by the most recent `fl_font()`. Otherwise it will be 0. `fl_xftfont` should be cast to `XftFont*`.

17.2.3 Changing the Display, Screen, or X Visual

FLTK uses only a single display, screen, X visual, and X colormap. This greatly simplifies its internal structure and makes it much smaller and faster. You can change which it uses by setting global variables

before the first `Fl_Window::show()` is called. You may also want to call `Fl::visual()`, which is a portable interface to get a full color and/or double buffered visual.

`int Fl::display(const char *)`

Set which X display to use. This actually does `putenv("DISPLAY=...")` so that child programs will display on the same screen if called with `exec()`. This must be done before the display is opened. This call is provided under MacOS and WIN32 but it has no effect.

`extern Display *fl_display`

The open X display. This is needed as an argument to most Xlib calls. Don't attempt to change it! This is `NULL` before the display is opened.

`void fl_open_display()`

Opens the display. Does nothing if it is already open. This will make sure `fl_display` is non-zero. You should call this if you wish to do X calls and there is a chance that your code will be called before the first `show()` of a window.

This may call `Fl::abort()` if there is an error opening the display.

`void fl_close_display()`

This closes the X connection. You do *not* need to call this to exit, and in fact it is faster to not do so! It may be useful to call this if you want your program to continue without the X connection. You cannot open the display again, and probably cannot call any FLTK functions.

`extern int fl_screen`

Which screen number to use. This is set by `fl_open_display()` to the default screen. You can change it by setting this to a different value immediately afterwards. It can also be set by changing the last number in the `Fl::display()` string to "host:0.#".

`extern XVisualInfo *fl_visual`

`extern Colormap fl_colormap`

The visual and colormap that FLTK will use for all windows. These are set by `fl_open_display()` to the default visual and colormap. You can change them before calling `show()` on the first window. Typical code for changing the default visual is:

```
Fl::args(argc, argv); // do this first so $DISPLAY is set
fl_open_display();
fl_visual = find_a_good_visual(fl_display, fl_screen);
if (!fl_visual) Fl::abort("No good visual");
fl_colormap = make_a_colormap(fl_display, fl_visual->visual, fl_visual->depth);
// it is now ok to show() windows:
window->show(argc, argv);
```

17.2.4 Using a Subclass of `Fl_Window` for Special X Stuff

FLTK can manage an X window on a different screen, visual and/or colormap, you just can't use FLTK's drawing routines to draw into it. But you can write your own `draw()` method that uses Xlib (and/or OpenGL) calls only.

FLTK can also manage XID's provided by other libraries or programs, and call those libraries when the window needs to be redrawn.

To do this, you need to make a subclass of `Fl_Window` and override some of these virtual functions:

`virtual void Fl_Window::show()`

If the window is already `shown()` this must cause it to be raised, this can usually be done by calling

`Fl_Window::show()`. If not `shown()` your implementation must call either `Fl_X::set_xid()` or `Fl_X::make_xid()`.

An example:

```
void MyWindow::show() {
    if (shown()) {Fl_Window::show(); return;} // you must do this!
    fl_open_display(); // necessary if this is first window
    // we only calculate the necessary visual colormap once:
    static XVisualInfo *visual;
    static Colormap colormap;
    if (!visual) {
        visual = figure_out_visual();
        colormap = XCreateColormap(fl_display, RootWindow(fl_display, fl_screen),
                                   vis->visual, AllocNone);
    }
    Fl_X::make_xid(this, visual, colormap);
}
```

`Fl_X *Fl_Xset_xid(Fl_Window *, Window xid)`

Allocate a hidden structure called an `Fl_X`, put the XID into it, and set a pointer to it from the `Fl_Window`. This causes `Fl_Window::shown()` to return true.

`void Fl_X::make_xid(Fl_Window *, XVisualInfo *vis, Colormap = fl_colormap)`

This static method does the most onerous parts of creating an X window, including setting the label, resize limitations, etc. It then does `Fl_X::set_xid()` with this new window and maps the window.

`virtual void Fl_Window::flush()`

This virtual function is called by `Fl::flush()` to update the window. For FLTK's own windows it does this by setting the global variables `fl_window` and `fl_gc` and then calling the `draw()` method. For your own windows you might just want to put all the drawing code in here.

The X region that is a combination of all `damage()` calls done so far is in `Fl_X::i(this)->region`. If NULL then you should redraw the entire window. The undocumented function `fl_clip_region(XRegion)` will initialize the FLTK clip stack with a region or NULL for no clipping. You must set region to NULL afterwards as `fl_clip_region()` will own and delete it when done.

If `damage()` & `FL_DAMAGE_EXPOSE` then only X expose events have happened. This may be useful if you have an undamaged image (such as a backing buffer) around.

Here is a sample where an undamaged image is kept somewhere:

```
void MyWindow::flush() {
    fl_clip_region(Fl_X::i(this)->region);
    Fl_X::i(this)->region = 0;
    if (damage() != 2) {... draw things into backing store ...}
    ... copy backing store to window ...
}
```

`virtual void Fl_Window::hide()`

Destroy the window server copy of the window. Usually you will destroy contexts, pixmaps, or other resources used by the window, and then call `Fl_Window::hide()` to get rid of the main window identified by `xid()`. If you override this, you must also override the destructor as shown:

```
void MyWindow::hide() {
    if (mypixmap) {
        XFreePixmap(fl_display, mypixmap);
        mypixmap = 0;
    }
}
```

```
Fl_Window::hide(); // you must call this
}
```

virtual void `Fl_Window::~~Fl_Window()`

Because of the way C++ works, if you override `hide()` you *must* override the destructor as well (otherwise only the base class `hide()` is called):

```
MyWindow::~MyWindow() {
    hide();
}
```

17.2.5 Setting the Icon of a Window

FLTK currently supports setting a window's icon **before** it is shown using the `Fl_Window::icon()` method.

void `Fl_Window::icon(char *)`

Sets the icon for the window to the passed pointer. You will need to cast the icon `Pixmap` to a `char *` when calling this method. To set a monochrome icon using a bitmap compiled with your application use:

```
#include "icon.xbm"

fl_open_display(); // needed if display has not been previously opened

Pixmap p = XCreateBitmapFromData(fl_display, DefaultRootWindow(fl_display),
                                icon_bits, icon_width, icon_height);

window->icon((char *)p);
```

To use a multi-colored icon, the XPM format and library should be used as follows:

```
#include <X11/xpm.h>
#include "icon.xpm"

fl_open_display(); // needed if display has not been previously opened

Pixmap p, mask;

XpmCreatePixmapFromData(fl_display, DefaultRootWindow(fl_display),
                       icon_xpm, &p, &mask, NULL);

window->icon((char *)p);
```

When using the Xpm library, be sure to include it in the list of libraries that are used to link the application (usually `-lXpm`).

NOTE:

You must call `Fl_Window::show(argc, argv)` for the icon to be used. The `Fl_Window::show()` method does not bind the icon to the window.

17.2.6 X Resources

When the `Fl_Window::show(argc, argv)` method is called, FLTK looks for the following X resources:

- `background` - The default background color for widgets (color).
- `dndTextOps` - The default setting for drag and drop text operations (boolean).
- `foreground` - The default foreground (label) color for widgets (color).
- `scheme` - The default scheme to use (string).
- `selectBackground` - The default selection color for menus, etc. (color).
- `Text.background` - The default background color for text fields (color).
- `tooltips` - The default setting for tooltips (boolean).
- `visibleFocus` - The default setting for visible keyboard focus on non-text widgets (boolean).

Resources associated with the first window's `Fl_Window::xclass()` string are queried first, or if no class has been specified then the class "fltk" is used (e.g. `fltk.background`). If no match is found, a global search is done (e.g. `*background`).

17.3 The Windows (WIN32) Interface

The Windows interface provides access to the WIN32 GDI state information and data structures.

17.3.1 Handling Other WIN32 Messages

By default a single WNDCLASSEX called "FLTK" is created. All `Fl_Window`'s are of this class unless you use `Fl_Window::xclass()`. The window class is created the first time `Fl_Window::show()` is called.

You can probably combine FLTK with other libraries that make their own WIN32 window classes. The easiest way is to call `Fl::wait()`, as it will call `DispatchMessage` for all messages to the other windows. If necessary you can let the other library take over as long as it calls `DispatchMessage()`, but you will have to arrange for the function `Fl::flush()` to be called regularly so that widgets are updated, timeouts are handled, and the idle functions are called.

`extern MSG fl_msg`

This variable contains the most recent message read by `GetMessage`, which is called by `Fl::wait()`. This may not be the most recent message sent to an FLTK window, because silly WIN32 calls the handle procedures directly for some events (sigh).

`void Fl::add_handler(int (*f)(int))`

Installs a function to parse unrecognized messages sent to FLTK windows. If FLTK cannot figure out what to do with a message, it calls each of these functions (most recent first) until one of them returns non-zero. The argument passed to the functions is the FLTK event that was not handled or zero for unknown messages. If all the handlers return zero then FLTK calls `DefWindowProc()`.

`HWND fl_xid(const Fl_Window *)`

Returns the window handle for a `Fl_Window`, or zero if not shown().

`Fl_Window *fl_find(HWND xid)`

Returns the `Fl_Window` that corresponds to the given window handle, or `NULL` if not found. This function uses a cache so it is slightly faster than iterating through the windows yourself.

17.3.2 Drawing Things Using the WIN32 GDI

When the virtual function `Fl_Widget::draw()` is called, FLTK stores all the silly extra arguments you need to make a proper GDI call in some global variables:

```
extern HINSTANCE fl_display;  
extern HWND fl_window;  
extern HDC fl_gc;  
COLORREF fl_rgb();  
HPEN fl_pen();  
HBRUSH fl_brush();
```

These global variables are set before `draw()` is called, or by `Fl_Window::make_current()`. You can refer to them when needed to produce GDI calls, but don't attempt to change them. The functions return GDI objects for the current color set by `fl_color()` and are created as needed and cached. A typical GDI drawing call is written like this:

```
DrawSomething(fl_gc, ..., fl_brush());
```

It may also be useful to refer to `Fl_Window::current()` to get the window's size or position.

17.3.3 Setting the Icon of a Window

FLTK currently supports setting a window's icon *before* it is shown using the `Fl_Window::icon()` method.

```
void Fl_Window::icon(char *)
```

Sets the icon for the window to the passed pointer. You will need to cast the `HICON` handle to a `char *` when calling this method. To set the icon using an icon resource compiled with your application use:

```
window->icon((char *)LoadIcon(fl_display, MAKEINTRESOURCE(IDI_ICON)));
```

You can also use the `LoadImage()` and related functions to load specific resolutions or create the icon from bitmap data.

NOTE:

You must call `Fl_Window::show(argc, argv)` for the icon to be used. The `Fl_Window::show()` method does not bind the icon to the window.

17.3.4 How to Not Get a MSDOS Console Window

WIN32 has a really stupid mode switch stored in the executables that controls whether or not to make a console window.

To always get a console window you simply create a console application (the `"/SUBSYSTEM:CONSOLE"` option for the linker). For a GUI-only application create a WIN32 application (the `"/SUBSYSTEM:WINDOWS"` option for the linker).

FLTK includes a `WinMain()` function that calls the ANSI standard `main()` entry point for you. *This function creates a console window when you use the debug version of the library.*

WIN32 applications without a console cannot write to `stdout` or `stderr`, even if they are run from a console window. Any output is silently thrown away. Additionally, WIN32 applications are run in the background by the console, although you can use `"start /wait program"` to run them in the foreground.

17.3.5 Known WIN32 Bugs and Problems

The following is a list of known bugs and problems in the WIN32 version of FLTK:

- If a program is deactivated, `Fl::wait()` does not return until it is activated again, even though many events are delivered to the program. This can cause idle background processes to stop unexpectedly. This also happens while the user is dragging or resizing windows or otherwise holding the mouse down. We were forced to remove most of the efficiency FLTK uses for redrawing in order to get windows to update while being moved. This is a design error in WIN32 and probably impossible to get around.
- `Fl_Gl_Window::can_do_overlay()` returns true until the first time it attempts to draw an overlay, and then correctly returns whether or not there is overlay hardware.
- `SetCapture` (used by `Fl::grab()`) doesn't work, and the main window title bar turns gray while menus are popped up.
- Compilation with `gcc 3.4.4` and `-Os` exposes an optimisation bug in `gcc`. The symptom is that when drawing filled circles only the perimeter is drawn. This can for instance be seen in the symbols demo. Other optimisation options such as `-O2` and `-O3` seem to work OK. More details can be found in STR#1656

17.4 The MacOS Interface

FLTK supports MacOS X using the Apple Carbon library. Older versions of MacOS are *not* supported.

Control, Option, and Command Modifier Keys

FLTK maps the Mac 'control' key to `FL_CTRL`, the 'option' key to `FL_ALT` and the 'Apple' key to `FL_META`. Keyboard events return the key name in `Fl::event_key()` and the keystroke translation in `Fl::event_text()`. For example, typing Option-Y on a Mac keyboard will set `FL_ALT` in `Fl::event_state()`, set `Fl::event_key()` to 'y' and return the Yen symbol in `Fl::event_text()`.

WindowRef fl_xid(const Fl_Window *)

Returns the window reference for an `Fl_Window`, or `NULL` if the window has not been shown.

`Fl_Window` *fl_find(WindowRef xid)

Returns the `Fl_Window` that corresponds to the give window handle, or `NULL` if not found. FLTK windows that are children of top-level windows share the WindowRef of the top-level window.

17.4.1 Apple "Quit" Event

When the user press Cmd-Q or requests a termination of the application, OS X will send a "Quit" Apple Event. FLTK handles this event by sending an `FL_CLOSE` event to all open windows. If all windows close, the application will terminate.

17.4.2 Apple "Open" Event

Whenever the user drops a file onto an application icon, OS X generates an Apple Event of the type "Open". You can have FLTK notify you of an Open event by setting the `fl_open_callback`.

```
void fl_open_callback(void (*cb)(const char *))
```

`cb` will be called with a single iUnix-style file name and path. If multiple files were dropped, `fl_open_callback` will be called multiple times.

17.4.3 Drawing Things Using QuickDraw

When the virtual function `Fl_Widget::draw()` is called, FLTK has prepared the Window and CGraf-Port for drawing. Clipping and offsets are prepared to allow correct subwindow drawing.

17.4.4 Drawing Things Using Quartz

If the FLTK library was compiled using the configuration flag `-enable-quartz`, all code inside `Fl_Widget::draw()` is expected to call Quartz drawing functions instead of QuickDraw. The Quartz coordinate system is flipped to match FLTK's coordinate system. The origin for all drawing is in the top left corner of the enclosing `Fl_Window`.

`Fl_Double_Window`

OS X double-buffers all windows automatically. On OS X, `Fl_Window` and `Fl_Double_Window` are handled internally in the same way.

17.4.5 Mac File System Specifics

Resource Forks

FLTK does not access the resource fork of an application. However, a minimal resource fork must be created for OS X applications

Caution:

When using UNIX commands to copy or move executables, OS X will NOT copy any resource forks! For copying and moving use `CpMac` and `MvMac` respectively. For creating a tar archive, all executables need to be stripped from their Resource Fork before packing, e.g. "DeRez fluid > fluid.r". After unpacking the Resource Fork needs to be reattached, e.g. "Rez fluid.r -o fluid".

It is advisable to use the Finder for moving and copying and Mac archiving tools like `Sit` for distribution as they will handle the Resource Fork correctly.

Mac File Paths

FLTK uses UNIX-style filenames and paths.

17.4.6 Known MacOS Bugs and Problems

The following is a list of known bugs and problems in the MacOS version of FLTK:

- Line styles are not well supported. This is due to limitations in the QuickDraw interface.
- Nested subwindows are not supported, i.e. you can have a `Fl_Window` widget inside a `Fl_Window`, but not a `Fl_Window` inside a `Fl_Window` inside a `Fl_Window`.

Chapter 18

G - Migrating Code from FLTK 1.0 to 1.1

This appendix describes the differences between the FLTK 1.0.x and FLTK 1.1.x functions and classes.

18.1 Color Values

Color values are now stored in a 32-bit unsigned integer instead of the unsigned character in 1.0.x. This allows for the specification of 24-bit RGB values or 8-bit FLTK color indices.

FL_BLACK and FL_WHITE now remain black and white, even if the base color of the gray ramp is changed using [Fl::background\(\)](#). FL_DARK3 and FL_LIGHT3 can be used instead to draw a very dark or a very bright background hue.

Widgets use the new color symbols FL_FORGROUND_COLOR, FL_BACKGROUND_COLOR, FL_BACKGROUND2_COLOR, FL_INACTIVE_COLOR, and FL_SELECTION_COLOR. More details can be found in the chapter [C - FLTK Enumerations](#).

18.2 Cut and Paste Support

The FLTK clipboard is now broken into two parts - a local selection value and a cut-and-paste value. This allows FLTK to support things like highlighting and replacing text that was previously cut or copied, which makes FLTK applications behave like traditional GUI applications.

18.3 File Chooser

The file chooser in FLTK 1.1.x is significantly different than the one supplied with FLTK 1.0.x. Any code that directly references the old FCB class or members will need to be ported to the new [Fl_File_Chooser](#) class.

18.4 Function Names

Some function names have changed from FLTK 1.0.x to 1.1.x in order to avoid name space collisions. You can still use the old function names by defining the FLTK_1_0_COMPAT symbol on the command-line when you compile (-DFLTK_1_0_COMPAT) or in your source, e.g.:

```
#define FLTK_1_0_COMPAT
#include <FL/Fl.H>
#include <FL/Enumerations.H>
#include <FL/FILENAME.H>
```

The following table shows the old and new function names:

Old 1.0.x Name	New 1.1.x Name
contrast()	fl_contrast()
down()	fl_down()
filename_absolute()	fl_filename_absolute()
filename_expand()	fl_filename_expand()
filename_ext()	fl_filename_ext()
filename_isdir()	fl_filename_isdir()
filename_list()	fl_filename_list()
filename_match()	fl_filename_match()
filename_name()	fl_filename_name()
filename_relative()	fl_filename_relative()
filename_setext()	fl_filename_setext()
frame()	fl_frame()
inactive()	fl_inactive()
numeric_sort()	fl_numeric_sort()

18.5 Image Support

Image support in FLTK has been significantly revamped in 1.1.x. The [Fl_Image](#) class is now a proper base class, with the core image drawing functionality in the [Fl_Bitmap](#), [Fl_Pixmap](#), and [Fl_RGB_Image](#) classes.

BMP, GIF, JPEG, PNG, XBM, and XPM image files can now be loaded using the appropriate image classes, and the [Fl_Shared_Image](#) class can be used to cache images in memory.

Image labels are no longer provided as an add-on label type. If you use the old `label()` methods on an image, the widget's `image()` method is called to set the image as the label.

Image labels in menu items must still use the old `labeltype` mechanism to preserve source compatibility.

18.6 Keyboard Navigation

FLTK 1.1.x now supports keyboard navigation and control with all widgets. To restore the old FLTK 1.0.x behavior so that only text widgets get keyboard focus, call the [Fl::visible_focus\(\)](#) method to disable it:

```
Fl::visible_focus(0);
```


Chapter 19

H - Migrating Code from FLTK 1.1 to 1.3

This appendix describes the differences between the FLTK 1.1.x and FLTK 1.3.x functions and classes.

19.1 Migrating From FLTK 1.0

If you want to migrate your code from FLTK 1.0 to FLTK 1.3, then you should first consult [Appendix G - Migrating Code from FLTK 1.0 to 1.1](#).

19.2 FL_Scroll Widget

[FL_Scroll::scroll_to\(int x, int y\)](#) replaces [FL_Scroll::position\(int x, int y\)](#).

This change was needed, because [FL_Scroll::position\(int,int\)](#) redefined [FL_Widget::position\(int,int\)](#), but with a completely different function (moving the scrollbars instead of moving the widget).

Please be aware that you need to change your application's code for all FL_Scroll-derived widgets, if you used [FL_Scroll::position\(int x, int y\)](#) to position **the scrollbars** (not the widget itself).

The compiler will not detect any errors, because your calls to *position(int x, int y)* **will** be calling [FL_Widget::position\(int x, int y\)](#).

19.3 Unicode (utf-8)

FLTK 1.3 uses Unicode (utf-8) encoding internally. If you are only using characters in the ASCII range (32-127), there is a high probability that you don't need to modify your code. However, if you use international characters (128-255), encoded as e.g. Windows codepage 1252, ISO-8859-1, ISO-8859-15 or any other encoding, then you will need to update your character string constants and widget input data accordingly.

Note:

It is important that, although your software uses only ASCII characters for input to FLTK widgets, the user may enter non-ASCII characters, and FLTK will return these characters with utf-8 encoding to your application, e.g. via [FL_Input::value\(\)](#). You **will** need to re-encode them to **your** (non-utf-8) encoding, otherwise you might see or print garbage in your data.

[For more information see here.](#)

19.4 Widget Coordinate Representation

FLTK 1.3 changed all Widget coordinate variables and methods, e.g. [FL_Widget::x\(\)](#), [FL_Widget::y\(\)](#), [FL_Widget::w\(\)](#), [FL_Widget::h\(\)](#), from short (16-bit) to int (32-bit) representation. This should not affect any existing code, but makes it possible to use bigger scroll areas (e.g. [FL_Scroll](#) widget).

Chapter 20

I - Developer Information

This chapter describes FLTK development and documentation.

Note:

documentation with doxygen will be described here.

Example

Note:

In the following code example(s) "*" will be replaced by "#" as a temporary solution.

```

/## \file
    Fl_Clock, Fl_Clock_Output widgets . #/

/##
    \class Fl_Clock_Output
    \brief This widget can be used to display a program-supplied time.

    The time shown on the clock is not updated. To display the current time,
    use Fl_Clock instead.

    \image html clock.gif
    \image latex clock.eps "" width=10cm
    \image html round_clock.gif
    \image latex clock.eps "" width=10cm
    \image html round_clock.eps "" width=10cm #/

/##
    Returns the displayed time.
    Returns the time in seconds since the UNIX epoch (January 1, 1970).
    \see value(ulong)
    #/
    ulong value() const {return value_;}

/##
    Set the displayed time.
    Set the time in seconds since the UNIX epoch (January 1, 1970).
    \param[in] v seconds since epoch
    \see value()
    #/
void Fl_Clock_Output::value(ulong v) {
    [...]
}

/##
    Create an Fl_Clock widget using the given position, size, and label string.
    The default boxtype is \c FL_NO_BOX.
    \param[in] X, Y, W, H position and size of the widget
    \param[in] L widget label, default is no label
    #/
Fl_Clock::Fl_Clock(int X, int Y, int W, int H, const char #L)
    : Fl_Clock_Output(X, Y, W, H, L) {}

/##
    Create an Fl_Clock widget using the given boxtype, position, size, and
    label string.
    \param[in] t boxtype
    \param[in] X, Y, W, H position and size of the widget
    \param[in] L widget label, default is no label
    #/

```

```
Fl_Clock::Fl_Clock(uchar t, int X, int Y, int W, int H, const char #L)
: Fl_Clock_Output(X, Y, W, H, L) {
    type(t);
    box(t==FL_ROUND_CLOCK ? FL_NO_BOX : FL_UP_BOX);
}
```

Note:

From Duncan: (will be removed later, just for now as a reminder)

5. I've just added comments for the `fl_color_chooser()` functions, and in order to keep them and the general Function Reference information for them together, I created a new doxygen group, and used `\ingroup` in the three comment blocks. This creates a new Modules page (which may not be what we want) with links to it from the File Members and [FL_Color_Chooser.H](#) pages. It needs a bit more experimentation on my part unless someone already knows how this should be handled. (Maybe we can add it to a `functions.dox` file that defines a functions group and do that for all of the function documentation?)

Update: the trick is not to create duplicate entries in a new group, but to move the function information into the doxygen comments for the class, and use the navigation links provided. Simply using `\relatesalso` as the first doxygen command in the function's comment puts it in the appropriate place. There is no need to have `\defgroup` and `\ingroup` as well, and indeed they don't work. So, to summarize:

```
Gizmo.H
/## \class Gizmo
    A gizmo that does everything
#/
class Gizmo {
    etc
};
extern int popup_gizmo(...);

Gizmo.cxx:
/## \relatesalso Gizmo
    Pops up a gizmo dialog with a Gizmo in it
#/
int popup_gizmo(...);
```

Example comment:

You can use HTML comment statements to embed comments in doxygen comment blocks. These comments will not be visible in the generated document.

The following text is a developer comment.

This will be visible again.

```
The following text is a developer comment.
<!-- *** This *** is *** invisible *** -->
This will be visible again.
```

Different Headlines:

```
<H1>Headline in big text (H1)</H1>
<H2>Headline in big text (H2)</H2>
<H3>Headline in big text (H3)</H3>
<H4>Headline in big text (H4)</H4>
```

Headline in big text (H1)

Headline in big text (H2)

Headline in big text (H3)

Headline in big text (H4)

20.1 Non-ASCII characters

if you came here from below: back to [Creating Links](#)

Doxygen understands many HTML quoting characters like `";`, `ü;`, `ç;`, `Ç;`, but not all HTML quoting characters.

This will appear in the document:

Doxygen understands many HTML quoting characters like ", ü, ç, Ç, but not all HTML quoting characters.

For further informations about quoting see <http://www.stack.nl/~dimitri/doxygen/htmlcmds.html>

Example with UTF-8 encoded text

```

<P>Assuming that the following source code was written on MS Windows,
this example will output the correct label on OS X and X11 as well.
Without the conversion call, the label on OS X would read
<tt>Fahrvergn,gen</tt> with a deformed umlaut u ("cedille",
html "&cedil;").
\#code
    btn = new Fl_Button(10, 10, 300, 25);
    btn->copy_label(fl_latin1_to_local("Fahrvergnügen"));
\#endcode

\note    If your application uses characters that are not part of both
         encodings, or it will be used in areas that commonly use different
         code pages, you might consider upgrading to FLTK 2 which supports
         UTF-8 encoding.

\todo    This is an example todo entry, please ignore !

```

This will appear in the document:

Assuming that the following source code was written on MS Windows, this example will output the correct label on OS X and X11 as well. Without the conversion call, the label on OS X would read Fahrvergn,gen with a deformed umlaut u ("cedille", html "¸"). #code btn = new Fl_Button(10, 10, 300, 25); btn->copy_label(fl_latin1_to_local("Fahrvergnügen")); #endcode

Note:

If your application uses characters that are not part of both encodings, or it will be used in areas that commonly use different code pages, you might consider upgrading to FLTK 2 which supports UTF-8 encoding.

Todo

This is an example todo entry, please ignore !

20.2 Document Structure

- `\page` creates a named page
- `\section` creates a named section within that page
- `\subsection` creates a named subsection within the current section
- `\subsubsection` creates a named subsubsection within the current subsection

All these statements take a "name" as their first argument, and a title as their second argument. The title can contain spaces.

The page, section, and subsection titles are formatted in blue color and a size like "`<H1>`", "`<H2>`", and "`<H3>`", and "`<H4>`", respectively.

By **FLTK documentation convention**, a file like this one with a doxygen documentation chapter has the name "`<chapter>.dox`". The `\page` statement at the top of the page is "`\page <chapter> This is the title`". Sections within a documentation page must be called "`<chapter>_<section>`", where "`<chapter>`" is the name part of the file, and "`<section>`" is a unique section name within the page that can be referenced in links. The same for subsections and subsubsections.

These doxygen page and section commands work only in special documentation chapters, not within normal source or header documentation blocks. However, links **from** normal (e.g. class) documentation **to** documentation sections **do work**.

This page has

```
\page development I - Developer Information
```

at its top.

This section is

```
\section development_structure Document structure
```

The following section is

```
\section development_links Creating Links
```

20.3 Creating Links

Links to other documents and external links can be embedded with

- normal HTML links
- HTML links without markup - doxygen creates "`http://...`" links automatically
- links to other doxygen chapters with the `\ref` statements
- links to named sections within the same or other doxygen chapters, if they are defined there with a `\section` statement

see chapter `\ref unicode` creates a link to the named chapter `unicode` that has been created with a `\subpage` statement.

see `chapter 5` creates a link to a named html anchor `"character_encoding"` within the same file.

For further informations about quoting see <http://www.stack.nl/~dimitri/doxygen/htmlcmds.html>

Bold link text: you can see the `online documentation` of FLTK 1.3 at `\b http://www.fltk.org/doc-1.3/index.html`

see section `\ref development_non-ascii`

appears as:

see chapter [11 - Unicode and utf-8 Support](#) creates a link to the named chapter `unicode` that has been created with a `\subpage` statement.

see [chapter 5](#) creates a link to a named html anchor `"character_encoding"` within the same file.

For further informations about quoting see <http://www.stack.nl/~dimitri/doxygen/htmlcmds.html>

Bold link text: you can see the **online documentation** of FLTK 1.3 at <http://www.fltk.org/doc-1.3/index.html>

see section [Non-ASCII characters](#)

20.4 Changing Old Links

Old HTML links and anchors in text documentation pages should be changed as follows:

```
<H2><A name="event_xxx">Fl::event_*() methods</A></H2>
```

becomes:

```
<A NAME="event_xxx"></A> <!-- For old HTML links only ! -->
\section events_event_xxx Fl::event_*() methods
```

The additional HTML `""` statement is temporary needed, until all links (references) are modified, then:

```
<H2><A name="event_xxx">Fl::event_*() methods</A></H2>
```

becomes:

```
\section events_event_xxx Fl::event_*() methods
```

The `"\section"` statement can also be a `"\subsection"` or `"\subsubsection"` statement.

The references (in this example from `index.dox`) are changed as follows:

```
\subpage events
```

```
<B>
<UL>
  <LI><A HREF="events.html#event_xxx">Fl::event_*() methods</A></LI>
  <LI><A HREF="events.html#propagation">Event Propagation</A></LI>
</UL>
</B>
```

becomes:

```
\subpage events

<b>
  \li \ref events_event_xxx
  \li \ref events_propagation
</b>
```

20.5 Paragraph Layout

There is no real need to use HTML `<P>` and `</P>` tags within the text to tell doxygen to start or stop a paragraph. In most cases, when doxygen encounters a blank line or some, but not all, `\commands` in the text it knows that it has reached the start or end of a paragraph. Doxygen also offers the `\par` command for special paragraph handling. It can be used to provide a paragraph title and also to indent a paragraph. Unfortunately `\par` won't do what you expect if you want to have doxygen links and sometimes html tags don't work either.

```
\par Normal Paragraph with title

This paragraph will have a title, but because there is a blank line
between the \par and the text, it will have the normal layout.

\par Indented Paragraph with title
This paragraph will also have a title, but because there is no blank
line between the \par and the text, it will be indented.

\par
It is also possible to have an indented paragraph without title.
This is how you indent subsequent paragraphs.

\par No link to Fl_Widget::draw()
Note that the paragraph title is treated as plain text.
Doxygen type links will not work.
HTML characters and tags may or may not work.

Fl_Widget::draw() links and "html" tags work<br>
\par
Use a single line ending with <br> for complicated paragraph titles.
```

The above code produces the following paragraphs:

Normal Paragraph with title

This paragraph will have a title, but because there is a blank line between the `\par` and the text, it will have the normal layout.

Indented Paragraph with title

This paragraph will also have a title, but because there is no blank line between the `\par` and the text, it will be indented.

It is also possible to have an indented paragraph without title. This is how you indent subsequent paragraphs.

Chapter 21

J - Software License

December 11, 2001

The FLTK library and included programs are provided under the terms of the GNU Library General Public License (LGPL) with the following exceptions:

1. Modifications to the FLTK configure script, config header file, and makefiles by themselves to support a specific platform do not constitute a modified or derivative work.

The authors do request that such modifications be contributed to the FLTK project - send all contributions to "ftk-bugs@ftk.org".

2. Widgets that are subclassed from FLTK widgets do not constitute a derivative work.
3. Static linking of applications and widgets to the FLTK library does not constitute a derivative work and does not require the author to provide source code for the application or widget, use the shared FLTK libraries, or link their applications or widgets against a user-supplied version of FLTK.

If you link the application or widget to a modified version of FLTK, then the changes to FLTK must be provided under the terms of the LGPL in sections 1, 2, and 4.

4. You do not have to provide a copy of the FLTK license with programs that are linked to the FLTK library, nor do you have to identify the FLTK license in your program or documentation as required by section 6 of the LGPL.

However, programs must still identify their use of FLTK. The following example statement can be included in user documentation to satisfy this requirement:

[program/widget] is based in part on the work of the FLTK project (<http://www.fltk.org>).

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this

service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright

holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works

based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2

above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to

refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Chapter 22

K - Example Source Code

March 19, 2005

The FLTK distribution contains over 60 sample applications written in, or ported to, FLTK. If the FLTK archive you received does not contain a 'test' directory, you can download the complete FLTK distribution from <http://fltk.org/software.php>.

Most of the example programs were created while testing a group of widgets. They are not meant to be great achievements in clean C++ programming, but merely a test platform to verify the functionality of the FLTK library.

22.1 Example Applications

adjuster	arc	ask	bitmap	blocks	boxtype
browser	button	buttons	checkers	clock	colbrowser
color_chooser	cube	CubeView	cursor	curve	demo
doublebuffer	editor	fast_slow	file_chooser	fluid	fonts
forms	fractals	fullscreen	gl_overlay	glpuzzle	hello
help	iconize	image	inactive	input	input_choice
keyboard	label	line_style	list_visuals	mandelbrot	menubar
message	minimum	navigation	output	overlay	pack
pixmap_browser	pixmap	preferences	radio	resizebox	resize
scroll	shape	subwindow	sudoku	symbols	tabs
threads	tile	tiled_image	valuators		

22.1.1 adjuster

`adjuster` shows a nifty little widget for quickly setting values in a great range.

22.1.2 arc

The `arc` demo explains how to derive your own widget to generate some custom drawings. The sample drawings use the matrix based arc drawing for some fun effects.

22.1.3 ask

`ask` shows some of FLTK's standard dialog boxes. Click the correct answers or you may end up in a loop, or you may end up in a loop, or you... .

22.1.4 bitmap

This simple test shows the use of a single color bitmap as a label for a box widget. Bitmaps are stored in the X11 '.bmp' file format and can be part of the source code.

22.1.5 blocks

A wonderful and addictive game that shows the usage of FLTK timers, graphics, and how to implement sound on all platforms. `blocks` is also a good example for the Mac OS X specific bundle format.

22.1.6 boxtype

`boxtype` gives an overview of readily available boxes and frames in FLTK. More types can be added by the application programmer. When using themes, FLTK shuffles boxtypes around to give your program a new look.

22.1.7 browser

`browser` shows the capabilities of the [Fl_Browser](#) widget. Important features tested are loading of files, line formatting, and correct positioning of the browser data window.

22.1.8 button

The `button` test is a simple demo of push-buttons and callbacks.

22.1.9 buttons

`buttons` shows a sample of FLTK button types.

22.1.10 checkers

Written by Steve Poulsen in early 1979, `checkers` shows how to convert a VT100 text-terminal based program into a neat application with a graphical UI. Check out the code that drags the pieces, and how the pieces are drawn by layering. Then tell me how to beat the computer at Checkers.

22.1.11 clock

The `clock` demo shows two analog clocks. The innards of the [Fl_Clock](#) widget are pretty interesting, explaining the use of timeouts and matrix based drawing.

22.1.12 colbrowser

`colbrowser` runs only on X11 systems. It reads `/usr/lib/X11/rgb.txt` to show the color representation of every text entry in the file. This is beautiful, but only moderately useful unless your UI is written in *Motif*.

22.1.13 color_chooser

The `color_chooser` gives a short demo of FLTK's palette based color chooser and of the RGB based color wheel.

22.1.14 cube

The `cube` demo shows the speed of OpenGL. It also tests the ability to render two OpenGL buffers into a single window, and shows OpenGL text.

22.1.15 CubeView

`CubeView` shows how to create a UI containing OpenGL with Fluid.

22.1.16 cursor

The `cursor` demo shows all mouse cursor shapes that come standard with FLTK. The *fgcolor* and *bgcolor* sliders work only on few systems (some version of Irix for example).

22.1.17 curve

`curve` draws a nice Bezier curve into a custom widget. The *points* option for splines is not supported on all platforms.

22.1.18 demo

This tool allows quick access to all programs in the `test` directory. `demo` is based on the visuals of the IrixGL demo program. The menu tree can be changed by editing `test/demo.menu`.

22.1.19 doublebuffer

The `doublebuffer` demo show the difference between a single buffered window, which may flicker during a slow redraw, and a double buffered window, which never flickers, but uses twice the amount of RAM. Some modern OS's double buffer all windows automatically to allow transparency and shadows on the desktop. FLTK is smart enough to not tripple buffer a window in that case.

22.1.20 editor

FLTK has two very different text input widgets. `FL_Input` and derived classes are rather light weight, however `FL_Text_Editor` is a complete port of *nedit* (with permission). The `editor` test is almost a full application, showing custom syntax highlighting and dialog creation.

22.1.21 fast_slow

`fast_slow` shows how an application can use the `FL_Widget::when()` setting to receive different kinds of callbacks.

22.1.22 file_chooser

The standard FLTK `file_chooser` is the result of many iterations, trying to find a middle ground between a complex browser and a fast light implementation.

22.1.23 fonts

`fonts` shows all available text fonts on the host system. If your machine still has some pixmap based fonts, the supported sizes will be shown in bold face. Only the first 256 fonts will be listed.

22.1.24 forms

`forms` is an XForms program with very few changes. Search for "ftk" to find all changes necessary to port to fltk. This demo shows the different boxtypes. Note that some boxtypes are not appropriate for some objects.

22.1.25 fractals

`fractals` shows how to mix OpenGL, Glut and FLTK code. FLTK supports a rather large subset of Glut, so that many Glut applications compile just fine.

22.1.26 fullscreen

This demo shows how to do many of the window manipulations that are popular for games. You can toggle the border on/off, switch between single- and double-buffered rendering, and take over the entire screen. More information in the source code.

22.1.27 gl_overlay

`gl_overlay` shows OpenGL overlay plane rendering. If no hardware overlay plane is available, FLTK will simulate it for you.

22.1.28 glpuzzle

The `glpuzzle` test shows how most Glut source code compiles easily under FLTK.

22.1.29 hello

`hello`: Hello, World. Need I say more? Well, maybe. This tiny demo shows how little is needed to get a functioning application running with FLTK. Quite impressive, I'd say.

22.1.30 help

`help` displays the built-in FLTK help browser. The [FL_Help_Dialog](#) understands a subset of html and renders various image formats. This widget makes it easy to provide help pages to the user without depending on the operating system's html browser.

22.1.31 iconize

`iconize` demonstrates the effect of the window functions `hide()`, `iconize()`, and `show()`.

22.1.32 image

The `image` demo shows how an image can be created on the fly. This generated image contains an alpha (transparency) channel which lets previous renderings 'shine through', either via true transparency or by using screen door transparency (pixelation).

22.1.33 inactive

`inactive` tests the correct rendering of inactive widgets. To see the inactive version of images, you can check out the `pixmap` or `image` test.

22.1.34 input

This tool shows and tests different types of text input fields based on `Fl_Input_`. The `input` program also tests various settings of `Fl_Input::when()`.

22.1.35 input_choice

`input_choice` tests the latest addition to FLTK1, a text input field with an attached pulldown menu. Windows users will recognize similarities to the 'ComboBox'. `input_choice` starts up in 'plastic' scheme, but the traditional scheme is also supported.

22.1.36 keyboard

FLTK unifies keyboard events for all platforms. The `keyboard` test can be used to check the return values of `Fl::event_key()` and `Fl::event_text()`. It is also great to see the modifier buttons and the scroll wheel at work. Quit this application by closing the window. The ESC key will not work.

22.1.37 label

Every FLTK widget can have a label attached to it. The `label` demo shows alignment, clipping and wrapping of text labels. Labels can contain symbols at the start and end of the text, like `@FLTK` or `@circle uh-huh @square`.

22.1.38 line_style

Advanced line drawing can be tested with `line_style`. Not all platforms support all line styles.

22.1.39 list_visuals

This little app finds all available pixel formats for the current X11 screen. But since you are now an FLTK user, you don't have to worry about any of this.

22.1.40 mandelbrot

`mandelbrot` shows two advanced topics in one test. It creates grayscale images on the fly, updating them via the *idle* callback system. This is one of the few occasions where the *idle* callback is very useful by giving all available processor time to the application without blocking the UI or other apps.

22.1.41 menubar

The `menubar` tests many aspects of FLTK's popup menu system. Among the features are radio buttons, menus taller than the screen, arbitrary sub menu depth, and global shortcuts.

22.1.42 message

`message` pops up a few of FLTK's standard message boxes.

22.1.43 minimum

The `minimum` test program verifies that the update regions are set correctly. In a real life application, the trail would be avoided by choosing a smaller label or by setting label clipping differently.

22.1.44 navigation

`navigation` demonstrates how the text cursor moves from text field to text field when using the arrow keys, tab, and shift-tab.

22.1.45 output

`output` shows the difference between the single line and multi line mode of the [Fl_Output](#) widget. Fonts can be selected from the FLTK standard list of fonts.

22.1.46 overlay

The `overlay` test app show how easy an FLTK window can be layered to display cursor and manipulator style elements. This example derives a new class from [Fl_Overlay_Window](#) and provides a new function to draw custom overlays.

22.1.47 pack

The `pack` test program demonstrates the resizing and repositioning of children of the [FL_Pack](#) group. Putting an [FL_Pack](#) into an [FL_Scroll](#) is a useful way to create a browser for large sets of data.

22.1.48 pixmap_browser

`pixmap_browser` tests the shared-image interface. When using the same image multiple times, [FL_Shared_Image](#) will keep it only once in memory.

22.1.49 pixmap

This simple test shows the use of a LUT based pixmap as a label for a box widget. Pixmapes are stored in the X11 '.xpm' file format and can be part of the source code. Pixmapes support one transparent color.

22.1.50 preferences

I do have my `preferences` in the morning, but sometimes I just can't remember a thing. This is where the [FL_Preferences](#) come in handy. They remember any kind of data between program launches.

22.1.51 radio

The `radio` tool was created entirely with *fluid*. It shows some of the available button types and tests radio button behavior.

22.1.52 resizebox

`resizebox` shows some possible ways of FLTK's automatic resize behavior..

22.1.53 resize

The `resize` demo tests size and position functions with the given window manager.

22.1.54 scroll

`scroll` shows how to scroll an area of widgets, one of them being a slow custom drawing. [FL_Scroll](#) uses clipping and smart window area copying to improve redraw speed. The buttons at the bottom of the window control decoration rendering and updates.

22.1.55 shape

`shape` is a very minimal demo that shows how to create your own OpenGL rendering widget. Now that you know that, go ahead and write that flight simulator you always dreamt of.

22.1.56 subwindow

The `subwindow` demo tests messaging and drawing between the main window and 'true' sub windows. A sub window is different to a group by resetting the FLTK coordinate system to 0, 0 in the top left corner. On Win32 and X11, subwindows have their own operating system specific handle.

22.1.57 sudoku

Another highly addictive game - don't play it, I warned you. The implementation shows how to create application icons, how to deal with OS specifics, and how to generate sound.

22.1.58 symbols

`symbols` are a speciality of FLTK. These little vector drawings can be integrated into labels. They scale and rotate, and with a little patience, you can define your own. The rotation number refers to 45 degree rotations if you were looking at a numeric keypad (2 is down, 6 is right, etc.).

22.1.59 tabs

The `tabs` tool was created with *fluid*. It tests correct hiding and redisplaying of tabs, navigation across tabs, resize behavior, and no unneeded redrawing of invisible widgets.

The `tabs` application shows the [Fl_Tabs](#) widget on the left and the [Fl_Wizard](#) widget on the right side for direct comparison of these two panel management widgets.

22.1.60 threads

FLTK can be used in a multithreading environment. There are some limitations, mostly due to the underlying operating system. `threads` shows how to use [Fl::lock\(\)](#), [Fl::unlock\(\)](#), and [Fl::awake\(\)](#) in secondary threads to keep FLTK happy. Although locking works on all platforms, this demo is not available on every machine.

22.1.61 tile

The `tile` tool shows a nice way of using [FL_Tile](#). To test correct resizing of subwindows, the widget for region 1 is created from an [FL_Window](#) class.

22.1.62 tiled_image

The `tiled_image` demo uses an image as the background for a window by repeating it over the full size of the widget. The window is resizable and shows how the image gets repeated.

22.1.63 valuator

`valuators` shows all of FLTK's nifty widgets to change numeric values.

22.1.64 fluid

`fluid` is not only a big test program, but also a very useful visual UI designer. Many parts of `fluid` were created using `fluid`. See the [Fluid Tutorial](#) for more details.

Chapter 23

Deprecated List

Member **Fl::release()** Use **Fl::grab(0)** instead.

Member **Fl::set_idle(void(*cb)())** This method is obsolete - use the **add_idle()** method instead.

Member **Fl_Group::focus(Fl_Widget *W)** This is for backwards compatibility only. You should use *W->take_focus()* instead.

Member **Fl_Menu_Item::check()** .

Member **Fl_Menu_Item::checked()** const .

Member **Fl_Menu_Item::uncheck()** .

Member **Fl_Spinner::maximum()** const

Member **Fl_Spinner::mininum()** const

Member **Fl_Widget::color2(unsigned a)** Use **selection_color(unsigned)** instead.

Member **Fl_Widget::color2()** const Use **selection_color()** instead.

Chapter 24

Todo List

Member [Fl_Browser::draw\(int, int, int, int\)](#) Find the implementation, if any, and document it there!

Class [Fl_Button](#) Refactor the doxygen comments for [Fl_Button](#) type() documentation.

Class [Fl_Button](#) Refactor the doxygen comments for [Fl_Button](#) when() documentation.

Class [Fl_Chart](#) Refactor [Fl_Chart::type\(\)](#) information.

Class [Fl_Check_Button](#) Refactor [Fl_Check_Button](#) doxygen comments (add color() info etc?)

Class [Fl_Check_Button](#) Generate [Fl_Check_Button.gif](#) with visible checkmark.

Class [Fl_Choice](#) Refactor the doxygen comments for [Fl_Choice](#) changed() documentation.

Member [Fl_Clock::update\(\)](#) Find [Fl_Clock::update\(\)](#) implementation, if any, and document it.

Class [Fl_Counter](#) Refactor the doxygen comments for [Fl_Counter](#) type() documentation.

Member [Fl_File_Input::errorcolor\(\)](#) const Better docs for [Fl_File_Input::errorcolor\(\)](#) - is it even used?

Member [Fl_Group::sizes\(\)](#) Should the internal representation of the sizes() array be documented?

Class [Fl_Label](#) For FLTK 1.3, the [Fl_Label](#) type will become a widget by itself. That way we will be avoiding a lot of code duplication by handling labels in a similar fashion to widgets containing text. We also provide an easy interface for very complex labels, containing html or vector graphics.

Member [Fl_Preferences::get\(const char *entry, void *value, const void *defaultValue, int defaultSize, int maxSize\)](#) maxSize should receive the number of bytes that were read.

Member [Fl_Scroll::bbox\(int &, int &, int &, int &\)](#) The visibility of the scrollbars ought to be checked/calculated outside of the draw() method (STR #1895).

Member [Fl_Text_Display::shortcut\(int s\)](#) FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Member [Fl_Text_Display::shortcut\(\)](#) const FIXME : get set methods pointing on shortcut_ have no effects as shortcut_ is unused in this class and derived!

Class [Fl_Text_Selection](#) members must be documented

Member [Fl_Widget::align\(\)](#) `const` This function should not take `uchar` as an argument. Apart from the fact that `uchar` is too short with only 8 bits, it does not provide type safety (in which case we don't need to declare `Fl_Align` an enum to begin with). NOTE* The current (FLTK 1.3) implementation (Dec 2008) is such that `Fl_Align` is (typedef'd to be) "unsigned" (int), but `Fl_Widget`'s "align_" member variable is a bit field of 8 bits only !

Member [Fl_Widget::argument\(\)](#) `(long v)` The user data value must be implemented using a *union* to avoid 64 bit machine incompatibilities.

Member [Fl_Widget::type\(\)](#) `const` Explain "simulate RTTI" (currently only used to decide if a widget is a window, i.e. `type() >= FL_WINDOW` ?). Is `type()` really used in a way that ensures "Forms compatibility" ?

Member [Fl_Color](#) enum `Fl_Color` needs some more comments for values, see `Fl/Enumerations.H`

Member [Fl_When](#) doxygen comments for values are incomplete and maybe wrong or unclear

Member [Fl_Labeltype](#) The doxygen comments are incomplete, and some labeltypes are starting with an underscore. Also, there are three external functions undocumented (yet):

- `fl_define_FL_SHADOW_LABEL()`
- `fl_define_FL_ENGRAVED_LABEL()`
- `fl_define_FL_EMBOSSED_LABEL()`

Member [Fl_String](#) FIXME: temporary (?) typedef to mark UTF8 and Unicode conversions

Member [fl_height](#) Is `fl_height(int, int size)` required for `Fl_Text_Display`? Why not use `size` parameter directly?

Member [fl_reset_spot](#) provide user documentation for `fl_reset_spot` function

Member [fl_set_spot](#) provide user documentation for `fl_set_spot` function

Member [fl_set_status](#) provide user documentation for `fl_set_status` function

Page 3 - Common Widgets and Attributes correct signatures for `label()` and friends to produce links in `common.dox`

Page 5 - Drawing Things in FLTK drawing.dox: I fixed the above encoding problem of these ¸ and umlaut characters, but this text is obsoleted by FLTK 1.3 with utf-8 encoding, or must be rewritten accordingly: How to use native (e.g. Windows "ANSI", or ISO-8859-x) encoding in embedded strings for labels, error messages and more. Please check this (utf-8) encoding on different OS'es and with different language and font environments.

Page 6 - Handling Events Does Fltk Compose Character Sequences text need updating after the addition of UTF-8 handling to FLTK-1.3.x ?

Page 7 - Adding and Extending Widgets Clarify `Fl_Window::damage(n)` handling - seems confused/wrong? ORing value doesn't match setting behaviour in `FL_Widget.H`!

Page 7 - Adding and Extending Widgets Clarify `Fl_Widget::test_shortcut()` explanations. `Fl_Widget.h` says Internal Use only, but subclassing chapter gives details!

Page 7 - Adding and Extending Widgets The new method `draw_children` should be documented below.

Page I - Developer Information This is an example todo entry, please ignore !

Page I - Developer Information HACK* : include image file for footer. Doxygen does not include the file "tiny.gif" from "html_footer" in its output html dir. Find out, how this can be done, or avoid using an image in the HTML footer.

Chapter 25

Module Index

25.1 Modules

Here is a list of all modules:

Windows handling functions	219
Events handling functions	222
Selection & Clipboard functions	232
Screen functions	235
Color & Font functions	237
Drawing functions	246
Multithreading support functions	265
Safe widget deletion support functions	267
Cairo support functions and classes	270
Common Dialogs classes and functions	272
File names and URI utility functions	279

Chapter 26

Class Index

26.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Fl	283
Fl_Cairo_State	347
FL_CHART_ENTRY	356
Fl_End	389
Fl_File_Chooser	393
Fl_File_Icon	402
Fl_Font_Descriptor	414
Fl_Glut_Bitmap_Font	430
Fl_Help_Dialog	444
Fl_Help_Font_Style	448
Fl_Help_Link	449
Fl_Help_Target	450
Fl_Image	457
Fl_Bitmap	308
Fl_XBM_Image	725
Fl_Pixmap	525
Fl_GIF_Image	422
Fl_XPM_Image	726
Fl_RGB_Image	557
Fl_BMP_Image	311
Fl_JPEG_Image	480
Fl_PNG_Image	529
Fl_PNM_Image	530
Fl_Shared_Image	578
Fl_Tiled_Image	647
Fl_Label	481
Fl_Menu_Item	500
Fl_Preferences	535
Fl_Preferences::Entry	547
Fl_Preferences::Name	548
Fl_Text_Buffer	599
Fl_Text_Display::Style_Table_Entry	633
Fl_Text_Editor::Key_Binding	641

Fl_Text_Selection	642
Fl_Tooltip	654
Fl_Widget	677
Fl_Box	312
Fl_Button	342
Fl_Light_Button	483
Fl_Check_Button	361
Fl_Round_Button	564
Fl_Repeat_Button	552
Fl_Return_Button	554
Fl_Toggle_Button	653
Fl_Chart	350
Fl_Clock_Output	370
Fl_Clock	367
Fl_Round_Clock	566
Fl_FormsBitmap	415
Fl_FormsPixmap	417
Fl_Free	419
Fl_Group	434
Fl_Browser_	332
Fl_Browser	314
Fl_File_Browser	390
Fl_Hold_Browser	456
Fl_Multi_Browser	515
Fl_Select_Browser	577
Fl_Check_Browser	357
Fl_Color_Chooser	374
Fl_Help_View	451
Fl_Input_Choice	475
Fl_Pack	523
Fl_Scroll	567
Fl_Spinner	590
Fl_Tabs	595
Fl_Text_Display	616
Fl_Text_Editor	634
Fl_Tile	644
Fl_Window	712
Fl_Double_Window	386
Fl_Cairo_Window	348
Fl_Overlay_Window	520
Fl_Gl_Window	423
Fl_Glut_Window	431
Fl_Single_Window	584
Fl_Menu_Window	512
Fl_Wizard	723
Fl_Input_	466
Fl_Input	462
Fl_File_Input	408
Fl_Float_Input	413
Fl_Int_Input	479
Fl_Multiline_Input	516
Fl_Output	518

Fl_Multiline_Output	517
Fl_Secret_Input	576
Fl_Menu_	486
Fl_Choice	363
Fl_Menu_Bar	494
Fl_Menu_Button	497
Fl_Positioner	531
Fl_Progress	550
Fl_Timer	650
Fl_Valuator	658
Fl_Adjuster	305
Fl_Counter	379
Fl_Simple_Counter	583
Fl_Dial	383
Fl_Fill_Dial	411
Fl_Roller	561
Fl_Slider	586
Fl_Fill_Slider	412
Fl_Scrollbar	572
Fl_Value_Slider	674
Fl_Value_Input	664
Fl_Value_Output	670
Fl_Widget_Tracker	710

Chapter 27

Class Index

27.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Fl (The Fl is the FLTK global (static) containing state information and global methods for the current application)	283
Fl_Adjuster (Was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range)	305
Fl_Bitmap (Supports caching and drawing of mono-color (bitmap) images)	308
Fl_BMP_Image (Supports loading, caching, and drawing of Windows Bitmap (BMP) image files)	311
Fl_Box (This widget simply draws its box, and possibly it's label)	312
Fl_Browser (Displays a scrolling list of text lines, and manages all the storage for the text) . . .	314
Fl_Browser_ (This is the base for browsers)	332
Fl_Button (Buttons generate callbacks when they are clicked by the user)	342
Fl_Cairo_State (Contains all the necessary info on the current cairo context)	347
Fl_Cairo_Window (This defines a pre-configured cairo fltk window)	348
Fl_Chart (Fl_Chart displays simple charts)	350
FL_CHART_ENTRY (For internal use only)	356
Fl_Check_Browser (Displays a scrolling list of text lines that may be selected and/or checked by the user)	357
Fl_Check_Button (A button with an "checkmark" to show its status)	361
Fl_Choice (A button that is used to pop up a menu)	363
Fl_Clock (This widget provides a round analog clock display)	367
Fl_Clock_Output (This widget can be used to display a program-supplied time)	370
Fl_Color_Chooser (Standard RGB color chooser)	374
Fl_Counter (Controls a single floating point value with button (or keyboard) arrows)	379
Fl_Dial (Circular dial to control a single floating point value)	383
Fl_Double_Window (The Fl_Double_Window provides a double-buffered window)	386
Fl_End (This is a dummy class that allows you to end a Fl_Group in a constructor list of a class:)	389
Fl_File_Browser (Displays a list of filenames, optionally with file-specific icons)	390
Fl_File_Chooser (Displays a standard file selection dialog that supports various selection modes)	393
Fl_File_Icon (Manages icon images that can be used as labels in other widgets and as icons in the FileBrowser widget)	402
Fl_File_Input (This widget displays a pathname in a text input field)	408
Fl_Fill_Dial (Draws a dial with a filled arc)	411
Fl_Fill_Slider (Widget that draws a filled horizontal slider, useful as a progress or value meter) .	412

Fl_Float_Input (Subclass of Fl_Input that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits))	413
Fl_Font_Descriptor (This a structure for an actual system font, with junk to help choose it and info on character sizes)	414
Fl_FormsBitmap (Forms compatibility Bitmap Image Widget)	415
Fl_FormsPixmap (Forms pixmap drawing routines)	417
Fl_Free (Emulation of the Forms "free" widget)	419
Fl_GIF_Image (Supports loading, caching, and drawing of Compuserve GIF SM images)	422
Fl_Gl_Window (Sets things up so OpenGL works)	423
Fl_Glut_Bitmap_Font (Fltk glut font/size attributes used in the glutXXX functions)	430
Fl_Glut_Window (GLUT is emulated using this window class and these static variables (plus several more static variables hidden in glut_compatibility.cxx):)	431
Fl_Group (FLTK container widget)	434
Fl_Help_Dialog (Displays a standard help dialog window using the Fl_Help_View widget) . . .	444
Fl_Help_Font_Style (Fl_Help_View font stack element definition)	448
Fl_Help_Link (Definition of a link for the html viewer)	449
Fl_Help_Target (Fl_Help_Target structure)	450
Fl_Help_View (Displays HTML text)	451
Fl_Hold_Browser (The Fl_Hold_Browser is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space)	456
Fl_Image (Fl_Image is the base class used for caching and drawing all kinds of images in FLTK)	457
Fl_Input (This is the FLTK text input widget)	462
Fl_Input_ (This is a virtual base class below Fl_Input)	466
Fl_Input_Choice (A combination of the input widget and a menu button)	475
Fl_Int_Input (Subclass of Fl_Input that only allows the user to type decimal digits (or hex numbers of the form 0xaeF))	479
Fl_JPEG_Image (Supports loading, caching, and drawing of Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) images)	480
Fl_Label (This struct stores all information for a text or mixed graphics label)	481
Fl_Light_Button	483
Fl_Menu_ (Base class of all widgets that have a menu in FLTK)	486
Fl_Menu_Bar (This widget provides a standard menubar interface)	494
Fl_Menu_Button (This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of Fl_Menu_Item objects)	497
Fl_Menu_Item (The Fl_Menu_Item structure defines a single menu item that is used by the Fl_Menu_ class)	500
Fl_Menu_Window (Window type used for menus)	512
Fl_Multi_Browser (Subclass of Fl_Browser which lets the user select any set of the lines) . . .	515
' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys)	516
Fl_Multiline_Output (This widget is a subclass of Fl_Output that displays multiple lines of text)	517
Fl_Output (This widget displays a piece of text)	518
Fl_Overlay_Window (This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image)	520
Fl_Pack (This widget was designed to add the functionality of compressing and aligning widgets)	523
Fl_Pixmap (Supports caching and drawing of colormap (pixmap) images, including transparency)	525
Fl_PNG_Image (Supports loading, caching, and drawing of Portable Network Graphics (PNG) image files)	529
Fl_PNM_Image (Supports loading, caching, and drawing of Portable Anymap (PNM, PBM, PGM, PPM) image files)	530
Fl_Positioner (This class is provided for Forms compatibility)	531
Fl_Preferences (Fl_Preferences provides methods to store user settings between application starts)	535
Fl_Preferences::Entry (An entry associates a preference name to its corresponding value)	547

Fl_Preferences::Name ('Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly)	548
Fl_Progress (Displays a progress bar for the user)	550
Fl_Repeat_Button (The Fl_Repeat_Button is a subclass of Fl_Button that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down)	552
Fl_Return_Button (The Fl_Return_Button is a subclass of Fl_Button that generates a callback when it is pressed or when the user presses the Enter key)	554
Fl_RGB_Image (Supports caching and drawing of full-color images with 1 to 4 channels of color information)	557
Fl_Roller ("dolly" control commonly used to move 3D objects)	561
Fl_Round_Button (Buttons generate callbacks when they are clicked by the user)	564
Fl_Round_Clock (A clock widget of type <code>FL_ROUND_CLOCK</code>)	566
Fl_Scroll (This container widget lets you maneuver around a set of widgets much larger than your window)	567
Fl_Scrollbar (Displays a slider with arrow buttons at the ends of the scrollbar)	572
Fl_Secret_Input (Subclass of Fl_Input that displays its input as a string of asterisks)	576
Fl_Select_Browser (The class is a subclass of Fl_Browser which lets the user select a single item, or no items by clicking on the empty space)	577
Fl_Shared_Image (This class supports caching, loading, and drawing of image files)	578
Fl_Simple_Counter (This widget creates a counter with only 2 arrow buttons)	583
Fl_Single_Window (This is the same as Fl_Window)	584
Fl_Slider (Sliding knob inside a box)	586
Fl_Spinner (This widget is a combination of the input widget and repeat buttons)	590
Fl_Tabs ("file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits)	595
Fl_Text_Buffer (Used by the Fl_Text_Display and Fl_Text_Editor to manage complex text data and is based upon the excellent NEdit text editor engine - see http://www.nedit.org/)	599
Fl_Text_Display (This is the FLTK text display widget)	616
Fl_Text_Display::Style_Table_Entry (This structure associates the color,font,size of a string to draw with an attribute mask matching attr)	633
Fl_Text_Editor (This is the FLTK text editor widget)	634
Fl_Text_Editor::Key_Binding (Simple linked list associating a key/state to a function)	641
Fl_Text_Selection (This is an internal class for Fl_Text_Buffer to manage text selections)	642
Fl_Tile (Lets you resize the children by dragging the border between them:)	644
Fl_Tiled_Image (This class supports tiling of images over a specified area)	647
Fl_Timer (This is provided only to emulate the Forms Timer widget)	650
Fl_Toggle_Button (The toggle button is a push button that needs to be clicked once to toggle on, and one more time to toggle off)	653
Fl_Tooltip (Tooltip support for all FLTK widgets)	654
Fl_Valuator (Controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object)	658
Fl_Value_Input (Displays a numeric value)	664
Fl_Value_Output (Displays a floating point value)	670
Fl_Value_Slider (Fl_Slider widget with a box displaying the current value)	674
Fl_Widget (Fl_Widget is the base class for all widgets in FLTK)	677
Fl_Widget_Tracker (This class should be used to control safe widget deletion)	710
Fl_Window (This widget produces an actual window)	712
Fl_Wizard (This widget is based off the Fl_Tabs widget, but instead of displaying tabs it only changes "tabs" under program control)	723
Fl_XBM_Image (Supports loading, caching, and drawing of X Bitmap (XBM) bitmap files)	725
Fl_XPM_Image (Supports loading, caching, and drawing of X Pixmap (XPM) images, including transparency)	726

Chapter 28

File Index

28.1 File List

Here is a list of all documented files with brief descriptions:

aimm.h	??
armscii_8.h	??
ascii.h	??
big5.h	??
big5_emacs.h	??
case.h	??
cgdebug.h	??
cp1133.h	??
cp1251.h	??
cp1255.h	??
cp1256.h	??
dingbats_.h	??
dirent.h	??
Enumerations.H (This file contains type definitions and general enumerations)	727
fastarrow.h	??
filename.H	??
Fl.H	??
Fl_Adjuster.H	??
fl_arc.cxx (Utility functions for drawing arcs and circles)	741
fl_arci.cxx (Utility functions for drawing circles using integers)	742
fl_ask.H	??
Fl_Bitmap.H	??
Fl_BMP_Image.H	??
Fl_Box.H	??
fl_boxtype.cxx (Drawing code for common box types)	743
Fl_Browser.H	??
Fl_Browser_.H	??
Fl_Button.H	??
Fl_Cairo.H	??
Fl_Cairo_Window.H	??
Fl_Chart.H	??
Fl_Check_Browser.H	??
Fl_Check_Button.H	??

Fl_Choice.H	??
Fl_Clock.H	??
fl_cmap.h	??
fl_color.cxx (Color handling)	745
Fl_Color_Chooser.H (Fl_Color_Chooser widget)	747
Fl_Counter.H	??
fl_curve.cxx (Utility for drawing Bezier curves, adding the points to the current fl_begin/fl_- vertex/fl_end path)	748
Fl_Dial.H	??
Fl_Double_Window.H	??
fl_draw.H (Utility header to pull drawing functions together)	749
Fl_Export.H	??
Fl_File_Browser.H	??
Fl_File_Chooser.H	??
Fl_File_Icon.H	??
Fl_File_Input.H	??
Fl_Fill_Dial.H	??
Fl_Fill_Slider.H	??
Fl_Float_Input.H	??
Fl_Font.H	??
Fl_FormsBitmap.H	??
Fl_FormsPixmap.H	??
Fl_Free.H	??
Fl_GIF_Image.H	??
Fl_Gl_Choice.H	??
Fl_Gl_Window.H	??
Fl_Group.H	??
Fl_Help_Dialog.H	??
Fl_Help_View.H	??
Fl_Hold_Browser.H	??
Fl_Hor_Fill_Slider.H	??
Fl_Hor_Nice_Slider.H	??
Fl_Hor_Slider.H	??
Fl_Hor_Value_Slider.H	??
Fl_Image.H	??
Fl_Input.H	??
Fl_Input_.H	??
Fl_Input_Choice.H	??
Fl_Int_Input.H	??
Fl_JPEG_Image.H	??
Fl_Light_Button.H	??
Fl_Line_Dial.H	??
fl_line_style.cxx (Line style drawing utility hiding different platforms)	757
Fl_Menu.H	??
Fl_Menu_.H	??
Fl_Menu_Bar.H	??
Fl_Menu_Button.H	??
Fl_Menu_Item.H	??
Fl_Menu_Window.H	??
fl_message.H	??
Fl_Multi_Browser.H	??
Fl_Multi_Label.H	??
Fl_Multiline_Input.H	??
Fl_Multiline_Output.H	??

Fl_Nice_Slider.H	??
Fl_Object.H	??
Fl_Output.H	??
Fl_Overlay_Window.H	??
Fl_Pack.H	??
Fl_Pixmap.H	??
Fl_PNG_Image.H	??
Fl_PNM_Image.H	??
Fl_Positioner.H	??
Fl_Preferences.H	??
Fl_Progress.H	??
Fl_Radio_Button.H	??
Fl_Radio_Light_Button.H	??
Fl_Radio_Round_Button.H	??
fl_rect.cxx (Drawing and clipping routines for rectangles)	758
Fl_Repeat_Button.H	??
Fl_Return_Button.H	??
Fl_RGB_Image.H	??
Fl_Roller.H	??
Fl_Round_Button.H	??
Fl_Round_Clock.H	??
Fl_Scroll.H	??
Fl_Scrollbar.H	??
Fl_Secret_Input.H	??
Fl_Select_Browser.H	??
Fl_Shared_Image.H	??
fl_show_colormap.H	??
fl_show_input.H	??
Fl_Simple_Counter.H	??
Fl_Single_Window.H	??
Fl_Slider.H	??
Fl_Spinner.H	??
Fl_Sys_Menu_Bar.H	??
Fl_Tabs.H	??
Fl_Text_Buffer.H	??
Fl_Text_Display.H	??
Fl_Text_Editor.H	??
Fl_Tile.H	??
Fl_Tiled_Image.H	??
Fl_Timer.H	??
Fl_Toggle_Button.H	??
Fl_Toggle_Light_Button.H	??
Fl_Toggle_Round_Button.H	??
Fl_Tooltip.H	??
fl_types.h (This file contains simple "C"-style type definitions)	761
fl_utf8.h	??
Fl_Valuator.H	??
Fl_Value_Input.H	??
Fl_Value_Output.H	??
Fl_Value_Slider.H	??
fl_vertex.cxx (Portable drawing code for drawing arbitrary shapes with simple 2D transformations)	762
Fl_Widget.H	??
Fl_Window.H	??

Fl_Wizard.H	??
Fl_XBM_Image.H	??
Fl_XColor.H	??
Fl_XPM_Image.H	??
flstring.h	??
forms.H	??
freeglut_teapot_data.h	??
gb2312.h	??
georgian_academy.h	??
georgian_ps.h	??
gl.h	??
gl2opengl.h	??
gl_draw.H	??
glu.h	??
glut.H	??
iso8859_1.h	??
iso8859_10.h	??
iso8859_11.h	??
iso8859_13.h	??
iso8859_14.h	??
iso8859_15.h	??
iso8859_16.h	??
iso8859_2.h	??
iso8859_3.h	??
iso8859_4.h	??
iso8859_5.h	??
iso8859_6.h	??
iso8859_7.h	??
iso8859_8.h	??
iso8859_9.h	??
iso8859_9e.h	??
jisx0201.h	??
jisx0208.h	??
jisx0212.h	??
koi8_c.h	??
koi8_r.h	??
koi8_u.h	??
ksc5601.h	??
mac.H	??
math.h	??
mediumarrow.h	??
mulelao.h	??
names.h	??
slowarrow.h	??
spacing.h	??
symbol_.h	??
tatar_cyr.h	??
tcvn.h	??
tis620.h	??
ucs2be.h	??
utf8.h	??
viscii.h	??
win32.H	??
x.H	??

Ximint.h	..	??
Xlibint.h	..	??
Xutf8.h	..	??

Chapter 29

Module Documentation

29.1 Windows handling functions

Windows and standard dialogs handling.

Functions

- static void `Fl::default_atclose` (`Fl_Window *`, void *)
Default callback for window widgets.
- static void `Fl::first_window` (`Fl_Window *`)
See `Fl_Window` `first_window()`.*
- static `Fl_Window *` `Fl::first_window` ()
Returns the first top-level window in the list of `shown()` windows.
- static void `Fl::grab` (`Fl_Window *`)
Selects the window to grab.
- static `Fl_Window *` `Fl::grab` ()
This is used when pop-up menu systems are active.
- static `Fl_Window *` `Fl::modal` ()
Returns the top-most `modal()` window currently shown.
- static `Fl_Window *` `Fl::next_window` (const `Fl_Window *`)
Returns the next top-level window in the list of `shown()` windows.
- static void `Fl::set_abort` (void(*f)(const char *,...))
For back compatibility, sets the void `Fl::fatal` handler callback.
- static void `Fl::set_atclose` (void(*f)(`Fl_Window *`, void *))
For back compatibility, sets the `Fl::atclose` handler callback.

Variables

- static void(* [Fl::atclose](#))(FL_Window *, void *) = default_atclose

Back compatibility: default window callback handler.

29.1.1 Detailed Description

Windows and standard dialogs handling.

29.1.2 Function Documentation

29.1.2.1 void [Fl::default_atclose](#) (FL_Window * *window*, void * *v*) [static, inherited]

Default callback for window widgets.

It hides the window and then calls the default widget callback.

29.1.2.2 FL_Window * [Fl::first_window](#) () [static, inherited]

Returns the first top-level window in the list of shown() windows.

If a [modal\(\)](#) window is shown this is the top-most modal window, otherwise it is the most recent window to get an event.

The second form sets the window that is returned by first_window. The window is removed from wherever it is in the list and inserted at the top. This is not done if [Fl::modal\(\)](#) is on or if the window is not shown(). Because the first window is used to set the "parent" of modal windows, this is often useful.

29.1.2.3 void [Fl::grab](#) (FL_Window * *win*) [static, inherited]

Selects the window to grab.

See FL_Window* [Fl::grab\(\)](#)

29.1.2.4 static FL_Window* [Fl::grab](#) () [inline, static, inherited]

This is used when pop-up menu systems are active.

Send all events to the passed window no matter where the pointer or focus is (including in other programs). The window *does not have to be shown()*, this lets the [handle\(\)](#) method of a "dummy" window override all event handling and allows you to map and unmap a complex set of windows (under both X and WIN32 some window must be mapped because the system interface needs a window id).

If [grab\(\)](#) is on it will also affect show() of windows by doing system-specific operations (on X it turns on override-redirect). These are designed to make menus pop up reliably and faster on the system.

To turn off grabbing do [Fl::grab\(0\)](#).

Be careful that your program does not enter an infinite loop while [grab\(\)](#) is on. On X this will lock up your screen! To avoid this potential lockup, all newer operating systems seem to limit mouse pointer grabbing to the time during which a mouse button is held down. Some OS's may not support grabbing at all.

29.1.2.5 static `Fl_Window* Fl::modal()` `[inline, static, inherited]`

Returns the top-most `modal()` window currently shown.

This is the most recently shown() window with `modal()` true, or NULL if there are no `modal()` windows shown(). The `modal()` window has its `handle()` method called for all events, and no other windows will have `handle()` called (`grab()` overrides this).

29.1.2.6 `Fl_Window * Fl::next_window (const Fl_Window * window)` `[static, inherited]`

Returns the next top-level window in the list of shown() windows.

You can use this call to iterate through all the windows that are shown().

29.1.2.7 static void `Fl::set_atclose (void(*)(Fl_Window *, void *)f)` `[inline, static, inherited]`

For back compatibility, sets the `Fl::atclose` handler callback.

You can now simply change the callback for the window instead.

See also:

[Fl_Window::callback\(Fl_Callback*\)](#)

29.1.3 Variable Documentation

29.1.3.1 void(* `Fl::atclose`)(Fl_Window *, void *) (Fl_Window *, void *) = `default_atclose` `[static, inherited]`

Back compatibility: default window callback handler.

See also:

[Fl::set_atclose\(\)](#)

29.2 Events handling functions

Fl class events handling API.

Functions

- static void `Fl::add_handler` (int(*h)(int))
Install a function to parse unrecognized events.
- static void `Fl::belowmouse` (Fl_Widget *)
Sets the widget that is below the mouse.
- static Fl_Widget * `Fl::belowmouse` ()
Gets the widget that is below the mouse.
- static int `Fl::compose` (int &del)
Any text editing widget should call this for each FL_KEYBOARD event.
- static void `Fl::compose_reset` ()
If the user moves the cursor, be sure to call `Fl::compose_reset()`.
- static int `Fl::event` ()
Returns the last event that was processed.
- static int `Fl::event_alt` ()
Returns non-zero if the Alt key is pressed.
- static int `Fl::event_button` ()
Gets which particular mouse button caused the current event.
- static int `Fl::event_button1` ()
Returns non-zero if mouse button 1 is currently held down.
- static int `Fl::event_button2` ()
Returns non-zero if button 2 is currently held down.
- static int `Fl::event_button3` ()
Returns non-zero if button 3 is currently held down.
- static int `Fl::event_buttons` ()
Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.
- static void `Fl::event_clicks` (int i)
Manually sets the number returned by `Fl::event_clicks()`.
- static int `Fl::event_clicks` ()
Returns non zero if we had a double click event.
- static int `Fl::event_ctrl` ()

Returns non-zero if the Control key is pressed.

- static int [Fl::event_dx](#) ()

Returns the current horizontal mouse scrolling associated with the [FL_MOUSEWHEEL](#) event.

- static int [Fl::event_dy](#) ()

Returns the current vertical mouse scrolling associated with the [FL_MOUSEWHEEL](#) event.

- static int [Fl::event_inside](#) (const [Fl_Widget](#) *)

Returns whether or not the mouse event is inside the given widget.

- static int [Fl::event_inside](#) (int, int, int, int)

Returns whether or not the mouse event is inside the given rectangle.

- static void [Fl::event_is_click](#) (int i)

Only i=0 works! See int [event_is_click](#)().

- static int [Fl::event_is_click](#) ()

The first form returns non-zero if the mouse has not moved far enough and not enough time has passed since the last [FL_PUSH](#) or [FL_KEYBOARD](#) event for it to be considered a "drag" rather than a "click".

- static int [Fl::event_key](#) (int key)

Returns true if the given `key` was held down (or pressed) during the last event.

- static int [Fl::event_key](#) ()

Gets which key on the keyboard was last pushed.

- static int [Fl::event_length](#) ()

Returns the length of the text in [Fl::event_text](#)().

- static int [Fl::event_original_key](#) ()

Returns the keycode of the last key event, regardless of the NumLock state.

- static int [Fl::event_shift](#) ()

Returns non-zero if the Shift key is pressed.

- static int [Fl::event_state](#) (int i)

See int [event_state](#)().

- static int [Fl::event_state](#) ()

This is a bitfield of what shift states were on and what mouse buttons were held down during the most recent event.

- static const char * [Fl::event_text](#) ()

Returns the text associated with the current [FL_PASTE](#) or [FL_DND_RELEASE](#) event.

- static int [Fl::event_x](#) ()

Returns the mouse position of the event relative to the [Fl_Window](#) it was passed to.

- static int [Fl::event_x_root](#) ()

Returns the mouse position on the screen of the event.

- static int [Fl::event_y](#) ()

Returns the mouse position of the event relative to the [Fl_Window](#) it was passed to.

- static int [Fl::event_y_root](#) ()

Returns the mouse position on the screen of the event.

- static void [Fl::focus](#) ([Fl_Widget](#) *)

Sets the widget that will receive `FL_KEYBOARD` events.

- static [Fl_Widget](#) * [Fl::focus](#) ()

Gets the current [Fl::focus\(\)](#) widget.

- static int [Fl::get_key](#) (int key)

Returns true if the given `key` is held down now.

- static void [Fl::get_mouse](#) (int &, int &)

Return where the mouse is on the screen by doing a round-trip query to the server.

- static int [Fl::handle](#) (int, [Fl_Window](#) *)

Sends the event to a window for processing.

- static void [Fl::pushed](#) ([Fl_Widget](#) *)

Sets the widget that is being pushed.

- static [Fl_Widget](#) * [Fl::pushed](#) ()

Gets the widget that is being pushed.

- static void [Fl::remove_handler](#) (int(*h)(int))

Removes a previously added event handler.

- static int [Fl::test_shortcut](#) (int)

Test the current event, which must be an `FL_KEYBOARD` or `FL_SHORTCUT`, against a shortcut value (described in [Fl_Button](#)).

29.2.1 Detailed Description

[Fl](#) class events handling API.

29.2.2 Function Documentation

29.2.2.1 void [Fl::add_handler](#) (int(*)(int) *ha*) [static, inherited]

Install a function to parse unrecognized events.

If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non zero then the event is ignored. Events that cause this to be called are:

- FL_SHORTCUT events that are not recognized by any widget. This lets you provide global shortcut keys.
- System events that FLTK does not recognize. See `fl_xevent`.
- *Some* other events when the widget FLTK selected returns zero from its `handle()` method. Exactly which ones may change in future versions, however.

29.2.2.2 `void Fl::belowmouse (Fl_Widget * o)` [static, inherited]

Sets the widget that is below the mouse.

This is for highlighting buttons. It is not used to send FL_PUSH or FL_MOVE directly, for several obscure reasons, but those events typically go to this widget. This is also the first widget tried for FL_SHORTCUT events.

If you change the belowmouse widget, the previous one and all parents (that don't contain the new widget) are sent FL_LEAVE events. Changing this does *not* send FL_ENTER to this or any widget, because sending FL_ENTER is supposed to *test* if the widget wants the mouse (by it returning non-zero from `handle()`).

29.2.2.3 `static Fl_Widget* Fl::belowmouse ()` [inline, static, inherited]

Gets the widget that is below the mouse.

See also:

[belowmouse\(Fl_Widget*\)](#)

29.2.2.4 `int Fl::compose (int & del)` [static, inherited]

Any text editing widget should call this for each FL_KEYBOARD event.

Use of this function is very simple.

If *true* is returned, then it has modified the `Fl::event_text()` and `Fl::event_length()` to a set of *bytes* to insert (it may be of zero length!). It will also set the "del" parameter to the number of *bytes* to the left of the cursor to delete, this is used to delete the results of the previous call to `Fl::compose()`.

If *false* is returned, the keys should be treated as function keys, and del is set to zero. You could insert the text anyways, if you don't know what else to do.

Though the current implementation returns immediately, future versions may take quite awhile, as they may pop up a window or do other user-interface things to allow characters to be selected.

29.2.2.5 `static void Fl::compose_reset ()` [inline, static, inherited]

If the user moves the cursor, be sure to call `Fl::compose_reset()`.

The next call to `Fl::compose()` will start out in an initial state. In particular it will not set "del" to non-zero. This call is very fast so it is ok to call it many times and in many places.

29.2.2.6 static int Fl::event () [inline, static, inherited]

Returns the last event that was processed.

This can be used to determine if a callback is being done in response to a keypress, mouse click, etc.

29.2.2.7 static int Fl::event_alt () [inline, static, inherited]

Returns non-zero if the Alt key is pressed.

29.2.2.8 static int Fl::event_button () [inline, static, inherited]

Gets which particular mouse button caused the current event.

This returns garbage if the most recent event was not a FL_PUSH or FL_RELEASE event.

Return values:

FL_LEFT_MOUSE

FL_MIDDLE_MOUSE

FL_RIGHT_MOUSE.

See also:

[Fl::event_buttons\(\)](#)

29.2.2.9 static int Fl::event_button1 () [inline, static, inherited]

Returns non-zero if mouse button 1 is currently held down.

For more details, see [Fl::event_buttons\(\)](#).

29.2.2.10 static int Fl::event_button2 () [inline, static, inherited]

Returns non-zero if button 2 is currently held down.

For more details, see [Fl::event_buttons\(\)](#).

29.2.2.11 static int Fl::event_button3 () [inline, static, inherited]

Returns non-zero if button 3 is currently held down.

For more details, see [Fl::event_buttons\(\)](#).

29.2.2.12 static int Fl::event_buttons () [inline, static, inherited]

Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.

This function returns the button state at the time of the event. During an FL_RELEASE event, the state of the released button will be 0. To find out, which button caused an FL_RELEASE event, you can use [Fl::event_button\(\)](#) instead.

Returns:

a bit mask value like { [FL_BUTTON1] | [FL_BUTTON2] | [FL_BUTTON3] }

29.2.2.13 static void Fl::event_clicks (int i) [inline, static, inherited]

Manually sets the number returned by [Fl::event_clicks\(\)](#).

This can be used to set it to zero so that later code does not think an item was double-clicked.

Parameters:

← *i* corresponds to no double-click if 0, i+1 mouse clicks otherwise

See also:

int [event_clicks\(\)](#)

29.2.2.14 static int Fl::event_clicks () [inline, static, inherited]

Returns non zero if we had a double click event.

Return values:

Non-zero if the most recent FL_PUSH or FL_KEYBOARD was a "double click".

N-1 for N clicks. A double click is counted if the same button is pressed again while [event_is_click\(\)](#) is true.

29.2.2.15 static int Fl::event_ctrl () [inline, static, inherited]

Returns non-zero if the Control key is pressed.

29.2.2.16 static int Fl::event_dx () [inline, static, inherited]

Returns the current horizontal mouse scrolling associated with the FL_MOUSEWHEEL event.

Right is positive.

29.2.2.17 static int Fl::event_dy () [inline, static, inherited]

Returns the current vertical mouse scrolling associated with the FL_MOUSEWHEEL event.

Down is positive.

29.2.2.18 int Fl::event_inside (const Fl_Widget *o) [static, inherited]

Returns whether or not the mouse event is inside the given widget.

Returns non-zero if the current event_x and event_y put it inside the widget or inside an arbitrary bounding box. You should always call this rather than doing your own comparison so you are consistent about edge effects.

29.2.2.19 `int Fl::event_inside (int xx, int yy, int ww, int hh)` `[static, inherited]`

Returns whether or not the mouse event is inside the given rectangle.

Returns non-zero if the current event_x and event_y put it inside the widget or inside an arbitrary bounding box. You should always call this rather than doing your own comparison so you are consistent about edge effects.

29.2.2.20 `static int Fl::event_is_click ()` `[inline, static, inherited]`

The first form returns non-zero if the mouse has not moved far enough and not enough time has passed since the last FL_PUSH or FL_KEYBOARD event for it to be considered a "drag" rather than a "click".

You can test this on FL_DRAG, FL_RELEASE, and FL_MOVE events. The second form clears the value returned by `Fl::event_is_click()`. Useful to prevent the *next* click from being counted as a double-click or to make a popup menu pick an item with a single click. Don't pass non-zero to this.

29.2.2.21 `int Fl::event_key (int key)` `[static, inherited]`

Returns true if the given key was held down (or pressed) *during* the last event.

This is constant until the next event is read from the server.

`Fl::get_key(int)` returns true if the given key is held down *now*. Under X this requires a round-trip to the server and is *much* slower than `Fl::event_key(int)`.

Keys are identified by the *unshifted* values. FLTK defines a set of symbols that should work on most modern machines for every key on the keyboard:

- All keys on the main keyboard producing a printable ASCII character use the value of that ASCII character (as though shift, ctrl, and caps lock were not on). The space bar is 32.
- All keys on the numeric keypad producing a printable ASCII character use the value of that ASCII character plus FL_KP. The highest possible value is FL_KP_Last so you can range-check to see if something is on the keypad.
- All numbered function keys use the number on the function key plus FL_F. The highest possible number is FL_F_Last, so you can range-check a value.
- Buttons on the mouse are considered keys, and use the button number (where the left button is 1) plus FL_Button.
- All other keys on the keypad have a symbol: FL_Escape, FL_BackSpace, FL_Tab, FL_Enter, FL_Print, FL_Scroll_Lock, FL_Pause, FL_Insert, FL_Home, FL_Page_Up, FL_Delete, FL_End, FL_Page_Down, FL_Left, FL_Up, FL_Right, FL_Down, FL_Shift_L, FL_Shift_R, FL_Control_L, FL_Control_R, FL_Caps_Lock, FL_Alt_L, FL_Alt_R, FL_Meta_L, FL_Meta_R, FL_Menu, FL_-Num_Lock, FL_KP_Enter. Be careful not to confuse these with the very similar, but all-caps, symbols used by `Fl::event_state()`.

On X `Fl::get_key(FL_Button+n)` does not work.

On WIN32 `Fl::get_key(FL_KP_Enter)` and `Fl::event_key(FL_KP_Enter)` do not work.

29.2.2.22 `static int Fl::event_key ()` `[inline, static, inherited]`

Gets which key on the keyboard was last pushed.

Return values:

0 if the last event was not a key press or release.

See also:

int [event_key\(int\)](#)

29.2.2.23 static int Fl::event_length () [inline, static, inherited]

Returns the length of the text in [Fl::event_text\(\)](#).

There will always be a nul at this position in the text. However there may be a nul before that if the keystroke translates to a nul character or you paste a nul character.

29.2.2.24 static int Fl::event_original_key () [inline, static, inherited]

Returns the keycode of the last key event, regardless of the NumLock state.

If NumLock is deactivated, FLTK translates events from the numeric keypad into the corresponding arrow key events. [event_key\(\)](#) returns the translated key code, whereas [event_original_key\(\)](#) returns the keycode before NumLock translation.

29.2.2.25 static int Fl::event_shift () [inline, static, inherited]

Returns non-zero if the Shift key is pressed.

29.2.2.26 static int Fl::event_state () [inline, static, inherited]

This is a bitfield of what shift states were on and what mouse buttons were held down during the most recent event.

The second version returns non-zero if any of the passed bits are turned on. The legal bits are:

- FL_SHIFT
- FL_CAPS_LOCK
- FL_CTRL
- FL_ALT
- FL_NUM_LOCK
- FL_META
- FL_SCROLL_LOCK
- FL_BUTTON1
- FL_BUTTON2
- FL_BUTTON3

X servers do not agree on shift states, and FL_NUM_LOCK, FL_META, and FL_SCROLL_LOCK may not work. The values were selected to match the XFree86 server on Linux. In addition there is a bug in the way X works so that the shift state is not correctly reported until the first event *after* the shift key is pressed or released.

29.2.2.27 static const char* Fl::event_text () [inline, static, inherited]

Returns the text associated with the current FL_PASTE or FL_DND_RELEASE event.

29.2.2.28 static int Fl::event_x_root () [inline, static, inherited]

Returns the mouse position on the screen of the event.

To find the absolute position of an [Fl_Window](#) on the screen, use the difference between [event_x_root\(\)](#), [event_y_root\(\)](#) and [event_x\(\)](#), [event_y\(\)](#).

29.2.2.29 static int Fl::event_y_root () [inline, static, inherited]

Returns the mouse position on the screen of the event.

To find the absolute position of an [Fl_Window](#) on the screen, use the difference between [event_x_root\(\)](#), [event_y_root\(\)](#) and [event_x\(\)](#), [event_y\(\)](#).

29.2.2.30 void Fl::focus (Fl_Widget *o) [static, inherited]

Sets the widget that will receive FL_KEYBOARD events.

If you change [Fl::focus\(\)](#), the previous widget and all parents (that don't contain the new widget) are sent FL_UNFOCUS events. Changing the focus does *not* send FL_FOCUS to this or any widget, because sending FL_FOCUS is supposed to *test* if the widget wants the focus (by it returning non-zero from [handle\(\)](#)).

See also:

[Fl_Widget::take_focus\(\)](#)

29.2.2.31 static Fl_Widget* Fl::focus () [inline, static, inherited]

Gets the current [Fl::focus\(\)](#) widget.

See also:

[Fl::focus\(Fl_Widget*\)](#)

29.2.2.32 int Fl::get_key (int key) [static, inherited]

Returns true if the given *key* is held down *now*.

Under X this requires a round-trip to the server and is *much* slower than [Fl::event_key\(int\)](#).

See also:

[event_key\(int\)](#)

29.2.2.33 `static void Fl::get_mouse (int &, int &) [static, inherited]`

Return where the mouse is on the screen by doing a round-trip query to the server.

You should use [Fl::event_x_root\(\)](#) and [Fl::event_y_root\(\)](#) if possible, but this is necessary if you are not sure if a mouse event has been processed recently (such as to position your first window). If the display is not open, this will open it.

29.2.2.34 `int Fl::handle (int e, Fl_Window * window) [static, inherited]`

Sends the event to a window for processing.

Returns non-zero if any widget uses the event.

29.2.2.35 `void Fl::pushed (Fl_Widget * o) [static, inherited]`

Sets the widget that is being pushed.

FL_DRAG or FL_RELEASE (and any more FL_PUSH) events will be sent to this widget.

If you change the pushed widget, the previous one and all parents (that don't contain the new widget) are sent FL_RELEASE events. Changing this does *not* send FL_PUSH to this or any widget, because sending FL_PUSH is supposed to *test* if the widget wants the mouse (by it returning non-zero from [handle\(\)](#)).

29.2.2.36 `static Fl_Widget* Fl::pushed () [inline, static, inherited]`

Gets the widget that is being pushed.

See also:

`void pushed\(Fl_Widget*\)`

29.2.2.37 `int Fl::test_shortcut (int shortcut) [static, inherited]`

Test the current event, which must be an FL_KEYBOARD or FL_SHORTCUT, against a shortcut value (described in [Fl_Button](#)).

Returns non-zero if there is a match. Not to be confused with [Fl_Widget::test_shortcut\(\)](#).

29.3 Selection & Clipboard functions

fl global copy/cut/paste functions

Functions

- static void **Fl::copy** (const char *stuff, int len, int clipboard=0)
Copies the data pointed to by stuff to the selection (0) or primary (1) clipboard.
- static int **Fl::dnd** ()
Initiate a Drag And Drop operation.
- static void **Fl::paste** (Fl_Widget &receiver)
Backward compatibility only: Set things up so the receiver widget will be called with an FL_PASTE event some time in the future for the specified clipboard.
- static void **Fl::paste** (Fl_Widget &receiver, int clipboard)
Pastes the data from the selection (0) or primary (1) clipboard into receiver.
- static void **Fl::selection** (Fl_Widget &owner, const char *, int len)
Changes the current selection.
- static void **Fl::selection_owner** (Fl_Widget *)
Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.
- static Fl_Widget * **Fl::selection_owner** ()
back-compatibility only: Gets the widget owning the current selection

29.3.1 Detailed Description

fl global copy/cut/paste functions

29.3.2 Function Documentation

29.3.2.1 static void Fl::copy (const char * stuff, int len, int clipboard = 0) [static, inherited]

Copies the data pointed to by stuff to the selection (0) or primary (1) clipboard.

The selection clipboard is used for middle-mouse pastes and for drag-and-drop selections. The primary clipboard is used for traditional copy/cut/paste operations.

29.3.2.2 int Fl::dnd () [static, inherited]

Initiate a Drag And Drop operation.

Drag and drop whatever is in the cut-copy-paste buffer.

drag and drop whatever is in the cut-copy-paste buffer

- create a selection first using: `Fl::copy(const char *stuff, int len, 0)`

The clipboard should be filled with relevant data before calling this method. FLTK will then initiate the system wide drag and drop handling. Dropped data will be marked as *text*.

Create a selection first using:

`Fl::copy(const char *stuff, int len, 0)`

29.3.2.3 void Fl::paste (Fl_Widget &receiver) [static, inherited]

Backward compatibility only: Set things up so the receiver widget will be called with an FL_PASTE event some time in the future for the specified clipboard.

The receiver should be prepared to be called *directly* by this, or for it to happen *later*, or possibly *not at all*. This allows the window system to take as long as necessary to retrieve the paste buffer (or even to screw up completely) without complex and error-prone synchronization code in FLTK.

See also:

[Fl::paste\(Fl_Widget &receiver, int clipboard\)](#)

29.3.2.4 static void Fl::paste (Fl_Widget &receiver, int clipboard) [static, inherited]

Pastes the data from the selection (0) or primary (1) clipboard into receiver.

The selection clipboard is used for middle-mouse pastes and for drag-and-drop selections. The primary clipboard is used for traditional copy/cut/paste operations.

29.3.2.5 void Fl::selection (Fl_Widget &owner, const char *text, int len) [static, inherited]

Changes the current selection.

The block of text is copied to an internal buffer by FLTK (be careful if doing this in response to an FL_PASTE as this *may* be the same buffer returned by [event_text\(\)](#)). The [selection_owner\(\)](#) widget is set to the passed owner.

29.3.2.6 void Fl::selection_owner (Fl_Widget *owner) [static, inherited]

Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.

FL_SELECTIONCLEAR is sent to the previous selection owner, if any.

Copying the buffer every time the selection is changed is obviously wasteful, especially for large selections. An interface will probably be added in a future version to allow the selection to be made by a callback function. The current interface will be emulated on top of this.

29.3.2.7 static Fl_Widget* Fl::selection_owner () [inline, static, inherited]

back-compatibility only: Gets the widget owning the current selection

See also:

Fl_Widget* [selection_owner\(Fl_Widget*\)](#)

29.4 Screen functions

fl global screen functions

Functions

- static int `Fl::h ()`
Returns the height of the screen in pixels.
- static int `Fl::screen_count ()`
Gets the number of available screens.
- static void `Fl::screen_xywh (int &X, int &Y, int &W, int &H, int n)`
Gets the screen bounding rect for the given screen.
- static void `Fl::screen_xywh (int &X, int &Y, int &W, int &H, int mx, int my)`
Gets the bounding box of a screen that contains the specified screen position `mx`, `my`.
- static void `Fl::screen_xywh (int &X, int &Y, int &W, int &H)`
Gets the bounding box of a screen that contains the mouse pointer.
- static int `Fl::w ()`
Returns the width of the screen in pixels.
- static int `Fl::x ()`
Returns the origin of the current screen, where 0 indicates the left side of the screen.
- static int `Fl::y ()`
Returns the origin of the current screen, where 0 indicates the top edge of the screen.

29.4.1 Detailed Description

fl global screen functions

29.4.2 Function Documentation

29.4.2.1 static int `Fl::h ()` [static, inherited]

Returns the height of the screen in pixels.

29.4.2.2 void `Fl::screen_xywh (int & X, int & Y, int & W, int & H, int n)` [static, inherited]

Gets the screen bounding rect for the given screen.

Parameters:

→ *X,Y,W,H* the corresponding screen bounding box

← *n* the screen number (0 to `Fl::screen_count()` - 1)

See also:

void `screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)`

29.4.2.3 void `Fl::screen_xywh (int & X, int & Y, int & W, int & H, int mx, int my)` [`static`, `inherited`]

Gets the bounding box of a screen that contains the specified screen position `mx`, `my`.

Parameters:

→ *X,Y,W,H* the corresponding screen bounding box

← *mx,my* the absolute screen position

29.4.2.4 static void `Fl::screen_xywh (int & X, int & Y, int & W, int & H)` [`inline`, `static`, `inherited`]

Gets the bounding box of a screen that contains the mouse pointer.

Parameters:

→ *X,Y,W,H* the corresponding screen bounding box

See also:

void `screen_xywh(int &x, int &y, int &w, int &h, int mx, int my)`

29.4.2.5 static int `Fl::w ()` [`static`, `inherited`]

Returns the width of the screen in pixels.

29.4.2.6 static int `Fl::x ()` [`static`, `inherited`]

Returns the origin of the current screen, where 0 indicates the left side of the screen.

29.4.2.7 static int `Fl::y ()` [`static`, `inherited`]

Returns the origin of the current screen, where 0 indicates the top edge of the screen.

29.5 Color & Font functions

fl global color, font functions

Functions

- `Fl_Color fl_color ()`
Returns the last `fl_color()` that was set.
- `FL_EXPORT void fl_color (uchar r, uchar g, uchar b)`
Set the color for all subsequent drawing operations.
- `void fl_color (int c)`
for back compatibility - use `fl_color(Fl_Color c)` instead
- `FL_EXPORT void fl_color (Fl_Color i)`
Sets the color for all subsequent drawing operations.
- `Fl_Color fl_color_average (Fl_Color color1, Fl_Color color2, float weight)`
Returns the weighted average color between the two given colors.
- `Fl_Color fl_contrast (Fl_Color fg, Fl_Color bg)`
Returns a color that contrasts with the background color.
- `FL_EXPORT int fl_descent ()`
Returns the recommended distance above the bottom of a `fl_height()` tall box to draw the text at so it looks centered vertically in that box.
- `Fl_Font fl_font ()`
Returns the `face` set by the most recent call to `fl_font()`.
- `FL_EXPORT void fl_font (Fl_Font face, Fl_Fontsize size)`
Sets the current font, which is then used in various drawing routines.
- `int fl_height (int, int size)`
Dummy passthru function called only in `Fl_Text_Display` that simply returns the font height as given by the `size` parameter in the same call!
- `FL_EXPORT int fl_height ()`
Returns the recommended minimum line spacing for the current font.
- `Fl_Color fl_inactive (Fl_Color c)`
Returns the inactive, dimmed version of the given color.
- `FL_EXPORT const char * fl_latin1_to_local (const char *t, int n=-1)`
convert text from Windows/X11 latin1 character set to local encoding.
- `FL_EXPORT const char * fl_local_to_latin1 (const char *t, int n=-1)`
convert text from local encoding to Windows/X11 latin1 character set.

- FL_EXPORT const char * [fl_local_to_mac_roman](#) (const char *t, int n=-1)
convert text from local encoding to Mac Roman character set.
- FL_EXPORT const char * [fl_mac_roman_to_local](#) (const char *t, int n=-1)
convert text from Mac Roman character set to local encoding.
- [Fl_Fontsize fl_size](#) ()
Returns the size set by the most recent call to [fl_font\(\)](#).
- FL_EXPORT void [fl_text_extents](#) (const char *, int n, int &dx, int &dy, int &w, int &h)
Determine the minimum pixel dimensions of a sequence of n characters.
- FL_EXPORT void [fl_text_extents](#) (const char *, int &dx, int &dy, int &w, int &h)
Determine the minimum pixel dimensions of a nul-terminated string.
- FL_EXPORT double [fl_width](#) (Fl_Unichar)
Return the typographical width of a single character .:
- FL_EXPORT double [fl_width](#) (const char *txt, int n)
Return the typographical width of a sequence of n characters.
- FL_EXPORT double [fl_width](#) (const char *txt)
Return the typographical width of a nul-terminated string.
- [ulong fl_xpixel](#) (Fl_Color i)
Returns the X pixel number used to draw the given FLTK color index.
- [ulong fl_xpixel](#) (uchar r, uchar g, uchar b)
Returns the X pixel number used to draw the given rgb color.
- static void [Fl::free_color](#) (Fl_Color, int overlay=0)
Frees the specified color from the colormap, if applicable.
- static void [Fl::get_color](#) (Fl_Color, uchar &, uchar &, uchar &)
See unsigned [get_color\(Fl_Color c\)](#).
- static unsigned [Fl::get_color](#) (Fl_Color)
Returns the RGB value(s) for the given FLTK color index.
- static const char * [Fl::get_font](#) (Fl_Font)
Gets the string for this face.
- static const char * [Fl::get_font_name](#) (Fl_Font, int *attributes=0)
Get a human-readable string describing the family of this face.
- static int [Fl::get_font_sizes](#) (Fl_Font, int *&sizep)
Return an array of sizes in sizep.
- static void [Fl::set_color](#) (Fl_Color, unsigned)
Sets an entry in the fl_color index table.

- static void `Fl::set_color (Fl_Color, uchar, uchar, uchar)`
Sets an entry in the `fl_color` index table.
- static void `Fl::set_font (Fl_Font, Fl_Font)`
Copies one face to another.
- static void `Fl::set_font (Fl_Font, const char *)`
Changes a face.
- static `Fl_Font Fl::set_fonts (const char *[]=0)`
FLTK will open the display, and add every fonts on the server to the face table.

Variables

- `Fl_Color fl_color_`
Current color for drawing operations.
- `FL_EXPORT Fl_Color fl_color_`
Current color for drawing operations.
- `FL_EXPORT Fl_Font fl_font_`
current font index
- `FL_EXPORT Fl_Fonsize fl_size_`
current font size

29.5.1 Detailed Description

fl global color, font functions

29.5.2 Function Documentation

29.5.2.1 `Fl_Color fl_color ()` `[inline]`

Returns the last `fl_color()` that was set.

This can be used for state save/restore.

29.5.2.2 `void fl_color (uchar r, uchar g, uchar b)`

Set the color for all subsequent drawing operations.

The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used. If no valid graphical context (`fl_gc`) is available, the foreground is not set for the current window.

Parameters:

← *r,g,b* color components

29.5.2.3 void fl_color (FL_Color *i*)

Sets the color for all subsequent drawing operations.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color. If no valid graphical context (`fl_gc`) is available, the foreground is not set for the current window.

Parameters:

← *i* color

29.5.2.4 FL_Color fl_color_average (FL_Color *color1*, FL_Color *color2*, float *weight*)

Returns the weighted average color between the two given colors.

The red, green and blue values are averages using the following formula:

```
color = color1 * weight + color2 * (1 - weight)
```

Thus, a `weight` value of 1.0 will return the first color, while a value of 0.0 will return the second color.

Parameters:

← *color1,color2* boundary colors

← *weight* weighting factor

29.5.2.5 FL_Color fl_contrast (FL_Color *fg*, FL_Color *bg*)

Returns a color that contrasts with the background color.

This will be the foreground color if it contrasts sufficiently with the background color. Otherwise, returns `FL_WHITE` or `FL_BLACK` depending on which color provides the best contrast.

Parameters:

← *fg,bg* foreground and background colors

Returns:

contrasting color

29.5.2.6 FL_Font fl_font () [inline]

Returns the `face` set by the most recent call to `fl_font()`.

This can be used to save/restore the font.

29.5.2.7 FL_EXPORT void fl_font (Fl_Font *face*, Fl_Fontsize *size*)

Sets the current font, which is then used in various drawing routines.

You may call this outside a draw context if necessary to call [fl_width\(\)](#), but on X this will open the display.

The font is identified by a *face* and a *size*. The size of the font is measured in pixels and not "points". Lines should be spaced *size* pixels apart or more.

29.5.2.8 int fl_height (int, int *size*) [inline]

Dummy passthru function called only in [Fl_Text_Display](#) that simply returns the font height as given by the *size* parameter in the same call!

Todo

Is [fl_height\(int, int size\)](#) required for [Fl_Text_Display](#)? Why not use *size* parameter directly?

29.5.2.9 FL_EXPORT int fl_height ()

Returns the recommended minimum line spacing for the current font.

You can also use the value of *size* passed to [fl_font\(\)](#)

29.5.2.10 FL_EXPORT const char* fl_latin1_to_local (const char * *t*, int *n* = -1)

convert text from Windows/X11 latin1 character set to local encoding.

Parameters:

- ← *t* character string (latin1 encoding)
- ← *n* optional number of characters to convert (default is all)

Returns:

pointer to internal buffer containing converted characters

29.5.2.11 FL_EXPORT const char* fl_local_to_latin1 (const char * *t*, int *n* = -1)

convert text from local encoding to Windows/X11 latin1 character set.

Parameters:

- ← *t* character string (local encoding)
- ← *n* optional number of characters to convert (default is all)

Returns:

pointer to internal buffer containing converted characters

29.5.2.12 FL_EXPORT const char* fl_local_to_mac_roman (const char * *t*, int *n* = -1)

convert text from local encoding to Mac Roman character set.

Parameters:

- ← *t* character string (local encoding)
- ← *n* optional number of characters to convert (default is all)

Returns:

pointer to internal buffer containing converted characters

29.5.2.13 FL_EXPORT const char* fl_mac_roman_to_local (const char * *t*, int *n* = -1)

convert text from Mac Roman character set to local encoding.

Parameters:

- ← *t* character string (Mac Roman encoding)
- ← *n* optional number of characters to convert (default is all)

Returns:

pointer to internal buffer containing converted characters

29.5.2.14 FL_Fontsize fl_size () [inline]

Returns the `size` set by the most recent call to `fl_font()`.

This can be used to save/restore the font.

29.5.2.15 FL_EXPORT void fl_text_extents (const char *, int *n*, int & *dx*, int & *dy*, int & *w*, int & *h*)

Determine the minimum pixel dimensions of a sequence of *n* characters.

See also:

[fl_text_extents\(const char*, int& dx, int& dy, int& w, int& h\)](#)

29.5.2.16 FL_EXPORT void fl_text_extents (const char *, int & *dx*, int & *dy*, int & *w*, int & *h*)

Determine the minimum pixel dimensions of a nul-terminated string.

Usage: given a string "txt" drawn using `fl_draw(txt, x, y)` you would determine its pixel extents on the display using `fl_text_extents(txt, dx, dy, wo, ho)` such that a bounding box that exactly fits around the text could be drawn with `fl_rect(x+dx, y+dy, wo, ho)`. Note the *dx*, *dy* values hold the offset of the first "colored in" pixel of the string, from the draw origin.

29.5.2.17 FL_EXPORT double fl_width (Fl_Unichar)

Return the typographical width of a single character :.

Note:

if a valid `fl_gc` is NOT found then it uses the first window `gc`, or the screen `gc` if no fltk window is available when called.

29.5.2.18 ulong fl_xpixel (Fl_Color *i*)

Returns the X pixel number used to draw the given FLTK color index.

This is the X pixel that `fl_color()` would use.

Parameters:

← *i* color index

Returns:

X pixel number

29.5.2.19 ulong fl_xpixel (uchar *r*, uchar *g*, uchar *b*)

Returns the X pixel number used to draw the given rgb color.

This is the X pixel that `fl_color()` would use.

Parameters:

← *r,g,b* color components

Returns:

X pixel number

29.5.2.20 void Fl::free_color (Fl_Color *i*, int *overlay* = 0) [static, inherited]

Frees the specified color from the colormap, if applicable.

Free color *i* if used, and clear mapping table entry.

If overlay is non-zero then the color is freed from the overlay colormap.

Parameters:

← *i* color index

← *overlay* 0 for normal, 1 for overlay color

29.5.2.21 unsigned Fl::get_color (Fl_Color *i*) [static, inherited]

Returns the RGB value(s) for the given FLTK color index.

The first form returns the RGB values packed in a 32-bit unsigned integer with the red value in the upper 8 bits, the green value in the next 8 bits, and the blue value in bits 8-15. The lower 8 bits will always be 0.

The second form returns the red, green, and blue values separately in referenced variables.

29.5.2.22 const char * Fl::get_font (Fl_Font *fnum*) [static, inherited]

Gets the string for this face.

This string is different for each face. Under X this value is passed to XListFonts to get all the sizes of this face.

29.5.2.23 const char * Fl::get_font_name (Fl_Font *fnum*, int * *attributes* = 0) [static, inherited]

Get a human-readable string describing the family of this face.

This is useful if you are presenting a choice to the user. There is no guarantee that each face has a different name. The return value points to a static buffer that is overwritten each call.

The integer pointed to by *attributes* (if the pointer is not zero) is set to zero, FL_BOLD or FL_ITALIC or FL_BOLD | FL_ITALIC. To locate a "family" of fonts, search forward and back for a set with non-zero attributes, these faces along with the face with a zero attribute before them constitute a family.

29.5.2.24 int Fl::get_font_sizes (Fl_Font *fnum*, int *& *sizep*) [static, inherited]

Return an array of sizes in *sizep*.

The return value is the length of this array. The sizes are sorted from smallest to largest and indicate what sizes can be given to `fl_font()` that will be matched exactly (`fl_font()` will pick the closest size for other sizes). A zero in the first location of the array indicates a scalable font, where any size works, although the array may list sizes that work "better" than others. Warning: the returned array points at a static buffer that is overwritten each call. Under X this will open the display.

29.5.2.25 void Fl::set_color (Fl_Color *i*, unsigned *c*) [static, inherited]

Sets an entry in the `fl_color` index table.

Set color mapping table entry *i* to color *c*.

You can set it to any 8-bit RGB color. The color is not allocated until `fl_color(i)` is used.

Parameters:

← *i* color index

← *c* color

29.5.2.26 void Fl::set_color (Fl_Color *i*, uchar *red*, uchar *green*, uchar *blue*) [static, inherited]

Sets an entry in the `fl_color` index table.

You can set it to any 8-bit RGB color. The color is not allocated until `fl_color(i)` is used.

29.5.2.27 `void Fl::set_font (Fl_Font fnum, Fl_Font from)` `[static, inherited]`

Copies one face to another.

29.5.2.28 `void Fl::set_font (Fl_Font fnum, const char * name)` `[static, inherited]`

Changes a face.

The string pointer is simply stored, the string is not copied, so the string must be in static memory.

29.5.2.29 `Fl_Font Fl::set_fonts (const char * xstarname = 0)` `[static, inherited]`

FLTK will open the display, and add every fonts on the server to the face table.

It will attempt to put "families" of faces together, so that the normal one is first, followed by bold, italic, and bold italic.

The optional argument is a string to describe the set of fonts to add. Passing NULL will select only fonts that have the ISO8859-1 character set (and are thus usable by normal text). Passing "-*" will select all fonts with any encoding as long as they have normal X font names with dashes in them. Passing "*" will list every font that exists (on X this may produce some strange output). Other values may be useful but are system dependent. With WIN32 NULL selects fonts with ISO8859-1 encoding and non-NULL selects all fonts.

The return value is how many faces are in the table after this is done.

29.6 Drawing functions

fl global graphics and gui drawing functions

Defines

- #define `fl_clip` `fl_push_clip`
The `fl_clip()` name is deprecated and will be removed from future releases.

Typedefs

- typedef void(* `Fl_Draw_Image_Cb`)(void *data, int x, int y, int w, `uchar` *buf)
signature of image generation callback function.

Enumerations

- enum {
 `FL_SOLID` = 0, `FL_DASH` = 1, `FL_DOT` = 2, `FL_DASHDOT` = 3,
 `FL_DASHDOTDOT` = 4, `FL_CAP_FLAT` = 0x100, `FL_CAP_ROUND` = 0x200, `FL_CAP_SQUARE` = 0x300,
 `FL_JOIN_MITER` = 0x1000, `FL_JOIN_ROUND` = 0x2000, `FL_JOIN_BEVEL` = 0x3000 }

Functions

- `FL_EXPORT int fl_add_symbol` (const char *name, void(*drawit)(`Fl_Color`), int scalable)
Adds a symbol to the system.
- `FL_EXPORT void fl_arc` (double x, double y, double r, double start, double a)
Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using `scale` and `rotate` before calling `fl_arc()`.
- `FL_EXPORT void fl_arc` (int x, int y, int w, int h, double a1, double a2)
Draw ellipse sections using integer coordinates.
- `FL_EXPORT void fl_begin_complex_polygon` ()
Starts drawing a complex filled polygon.
- `FL_EXPORT void fl_begin_line` ()
Starts drawing a list of lines.
- `FL_EXPORT void fl_begin_loop` ()
Starts drawing a closed sequence of lines.
- `FL_EXPORT void fl_begin_points` ()
Starts drawing a list of points.

- FL_EXPORT void [fl_begin_polygon](#) ()
Starts drawing a convex filled polygon.
- FL_EXPORT char [fl_can_do_alpha_blending](#) ()
Checks whether platform supports true alpha blending for RGBA images.
- FL_EXPORT void [fl_chord](#) (int x, int y, int w, int h, double a1, double a2)
fl_chord declaration is a place holder - the function does not yet exist
- FL_EXPORT void [fl_circle](#) (double x, double y, double r)
fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- FL_EXPORT int [fl_clip_box](#) (int, int, int, int, int &x, int &y, int &w, int &h)
Intersects the rectangle with the current clip region and returns the bounding box of the result.
- FL_EXPORT void [fl_cursor](#) (FL_Cursor, [FL_Color](#) fg=FL_BLACK, [FL_Color](#) bg=FL_WHITE)
Sets the cursor for the current window to the specified shape and colors.
- FL_EXPORT void [fl_curve](#) (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)
Add a series of points on a Bezier curve to the path.
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y, int w, int h, [FL_Align](#) align, void(*callthis)(const char *, int, int, int), [FL_Image](#) *img=0, int draw_symbols=1)
The same as fl_draw(const char,int,int,int,int,FL_Align,FL_Image*,int) with the addition of the callthis parameter, which is a pointer to a text drawing function such as fl_draw(const char*, int, int, int) to do the real work.*
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y, int w, int h, [FL_Align](#) align, [FL_Image](#) *img=0, int draw_symbols=1)
Fancy string drawing function which is used to draw all the labels.
- FL_EXPORT void [fl_draw](#) (const char *str, int n, int x, int y)
Draws an array of n characters starting at the given location.
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y)
Draws a nul-terminated string starting at the given location.
- FL_EXPORT void [fl_draw_box](#) ([FL_Boxtype](#), int x, int y, int w, int h, [FL_Color](#))
Draws a box using given type, position, size and color.
- FL_EXPORT void [fl_draw_image](#) ([FL_Draw_Image_Cb](#) cb, void *data, int X, int Y, int W, int H, int D=3)
Draw image using callback function to generate image data.
- FL_EXPORT void [fl_draw_image](#) (const [uchar](#) *buf, int X, int Y, int W, int H, int D=3, int L=0)
Draw an 8-bit per color RGB or luminance image.
- FL_EXPORT void [fl_draw_image_mono](#) ([FL_Draw_Image_Cb](#) cb, void *data, int X, int Y, int W, int H, int D=1)

Draw gray-scale image using callback function to generate image data.

- FL_EXPORT void [fl_draw_image_mono](#) (const [uchar](#) *buf, int X, int Y, int W, int H, int D=1, int L=0)

Draw a gray-scale (1 channel) image.

- FL_EXPORT int [fl_draw_pixmap](#) (const char *const *cdata, int x, int y, [FL_Color](#)=FL_GRAY)

Draw XPM image data, with the top-left corner at the given position.

- FL_EXPORT int [fl_draw_pixmap](#) (char *const *data, int x, int y, [FL_Color](#)=FL_GRAY)

Draw XPM image data, with the top-left corner at the given position.

- FL_EXPORT int [fl_draw_symbol](#) (const char *label, int x, int y, int w, int h, [FL_Color](#))

Draw the named symbol in the given rectangle using the given color.

- FL_EXPORT void [fl_end_complex_polygon](#) ()

Ends complex filled polygon, and draws.

- FL_EXPORT void [fl_end_line](#) ()

Ends list of lines, and draws.

- FL_EXPORT void [fl_end_loop](#) ()

Ends closed sequence of lines, and draws.

- FL_EXPORT void [fl_end_points](#) ()

Ends list of points, and draws.

- FL_EXPORT void [fl_end_polygon](#) ()

Ends convex filled polygon, and draws.

- FL_EXPORT const char * [fl_expand_text](#) (const char *from, char *buf, int maxbuf, double maxw, int &n, double &width, int wrap, int draw_symbols=0)

Copy from to buf, replacing unprintable characters with ^X and \nnn.

- FL_EXPORT void [fl_frame](#) (const char *s, int x, int y, int w, int h)

Draws a series of line segments around the given box.

- FL_EXPORT void [fl_frame2](#) (const char *s, int x, int y, int w, int h)

Draws a series of line segments around the given box.

- FL_EXPORT void [fl_gap](#) ()

Call [fl_gap\(\)](#) to separate loops of the path.

- FL_EXPORT void [fl_line](#) (int x, int y, int x1, int y1, int x2, int y2)

Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2).

- FL_EXPORT void [fl_line](#) (int x, int y, int x1, int y1)

Draws a line from (x,y) to (x1,y1).

- FL_EXPORT void [fl_line_style](#) (int style, int width=0, char *dashes=0)

Sets how to draw lines (the "pen").

- FL_EXPORT void [fl_loop](#) (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
Outlines a 4-sided polygon with lines.
- FL_EXPORT void [fl_loop](#) (int x, int y, int x1, int y1, int x2, int y2)
Outlines a 3-sided polygon with lines.
- FL_EXPORT void [fl_measure](#) (const char *str, int &x, int &y, int draw_symbols=1)
Measure how wide and tall the string will be when printed by the [fl_draw\(\)](#) function with align parameter.
- FL_EXPORT int [fl_measure_pixmap](#) (const char *const *cdata, int &w, int &h)
Get the dimensions of a pixmap.
- FL_EXPORT int [fl_measure_pixmap](#) (char *const *data, int &w, int &h)
Get the dimensions of a pixmap.
- FL_EXPORT void [fl_mult_matrix](#) (double a, double b, double c, double d, double x, double y)
Concatenates another transformation onto the current one.
- FL_EXPORT int [fl_not_clipped](#) (int x, int y, int w, int h)
Does the rectangle intersect the current clip region?
- FL_EXPORT void [fl_overlay_clear](#) ()
Erase a selection rectangle without drawing a new one.
- FL_EXPORT void [fl_overlay_rect](#) (int x, int y, int w, int h)
Draws a selection rectangle, erasing a previous one by XOR'ing it first.
- FL_EXPORT void [fl_pie](#) (int x, int y, int w, int h, double a1, double a2)
Draw filled ellipse sections using integer coordinates.
- FL_EXPORT void [fl_point](#) (int x, int y)
Draws a single pixel at the given coordinates.
- FL_EXPORT void [fl_polygon](#) (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
Fills a 4-sided polygon.
- FL_EXPORT void [fl_polygon](#) (int x, int y, int x1, int y1, int x2, int y2)
Fills a 3-sided polygon.
- FL_EXPORT void [fl_pop_clip](#) ()
Restores the previous clip region.
- FL_EXPORT void [fl_pop_matrix](#) ()
Restores the current transformation matrix from the stack.
- FL_EXPORT void [fl_push_clip](#) (int x, int y, int w, int h)
Intersects the current clip region with a rectangle and pushes this new region onto the stack.

- FL_EXPORT void [fl_push_matrix](#) ()
Saves the current transformation matrix on the stack.
- FL_EXPORT void [fl_push_no_clip](#) ()
Pushes an empty clip region onto the stack so nothing will be clipped.
- FL_EXPORT [uchar](#) * [fl_read_image](#) ([uchar](#) *p, int X, int Y, int W, int H, int alpha=0)
Read an RGB(A) image from the current window or off-screen buffer.
- void [fl_rect](#) (int x, int y, int w, int h, [Fl_Color](#) c)
Draws a 1-pixel border inside the given bounding box.
- FL_EXPORT void [fl_rect](#) (int x, int y, int w, int h)
Draws a 1-pixel border inside the given bounding box.
- FL_EXPORT void [fl_rectf](#) (int x, int y, int w, int h, [uchar](#) r, [uchar](#) g, [uchar](#) b)
Color a rectangle with "exactly" the passed r, g, b color.
- void [fl_rectf](#) (int x, int y, int w, int h, [Fl_Color](#) c)
Colors a rectangle that exactly fills the given bounding box.
- FL_EXPORT void [fl_rectf](#) (int x, int y, int w, int h)
Colors a rectangle that exactly fills the given bounding box.
- FL_EXPORT void [fl_reset_spot](#) (void)
- FL_EXPORT void [fl_rotate](#) (double d)
Concatenates rotation transformation onto the current one.
- FL_EXPORT void [fl_rtl_draw](#) (const char *, int n, int x, int y)
Draws an array of n characters right to left starting at given location.
- FL_EXPORT void [fl_scale](#) (double x)
Concatenates scaling transformation onto the current one.
- FL_EXPORT void [fl_scale](#) (double x, double y)
Concatenates scaling transformation onto the current one.
- FL_EXPORT void [fl_scroll](#) (int X, int Y, int W, int H, int dx, int dy, void(*draw_area)(void *, int, int, int), void *data)
Scroll a rectangle and draw the newly exposed portions.
- FL_EXPORT void [fl_set_spot](#) (int font, int size, int X, int Y, int W, int H, [Fl_Window](#) *win=0)
- FL_EXPORT void [fl_set_status](#) (int X, int Y, int W, int H)
- FL_EXPORT const char * [fl_shortcut_label](#) (int shortcut)
Get a human-readable string from a shortcut value.
- FL_EXPORT double [fl_transform_dx](#) (double x, double y)
Transforms distance using current transformation matrix.
- FL_EXPORT double [fl_transform_dy](#) (double x, double y)

Transforms distance using current transformation matrix.

- FL_EXPORT double [fl_transform_x](#) (double x, double y)
Transforms coordinate using the current transformation matrix.
- FL_EXPORT double [fl_transform_y](#) (double x, double y)
Transform coordinate using the current transformation matrix.
- FL_EXPORT void [fl_transformed_vertex](#) (double x, double y)
Adds coordinate pair to the vertex list without further transformations.
- FL_EXPORT void [fl_translate](#) (double x, double y)
Concatenates translation transformation onto the current one.
- FL_EXPORT void [fl_vertex](#) (double x, double y)
Adds a single vertex to the current path.
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1, int y2, int x3)
Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2).
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1, int y2)
Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2).
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1)
Draws a horizontal line from (x,y) to (x1,y).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1, int x2, int y3)
Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1, int x2)
Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1)
Draws a vertical line from (x,y) to (x,y1).

29.6.1 Detailed Description

fl global graphics and gui drawing functions

29.6.2 Typedef Documentation

29.6.2.1 typedef void(* FL_Draw_Image_Cb)(void *data, int x, int y, int w, uchar *buf)

signature of image generation callback function.

Parameters:

← **data** user data passed to function

- ← *x,y,w* position and width of scan line in image
- *buf* buffer for generated image data. You must copy *w* pixels from scanline *y*, starting at pixel *x* to this buffer.

29.6.3 Enumeration Type Documentation

29.6.3.1 anonymous enum

Enumerator:

FL_SOLID line style: _____
FL_DASH line style: _ _ _ _ _
FL_DOT line style:
FL_DASHDOT line style: _ . _ . _ .
FL_DASHDOTDOT line style: _ . . _ . .
FL_CAP_FLAT cap style: end is flat
FL_CAP_ROUND cap style: end is round
FL_CAP_SQUARE cap style: end wraps end point
FL_JOIN_MITER join style: line join extends to a point
FL_JOIN_ROUND join style: line join is rounded
FL_JOIN_BEVEL join style: line join is tidied

29.6.4 Function Documentation

29.6.4.1 **FL_EXPORT int fl_add_symbol (const char * *name*, void(*)(*FL_Color*) *drawit*, int *scalable*)**

Adds a symbol to the system.

Parameters:

- ← *name* name of symbol (without the "@")
- ← *drawit* function to draw symbol
- ← *scalable* set to 1 if *drawit* uses scalable vector drawing

Returns:

1 on success, 0 on failure

29.6.4.2 **FL_EXPORT void fl_arc (double *x*, double *y*, double *r*, double *start*, double *end*)**

Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using *scale* and rotate before calling [fl_arc\(\)](#).

Parameters:

- ← *x,y,r* center and radius of circular arc
- ← *start,end* angles of start and end of arc measured in degrees counter-clockwise from 3 o'clock. If *end* is less than *start* then it draws the arc in a clockwise direction.

29.6.4.3 FL_EXPORT void fl_arc (int x, int y, int w, int h, double a1, double a2)

Draw ellipse sections using integer coordinates.

These functions match the rather limited circle drawing code provided by X and WIN32. The advantage over using fl_arc with floating point coordinates is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counterclockwise from 3 o'clock and are the starting and ending angle of the arc, a2 must be greater or equal to a1.

fl_arc() draws a series of lines to approximate the arc. Notice that the integer version of fl_arc() has a different number of arguments than the double version fl_arc(double x, double y, double r, double start, double a)

Parameters:

← **x,y,w,h** bounding box of complete circle

← **a1,a2** start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. a2 must be greater than or equal to a1.

29.6.4.4 FL_EXPORT void fl_begin_complex_polygon ()

Starts drawing a complex filled polygon.

The polygon may be concave, may have holes in it, or may be several disconnected pieces. Call fl_gap() to separate loops of the path.

To outline the polygon, use fl_begin_loop() and replace each fl_gap() with fl_end_loop();fl_begin_loop() pairs.

Note:

For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. Holes should be drawn in the opposite direction to the outside loop.

29.6.4.5 FL_EXPORT void fl_begin_points ()

Starts drawing a list of points.

Points are added to the list with fl_vertex()

29.6.4.6 FL_EXPORT char fl_can_do_alpha_blending ()

Checks whether platform supports true alpha blending for RGBA images.

Returns:

- 1 if true alpha blending supported by platform
- 0 not supported so FLTK will use screen door transparency

29.6.4.7 FL_EXPORT void fl_circle (double *x*, double *y*, double *r*)

[fl_circle\(\)](#) is equivalent to [fl_arc\(x,y,r,0,360\)](#), but may be faster.

It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use [fl_arc\(\)](#)

Parameters:

← *x,y,r* center and radius of circle

29.6.4.8 FL_EXPORT int fl_clip_box (int *x*, int *y*, int *w*, int *h*, int & *X*, int & *Y*, int & *W*, int & *H*)

Intersects the rectangle with the current clip region and returns the bounding box of the result.

Returns non-zero if the resulting rectangle is different to the original. This can be used to limit the necessary drawing to a rectangle. *W* and *H* are set to zero if the rectangle is completely outside the region.

Parameters:

← *x,y,w,h* position and size of rectangle

→ *X,Y,W,H* position and size of resulting bounding box. *W* and *H* are set to zero if the rectangle is completely outside the region.

Returns:

Non-zero if the resulting rectangle is different to the original.

29.6.4.9 FL_EXPORT void fl_cursor (FL_Cursor *c*, FL_Color *fg*, FL_Color *bg*)

Sets the cursor for the current window to the specified shape and colors.

The cursors are defined in the <[FL/Enumerations.H](#)> header file.

29.6.4.10 FL_EXPORT void fl_curve (double *X0*, double *Y0*, double *X1*, double *Y1*, double *X2*, double *Y2*, double *X3*, double *Y3*)

Add a series of points on a Bezier curve to the path.

The curve ends (and two of the points) are at *X0,Y0* and *X3,Y3*.

Parameters:

← *X0,Y0* curve start point

← *X1,Y1* curve control point

← *X2,Y2* curve control point

← *X3,Y3* curve end point

29.6.4.11 **FL_EXPORT void fl_draw (const char * *str*, int *x*, int *y*, int *w*, int *h*, FL_Align *align*, FL_Image * *img*, int *draw_symbols*)**

Fancy string drawing function which is used to draw all the labels.

The string is formatted and aligned inside the passed box. Handles '\t' and '\n', expands all other control characters to '^X', and aligns inside or against the edges of the box. See [FL_Widget::align\(\)](#) for values of *align*. The value FL_ALIGN_INSIDE is ignored, as this function always prints inside the box. If *img* is provided and is not NULL, the image is drawn above or below the text as specified by the *align* value. The *draw_symbols* argument specifies whether or not to look for symbol names starting with the '@' character. The text length is limited to 1024 characters per line.

29.6.4.12 **FL_EXPORT void fl_draw (const char * *str*, int *x*, int *y*)**

Draws a nul-terminated string starting at the given location.

Text is aligned to the left and to the baseline of the font. To align to the bottom, subtract [fl_descent\(\)](#) from *y*. To align to the top, subtract [fl_descent\(\)](#) and add [fl_height\(\)](#). This version of *fl_draw* provides direct access to the text drawing function of the underlying OS. It does not apply any special handling to control characters.

29.6.4.13 **FL_EXPORT void fl_draw_box (FL_Boxtype *t*, int *x*, int *y*, int *w*, int *h*, FL_Color *c*)**

Draws a box using given type, position, size and color.

Parameters:

- ← *t* box type
- ← *x,y,w,h* position and size
- ← *c* color

29.6.4.14 **FL_EXPORT void fl_draw_image (FL_Draw_Image_Cb *cb*, void * *data*, int *X*, int *Y*, int *W*, int *H*, int *D* = 3)**

Draw image using callback function to generate image data.

You can generate the image as it is being drawn, or do arbitrary decompression of stored data, provided it can be decompressed to individual scan lines easily.

Parameters:

- ← *cb* callback function to generate scan line data
- ← *data* user data passed to callback function
- ← *X,Y*
- ← *W,H*
- ← *D*

See also:

[fl_draw_image\(const uchar* buf, int X,int Y,int W,int H, int D, int L\)](#)

The callback function `cb` is called with the `void*` `data` user data pointer to allow access to a structure of information about the image, and the `x`, `y`, and `w` of the scan line desired from the image. 0,0 is the upper-left corner of the image, not `X`, `Y`. A pointer to a buffer to put the data into is passed. You must copy `w` pixels from scanline `y`, starting at pixel `x`, to this buffer.

Due to cropping, less than the whole image may be requested. So `x` may be greater than zero, the first `y` may be greater than zero, and `w` may be less than `W`. The buffer is long enough to store the entire `W * D` pixels, this is for convenience with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if `x` is not zero, copy the data over so the `x'`th pixel is at the start of the buffer.

You can assume the `y'`s will be consecutive, except the first one may be greater than zero.

If `D` is 4 or more, you must fill in the unused bytes with zero.

29.6.4.15 `FL_EXPORT void fl_draw_image (const uchar *buf, int X, int Y, int W, int H, int D = 3, int L = 0)`

Draw an 8-bit per color RGB or luminance image.

Parameters:

- ← *buf* points at the "r" data of the top-left pixel. Color data must be in *r*, *g*, *b* order.
- ← *X,Y* position where to put top-left corner of image
- ← *W,H* size of the image
- ← *D* delta to add to the pointer between pixels. it may be any value greater than or equal to 3, or it can be negative to flip the image horizontally
- ← *L* delta to add to the pointer between lines (if 0 is passed it uses `W * D`), and may be larger than `W * D` to crop data, or negative to flip the image vertically

It is highly recommended that you put the following code before the first `show()` of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling `fl_draw_image_mono()`. Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting `D` greater than 1 will let you display one channel of a color image.

Note:

The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

29.6.4.16 `FL_EXPORT void fl_draw_image_mono (Fl_Draw_Image_Cb cb, void *data, int X, int Y, int W, int H, int D = 1)`

Draw gray-scale image using callback function to generate image data.

See also:

[fl_draw_image\(Fl_Draw_Image_Cb cb, void* data, int X,int Y,int W,int H, int D\)](#)

29.6.4.17 **FL_EXPORT** void **fl_draw_image_mono** (const uchar * *buf*, int *X*, int *Y*, int *W*, int *H*, int *D* = 1, int *L* = 0)

Draw a gray-scale (1 channel) image.

See also:

[fl_draw_image\(const uchar* buf, int X,int Y,int W,int H, int D, int L\)](#)

29.6.4.18 **FL_EXPORT** int **fl_draw_pixmap** (const char *const * *cdata*, int *x*, int *y*, FL_Color *bg*)

Draw XPM image data, with the top-left corner at the given position.

See also:

[fl_draw_pixmap\(char* const* data, int x, int y, FL_Color bg\)](#)

29.6.4.19 **FL_EXPORT** int **fl_draw_pixmap** (char *const * *data*, int *x*, int *y*, FL_Color *bg*)

Draw XPM image data, with the top-left corner at the given position.

The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps.

Parameters:

- ← *data* pointer to XPM image data
- ← *x,y* position of top-left corner
- ← *bg* background color

Returns:

0 if there was any error decoding the XPM data.

29.6.4.20 **FL_EXPORT** int **fl_draw_symbol** (const char * *label*, int *x*, int *y*, int *w*, int *h*, FL_Color *col*)

Draw the named symbol in the given rectangle using the given color.

Parameters:

- ← *label* name of symbol
- ← *x,y* position of symbol
- ← *w,h* size of symbol
- ← *col* color of symbox

Returns:

1 on success, 0 on failure

29.6.4.21 **FL_EXPORT** `const char* fl_expand_text (const char *from, char *buf, int maxbuf, double maxw, int &n, double &width, int wrap, int draw_symbols)`

Copy `from` to `buf`, replacing unprintable characters with `^X` and `\nnn`.

Stop at a newline or if `MAXBUF` characters written to buffer. Also word-wrap if width exceeds `maxw`. Returns a pointer to the start of the next line of characters. Sets `n` to the number of characters put into the buffer. Sets `width` to the width of the string in the current font.

29.6.4.22 **FL_EXPORT** `void fl_frame (const char *s, int x, int y, int w, int h)`

Draws a series of line segments around the given box.

The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: top, left, bottom, right. The result of calling `fl_frame()` with a string that is not a multiple of 4 characters in length is undefined. The only difference between this function and `fl_frame2()` is the order of the line segments.

Parameters:

- ← `s` sets of 4 grayscale values in top, left, bottom, right order
- ← `x,y,w,h` position and size

29.6.4.23 **FL_EXPORT** `void fl_frame2 (const char *s, int x, int y, int w, int h)`

Draws a series of line segments around the given box.

The string `s` must contain groups of 4 letters which specify one of 24 standard grayscale values, where 'A' is black and 'X' is white. The order of each set of 4 characters is: bottom, right, top, left. The result of calling `fl_frame2()` with a string that is not a multiple of 4 characters in length is undefined. The only difference between this function and `fl_frame()` is the order of the line segments.

Parameters:

- ← `s` sets of 4 grayscale values in bottom, right, top, left order
- ← `x,y,w,h` position and size

29.6.4.24 **FL_EXPORT** `void fl_gap ()`

Call `fl_gap()` to separate loops of the path.

It is unnecessary but harmless to call `fl_gap()` before the first vertex, after the last vertex, or several times in a row.

29.6.4.25 **FL_EXPORT** `void fl_line_style (int style, int width, char *dashes)`

Sets how to draw lines (the "pen").

If you change this it is your responsibility to set it back to the default using `fl_line_style(0)`.

Parameters:

- ← `style` A bitmask which is a bitwise-OR of a line style, a cap style, and a join style. If you don't specify a dash type you will get a solid line. If you don't specify a cap or join type you will get a system-defined default of whatever value is fastest.

- ← **width** The thickness of the lines in pixels. Zero results in the system defined default, which on both X and Windows is somewhat different and nicer than 1.
- ← **dashes** A pointer to an array of dash lengths, measured in pixels. The first location is how long to draw a solid portion, the next is how long to draw the gap, then the solid, etc. It is terminated with a zero-length entry. A `NULL` pointer or a zero-length array results in a solid line. Odd array sizes are not supported and result in undefined behavior.

Note:

Because of how line styles are implemented on Win32 systems, you *must* set the line style *after* setting the drawing color. If you set the color after the line style you will lose the line style settings. The `dashes` array does not work under Windows 95, 98 or Me, since those operating systems do not support complex line styles.

29.6.4.26 FL_EXPORT void fl_measure (const char *str, int &w, int &h, int draw_symbols)

Measure how wide and tall the string will be when printed by the `fl_draw()` function with `align` parameter. If the incoming `w` is non-zero it will wrap to that width.

Parameters:

- ← **str** nul-terminated string
- **w,h** width and height of string in current font
- ← **draw_symbols** non-zero to enable @symbol handling [default=1]

29.6.4.27 FL_EXPORT int fl_measure_pixmap (const char *const *cdata, int &w, int &h)

Get the dimensions of a pixmap.

See also:

[fl_measure_pixmap\(char* const* data, int &w, int &h\)](#)

29.6.4.28 FL_EXPORT int fl_measure_pixmap (char *const *data, int &w, int &h)

Get the dimensions of a pixmap.

An XPM image contains the dimensions in its data. This function returns te width and height.

Parameters:

- ← **data** pointer to XPM image data.
- **w,h** width and height of image

Returns:

non-zero if the dimensions were parsed OK
0 if there were any problems

29.6.4.29 FL_EXPORT void fl_mult_matrix (double *a*, double *b*, double *c*, double *d*, double *x*, double *y*)

Concatenates another transformation onto the current one.

Parameters:

← *a,b,c,d,x,y* transformation matrix elements such that $X' = aX + cY + x$ and $Y' = bX + dY + y$

29.6.4.30 FL_EXPORT int fl_not_clipped (int *x*, int *y*, int *w*, int *h*)

Does the rectangle intersect the current clip region?

Parameters:

← *x,y,w,h* position and size of rectangle

Returns:

non-zero if any of the rectangle intersects the current clip region. If this returns 0 you don't have to draw the object.

Note:

Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.

29.6.4.31 FL_EXPORT void fl_pie (int *x*, int *y*, int *w*, int *h*, double *a1*, double *a2*)

Draw filled ellipse sections using integer coordinates.

Like [fl_arc\(\)](#), but [fl_pie\(\)](#) draws a filled-in pie slice. This slice may extend outside the line drawn by [fl_arc\(\)](#); to avoid this use *w* - 1 and *h* - 1.

Parameters:

← *x,y,w,h* bounding box of complete circle

← *a1,a2* start and end angles of arc measured in degrees counter-clockwise from 3 o'clock. *a2* must be greater than or equal to *a1*.

29.6.4.32 FL_EXPORT void fl_polygon (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*)

Fills a 4-sided polygon.

The polygon must be convex.

29.6.4.33 FL_EXPORT void fl_polygon (int *x*, int *y*, int *x1*, int *y1*, int *x2*, int *y2*)

Fills a 3-sided polygon.

The polygon must be convex.

29.6.4.34 FL_EXPORT void fl_pop_clip ()

Restores the previous clip region.

You must call [fl_pop_clip\(\)](#) once for every time you call [fl_push_clip\(\)](#). Unpredictable results may occur if the clip stack is not empty when you return to FLTK.

29.6.4.35 FL_EXPORT void fl_push_clip (int *x*, int *y*, int *w*, int *h*)

Intersects the current clip region with a rectangle and pushes this new region onto the stack.

Parameters:

← *x,y,w,h* position and size

29.6.4.36 FL_EXPORT void fl_push_matrix ()

Saves the current transformation matrix on the stack.

The maximum depth of the stack is 4.

29.6.4.37 FL_EXPORT uchar* fl_read_image (uchar **p*, int *X*, int *Y*, int *W*, int *H*, int *alpha* = 0)

Read an RGB(A) image from the current window or off-screen buffer.

Parameters:

- ← *p* pixel buffer, or NULL to allocate one
- ← *X,Y* position of top-left of image to read
- ← *W,H* width and height of image to read
- ← *alpha* alpha value for image (0 fr none)

Returns:

pointer to pixel buffer, or NULL if allocation failed.

The *p* argument points to a buffer that can hold the image and must be at least $W \times H \times 3$ bytes when reading RGB images, or $W \times H \times 4$ bytes when reading RGBA images. If NULL, [fl_read_image\(\)](#) will create an array of the proper size which can be freed using `delete[]`.

The *alpha* parameter controls whether an alpha channel is created and the value that is placed in the alpha channel. If 0, no alpha channel is generated.

29.6.4.38 FL_EXPORT void fl_rectf (int *x*, int *y*, int *w*, int *h*, uchar *r*, uchar *g*, uchar *b*)

Color a rectangle with "exactly" the passed *r, g, b* color.

On screens with less than 24 bits of color this is done by drawing a solid-colored block using [fl_draw_image\(\)](#) so that the correct color shade is produced.

29.6.4.39 FL_EXPORT void fl_reset_spot (void)**Todo**

provide user documentation for fl_reset_spot function

29.6.4.40 FL_EXPORT void fl_rotate (double *d*)

Concatenates rotation transformation onto the current one.

Parameters:

← *d* - rotation angle, counter-clockwise in degrees (not radians)

29.6.4.41 FL_EXPORT void fl_scale (double *x*)

Concatenates scaling transformation onto the current one.

Parameters:

← *x* scale factor in both x-direction and y-direction

29.6.4.42 FL_EXPORT void fl_scale (double *x*, double *y*)

Concatenates scaling transformation onto the current one.

Parameters:

← *x,y* scale factors in x-direction and y-direction

29.6.4.43 FL_EXPORT void fl_scroll (int *X*, int *Y*, int *W*, int *H*, int *dx*, int *dy*, void(*) (void *, int, int, int) *draw_area*, void * *data*)

Scroll a rectangle and draw the newly exposed portions.

Parameters:

← *X,Y* position of top-left of rectangle

← *W,H* size of rectangle

← *dx,dy* pixel offsets for shifting rectangle

← *draw_area* callback function to draw rectangular areas

← *data* pointer to user data for callback The contents of the rectangular area is first shifted by *dx* and *dy* pixels. The *draw_area* callback is then called for every newly exposed rectangular area.

29.6.4.44 FL_EXPORT void fl_set_spot (int *font*, int *size*, int *X*, int *Y*, int *W*, int *H*, FL_Window * *win* = 0)**Todo**

provide user documentation for fl_set_spot function

29.6.4.45 FL_EXPORT void fl_set_status (int X, int Y, int W, int H)**Todo**

provide user documentation for fl_set_status function

29.6.4.46 FL_EXPORT const char* fl_shortcut_label (int shortcut)

Get a human-readable string from a shortcut value.

Unparse a shortcut value as used by [Fl_Button](#) or [Fl_Menu_Item](#) into a human-readable string like "Alt+N". This only works if the shortcut is a character key or a numbered function key. If the shortcut is zero then an empty string is returned. The return value points at a static buffer that is overwritten with each call.

29.6.4.47 FL_EXPORT double fl_transform_dx (double x, double y)

Transforms distance using current transformation matrix.

Parameters:

← *x,y* coordinate

29.6.4.48 FL_EXPORT double fl_transform_dy (double x, double y)

Transforms distance using current transformation matrix.

Parameters:

← *x,y* coordinate

29.6.4.49 FL_EXPORT double fl_transform_x (double x, double y)

Transforms coordinate using the current transformation matrix.

Parameters:

← *x,y* coordinate

29.6.4.50 FL_EXPORT double fl_transform_y (double x, double y)

Transform coordinate using the current transformation matrix.

Parameters:

← *x,y* coordinate

29.6.4.51 FL_EXPORT void fl_transformed_vertex (double *xf*, double *yf*)

Adds coordinate pair to the vertex list without further transformations.

Parameters:

← *xf,yf* transformed coordinate

29.6.4.52 FL_EXPORT void fl_translate (double *x*, double *y*)

Concatenates translation transformation onto the current one.

Parameters:

← *x,y* translation factor in x-direction and y-direction

29.6.4.53 FL_EXPORT void fl_vertex (double *x*, double *y*)

Adds a single vertex to the current path.

Parameters:

← *x,y* coordinate

29.7 Multithreading support functions

fl multithreading support functions

Functions

- static int [Fl::awake](#) (Fl_Awake_Handler cb, void *message=0)
See void awake(void message=0).*
- static void [Fl::awake](#) (void *message=0)
The [awake\(\)](#) method sends a message pointer to the main thread, causing any pending [Fl::wait\(\)](#) call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.
- static void [Fl::lock](#) ()
The [lock\(\)](#) method blocks the current thread until it can safely access FLTK widgets and data.
- static void * [Fl::thread_message](#) ()
The [thread_message\(\)](#) method returns the last message that was sent from a child by the [awake\(\)](#) method.
- static void [Fl::unlock](#) ()
The [unlock\(\)](#) method releases the lock that was set using the [lock\(\)](#) method.

29.7.1 Detailed Description

fl multithreading support functions

29.7.2 Function Documentation

29.7.2.1 int Fl::awake (Fl_Awake_Handler cb, void * message = 0) [static, inherited]

See void awake(void* message=0).

Let the main thread know an update is pending and have it call a specific function See void awake(void* message=0).

29.7.2.2 void Fl::awake (void * msg = 0) [static, inherited]

The [awake\(\)](#) method sends a message pointer to the main thread, causing any pending [Fl::wait\(\)](#) call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.

Multiple calls to [Fl::awake\(\)](#) will queue multiple pointers for the main thread to process, up to a system-defined (typically several thousand) depth. The default message handler saves the last message which can be accessed using the [Fl::thread_message\(\)](#) function.

The second form of [awake\(\)](#) registers a function that will be called by the main thread during the next message handling cycle. [awake\(\)](#) will return 0 if the callback function was registered, and -1 if registration failed. Over a thousand awake callbacks can be registered simultaneously.

See also: multithreading.

29.7.2.3 void Fl::lock () [static, inherited]

The [lock\(\)](#) method blocks the current thread until it can safely access FLTK widgets and data.

Child threads should call this method prior to updating any widgets or accessing data. The main thread must call [lock\(\)](#) to initialize the threading support in FLTK.

Child threads must call [unlock\(\)](#) when they are done accessing FLTK.

When the [wait\(\)](#) method is waiting for input or timeouts, child threads are given access to FLTK. Similarly, when the main thread needs to do processing, it will wait until all child threads have called [unlock\(\)](#) before processing additional data.

See also: multithreading

29.7.2.4 static void* Fl::thread_message () [static, inherited]

The [thread_message\(\)](#) method returns the last message that was sent from a child by the [awake\(\)](#) method.

See also: multithreading

29.7.2.5 void Fl::unlock () [static, inherited]

The [unlock\(\)](#) method releases the lock that was set using the [lock\(\)](#) method.

Child threads should call this method as soon as they are finished accessing FLTK.

See also: multithreading

29.8 Safe widget deletion support functions

These functions support deletion of widgets inside callbacks.

Functions

- static void [Fl::clear_widget_pointer](#) ([Fl_Widget](#) const *w)
Clears a widget pointer in the watch list.
- static void [Fl::delete_widget](#) ([Fl_Widget](#) *w)
Schedules a widget for deletion at the next call to the event loop.
- static void [Fl::do_widget_deletion](#) ()
Deletes widgets previously scheduled for deletion.
- static void [Fl::release_widget_pointer](#) ([Fl_Widget](#) *&w)
Releases a widget pointer from the watch list.
- static void [Fl::watch_widget_pointer](#) ([Fl_Widget](#) *&w)
Adds a widget pointer to the widget watch list.

29.8.1 Detailed Description

These functions support deletion of widgets inside callbacks.

[Fl::delete_widget\(\)](#) should be called when deleting widgets or complete widget trees ([Fl_Group](#), [Fl_Window](#), ...) inside callbacks.

The other functions are intended for internal use. The preferred way to use them is by using the helper class [Fl_Widget_Tracker](#).

The following is to show how it works ...

There are three groups of related methods:

1. scheduled widget deletion
 - [Fl::delete_widget\(\)](#) schedules widgets for deletion
 - [Fl::do_widget_deletion\(\)](#) deletes all scheduled widgets
2. widget watch list ("smart pointers")
 - [Fl::watch_widget_pointer\(\)](#) adds a widget pointer to the watch list
 - [Fl::release_widget_pointer\(\)](#) removes a widget pointer from the watch list
 - [Fl::clear_widget_pointer\(\)](#) clears a widget pointer *in* the watch list
3. the class [Fl_Widget_Tracker](#):
 - the constructor calls [Fl::watch_widget_pointer\(\)](#)
 - the destructor calls [Fl::release_widget_pointer\(\)](#)
 - the access methods can be used to test, if a widget has been deleted

See also:

[Fl_Widget_Tracker](#).

29.8.2 Function Documentation

29.8.2.1 void Fl::clear_widget_pointer (Fl_Widget const * w) [static, inherited]

Clears a widget pointer *in* the watch list.

This is called when a widget is destroyed (by its destructor). You should never call this directly.

Note:

Internal use only !

This method searches the widget watch list for pointers to the widget and clears each pointer that points to it. Widget pointers can be added to the widget watch list by calling [Fl::watch_widget_pointer\(\)](#) or by using the helper class [Fl_Widget_Tracker](#) (recommended).

See also:

[Fl::watch_widget_pointer\(\)](#)
class [Fl_Widget_Tracker](#)

29.8.2.2 void Fl::delete_widget (Fl_Widget * wi) [static, inherited]

Schedules a widget for deletion at the next call to the event loop.

Use this method to delete a widget inside a callback function. To avoid early deletion of widgets, this function should be called toward the end of a callback and only after any call to the event loop ([Fl::wait\(\)](#), [Fl::flush\(\)](#), [fl_ask\(\)](#), etc).

When deleting groups or windows, you must only delete the group or window widget and not the individual child widgets.

29.8.2.3 void Fl::do_widget_deletion () [static, inherited]

Deletes widgets previously scheduled for deletion.

This is for internal use only. You should never call this directly.

[Fl::do_widget_deletion\(\)](#) is called from the FLTK event loop or whenever you call [Fl::wait\(\)](#). The previously scheduled widgets are deleted in the same order they were scheduled by calling [Fl::delete_widget\(\)](#).

See also:

[Fl::delete_widget\(Fl_Widget *wi\)](#)

29.8.2.4 void Fl::release_widget_pointer (Fl_Widget *& w) [static, inherited]

Releases a widget pointer from the watch list.

This is used to remove a widget pointer that has been added to the watch list with [Fl::watch_widget_pointer\(\)](#), when it is not needed anymore.

Note:

Internal use only, please use class [Fl_Widget_Tracker](#) instead.

See also:

[Fl::watch_widget_pointer\(\)](#)

29.8.2.5 void Fl::watch_widget_pointer (Fl_Widget *& w) [static, inherited]

Adds a widget pointer to the widget watch list.

Note:

Internal use only, please use class [Fl_Widget_Tracker](#) instead.

This can be used, if it is possible that a widget might be deleted during a callback or similar function. The widget pointer must be added to the watch list before calling the callback. After the callback the widget pointer can be queried, if it is NULL. *If* it is NULL, then the widget has been deleted during the callback and must not be accessed anymore. If the widget pointer is *not* NULL, then the widget has not been deleted and can be accessed safely.

After accessing the widget, the widget pointer must be released from the watch list by calling [Fl::release_widget_pointer\(\)](#).

Example for a button that is clicked (from its [handle\(\)](#) method):

```
Fl_Widget *wp = this;           // save 'this' in a pointer variable
Fl::watch_widget_pointer(wp);    // add the pointer to the watch list
set_changed();                  // set the changed flag
do_callback();                  // call the callback
if (!wp) {                       // the widget has been deleted

    // DO NOT ACCESS THE DELETED WIDGET !

} else {                         // the widget still exists
    clear_changed();             // reset the changed flag
}

Fl::release_widget_pointer(wp);   // remove the pointer from the watch list
```

This works, because all widgets call [Fl::clear_widget_pointer\(\)](#) in their destructors.

See also:

[Fl::release_widget_pointer\(\)](#)
[Fl::clear_widget_pointer\(\)](#)

An easier and more convenient method to control widget deletion during callbacks is to use the class [Fl_Widget_Tracker](#) with a local (automatic) variable.

See also:

class [Fl_Widget_Tracker](#)

29.9 Cairo support functions and classes

Classes

- class [Fl_Cairo_State](#)
Contains all the necessary info on the current cairo context.
- class [Fl_Cairo_Window](#)
This defines a pre-configured cairo fltk window.

Functions

- static bool [Fl::cairo_autolink_context](#) ()
Gets the current autolink mode for cairo support.
- static void [Fl::cairo_autolink_context](#) (bool alink)
when HAVE_CAIRO is defined and [cairo_autolink_context\(\)](#) is true, any current window dc is linked to a current context.
- static void [Fl::cairo_cc](#) (cairo_t *c, bool own=false)
Sets the current cairo context to c.
- static cairo_t * [Fl::cairo_cc](#) ()
Gets the current cairo context linked with a fltk window.
- static cairo_t * [Fl::cairo_make_current](#) ([Fl_Window](#) *w)

29.9.1 Function Documentation

29.9.1.1 static bool [Fl::cairo_autolink_context](#) () [inline, static, inherited]

Gets the current autolink mode for cairo support.

Return values:

- false* if no cairo context autolink is made for each window.
true if any fltk window is attached a cairo context when it is current.

See also:

void [cairo_autolink_context](#)(bool alink)

Note:

Only available when configure has the `--enable-cairo` option

29.9.1.2 static void Fl::cairo_autolink_context (bool *alink*) [inline, static, inherited]

when HAVE_CAIRO is defined and [cairo_autolink_context\(\)](#) is true, any current window dc is linked to a current context.

This is not the default, because it may not be necessary to add cairo support to all fltk supported windows. When you wish to associate a cairo context in this mode, you need to call explicitly in your draw() overridden method, `Fl::cairo_make_current(Fl_Window*)`. This will create a cairo context but only for this Window. Still in custom cairo application it is possible to handle completely this process automatically by setting `alink` to true. In this last case, you don't need anymore to call `Fl::cairo_make_current()`. You can use [Fl::cairo_cc\(\)](#) to get the current cairo context anytime.

Note:

Only available when configure has the `--enable-cairo` option

29.9.1.3 static void Fl::cairo_cc (cairo_t * *c*, bool *own* = false) [inline, static, inherited]

Sets the current cairo context to *c*.

Set *own* to true if you want fltk to handle this cc deletion.

Note:

Only available when configure has the `--enable-cairo` option

29.9.1.4 static cairo_t* Fl::cairo_cc () [inline, static, inherited]

Gets the current cairo context linked with a fltk window.

29.10 Common Dialogs classes and functions

Classes

- class [FL_Color_Chooser](#)

The [FL_Color_Chooser](#) widget provides a standard RGB color chooser.

- class [FL_File_Chooser](#)

The [FL_File_Chooser](#) widget displays a standard file selection dialog that supports various selection modes.

Functions

- void [fl_alert](#) (const char *fmt,...)
Shows an alert message dialog box.
- int [fl_ask](#) (const char *fmt,...)
Shows a dialog displaying the `fmt` message, this dialog features 2 yes/no buttons.
- void [fl_beep](#) (int type)
Emits a system beep message.
- int [fl_choice](#) (const char *fmt, const char *b0, const char *b1, const char *b2,...)
Shows a dialog displaying the `fmt` message, this dialog features up to 3 customizable choice buttons.
- int [FL_Color_Chooser::fl_color_chooser](#) (const char *name, [uchar](#) &r, [uchar](#) &g, [uchar](#) &b)
Pops up a window to let the user pick an arbitrary RGB color.
- int [FL_Color_Chooser::fl_color_chooser](#) (const char *name, double &r, double &g, double &b)
Pops up a window to let the user pick an arbitrary RGB color.
- char * [FL_File_Chooser::fl_dir_chooser](#) (const char *message, const char *fname, int relative)
- char * [FL_File_Chooser::fl_file_chooser](#) (const char *message, const char *pat, const char *fname, int relative)
- void [FL_File_Chooser::fl_file_chooser_callback](#) (void(*cb)(const char *))
- void [fl_file_chooser_callback](#) (void(*cb)(const char *))
- void [FL_File_Chooser::fl_file_chooser_ok_label](#) (const char *l)
- void [fl_file_chooser_ok_label](#) (const char *l)
- const char * [fl_input](#) (const char *fmt, const char *defstr,...)
Shows an input dialog displaying the `fmt` message.
- void [fl_message](#) (const char *fmt,...)
Shows an information message dialog box.
- [FL_Widget](#) * [fl_message_icon](#) ()
Gets the [FL_Box](#) icon container of the current default dialog used in many common dialogs like [fl_message\(\)](#), [fl_alert\(\)](#), [fl_ask\(\)](#), [fl_choice\(\)](#), [fl_input\(\)](#), [fl_password\(\)](#).
- const char * [fl_password](#) (const char *fmt, const char *defstr,...)
Shows an input dialog displaying the `fmt` message.

Variables

- static void(* [Fl::error](#))(const char *,...) = ::error
FLTK calls [Fl::error\(\)](#) to output a normal error message.
- static void(* [Fl::fatal](#))(const char *,...) = ::fatal
FLTK calls [Fl::fatal\(\)](#) to output a fatal error message.
- const char * [fl_cancel](#) = "Cancel"
string pointer used in common dialogs, you can change it to a foreign language
- const char * [fl_close](#) = "Close"
string pointer used in common dialogs, you can change it to a foreign language
- const char * [fl_no](#) = "No"
string pointer used in common dialogs, you can change it to a foreign language
- const char * [fl_ok](#) = "OK"
string pointer used in common dialogs, you can change it to a foreign language
- const char * [fl_yes](#) = "Yes"
string pointer used in common dialogs, you can change it to a foreign language
- static void(* [Fl::warning](#))(const char *,...) = ::warning
FLTK calls [Fl::warning\(\)](#) to output a warning message.

29.10.1 Function Documentation

29.10.1.1 void fl_alert (const char *fmt, ...)

Shows an alert message dialog box.

Parameters:

← *fmt* can be used as an sprintf-like format and variables for the message text

29.10.1.2 int fl_ask (const char *fmt, ...)

Shows a dialog displaying the *fmt* message, this dialog features 2 yes/no buttons.

Parameters:

← *fmt* can be used as an sprintf-like format and variables for the message text

Return values:

- 0* if the no button is selected
- 1* if yes is selected

29.10.1.3 int fl_choice (const char * *fmt*, const char * *b0*, const char * *b1*, const char * *b2*, ...)

Shows a dialog displaying the *fmt* message, this dialog features up to 3 customizable choice buttons.

Parameters:

- ← *fmt* can be used as an sprintf-like format and variables for the message text
- ← *b0* text label of button 0
- ← *b1* text label of button 1
- ← *b2* text label of button 2

Return values:

- 0* if the first button with *b0* text is selected
- 1* if the second button with *b1* text is selected
- 2* if the third button with *b2* text is selected

29.10.1.4 int fl_color_chooser (const char * *name*, uchar & *r*, uchar & *g*, uchar & *b*) [related, inherited]

Pops up a window to let the user pick an arbitrary RGB color.

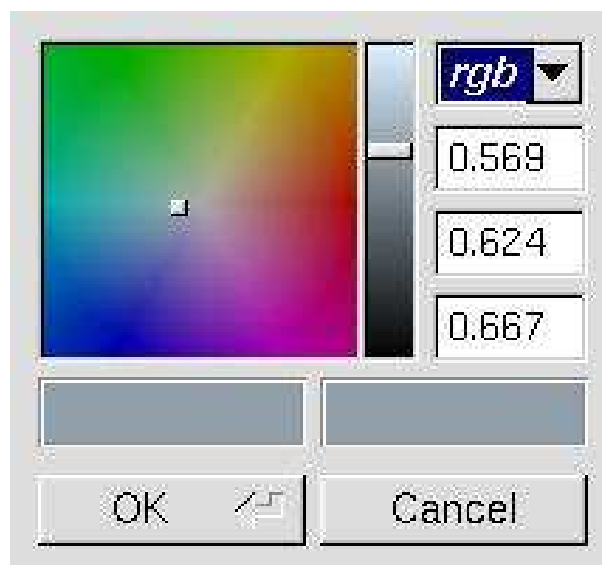


Figure 29.1: fl_color_chooser

Parameters:

- ← *name* title label for the window
- ↔ *r,g,b* color components in the range 0 to 255.

Return values:

- 1* if user confirms the selection
- 0* if user cancels the dialog

29.10.1.5 int fl_color_chooser (const char * *name*, double & *r*, double & *g*, double & *b*)
[related, inherited]

Pops up a window to let the user pick an arbitrary RGB color.

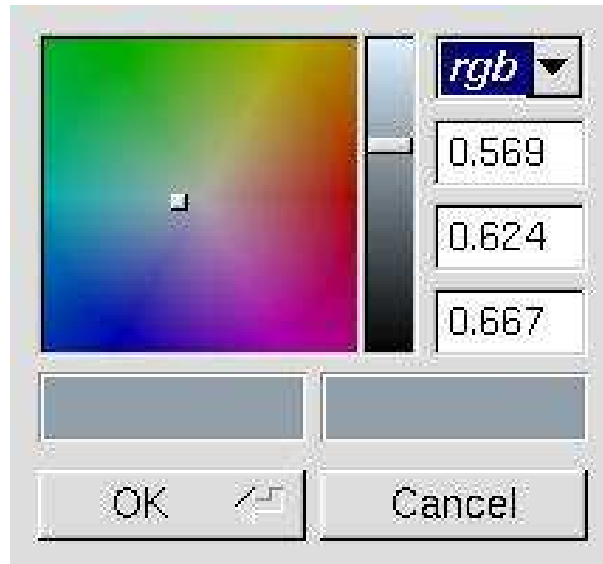


Figure 29.2: fl_color_chooser

Parameters:

- ← *name* title label for the window
- ↔ *r,g,b* color components in the range 0.0 to 1.0.

Return values:

- 1* if user confirms the selection
- 0* if user cancels the dialog

29.10.1.6 char * fl_dir_chooser (const char * *message*, const char * *fname*, int *relative*)
[related, inherited]

Shows a file chooser dialog and gets a directory.

Parameters:

- ← *message* title bar text
- ← *fname* initial/default directory name
- ← *relative* 0 for absolute path return, relative otherwise

Returns:

- the directory path string chosen by the user or NULL if user cancels

29.10.1.7 `char * fl_file_chooser (const char * message, const char * pat, const char * fname, int relative)` [related, inherited]

Shows a file chooser dialog and gets a filename.

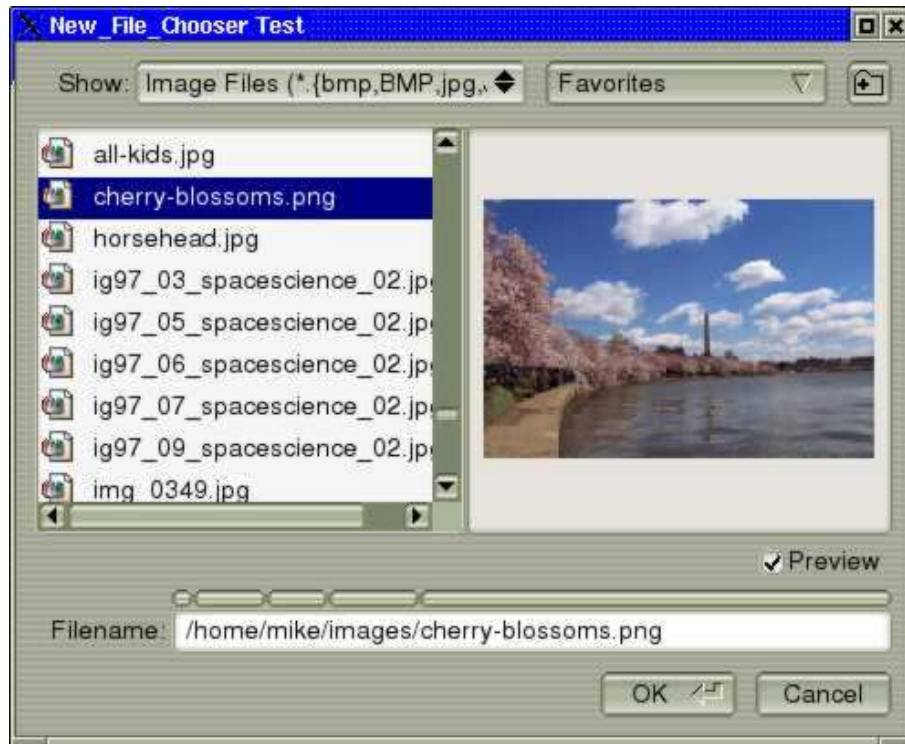


Figure 29.3: Fl_File_Chooser

Parameters:

- ← *message* text in title bar
- ← *pat* filename pattern filter
- ← *fname* initial/default filename selection
- ← *relative* 0 for absolute path name, relative path name otherwise

Returns:

the user selected filename, in absolute or relative format or NULL if user cancels

29.10.1.8 `void fl_file_chooser_callback (void(*) (const char *) cb)` [related, inherited]

Set the file chooser callback

29.10.1.9 `void fl_file_chooser_callback (void(*) (const char *) cb)`

Set the file chooser callback

29.10.1.10 void fl_file_chooser_ok_label (const char * l) [related, inherited]

Set the "OK" button label

29.10.1.11 void fl_file_chooser_ok_label (const char * l)

Set the "OK" button label

29.10.1.12 const char* fl_input (const char * fmt, const char * defstr, ...)

Shows an input dialog displaying the `fmt` message.

Parameters:

- ← *fmt* can be used as an sprintf-like format and variables for the message text
- ← *defstr* defines the default returned string if no text is entered

Returns:

the user string input if OK was pushed, NULL if Cancel was pushed

29.10.1.13 void fl_message (const char * fmt, ...)

Shows an information message dialog box.

Parameters:

- ← *fmt* can be used as an sprintf-like format and variables for the message text

29.10.1.14 const char* fl_password (const char * fmt, const char * defstr, ...)

Shows an input dialog displaying the `fmt` message.

Like `fl_input()` except the input text is not shown, '*' characters are displayed instead.

Parameters:

- ← *fmt* can be used as an sprintf-like format and variables for the message text
- ← *defstr* defines the default returned string if no text is entered

Returns:

the user string input if OK was pushed, NULL if Cancel was pushed

29.10.2 Variable Documentation**29.10.2.1 void(* Fl::error)(const char *format,...) (const char *, ...) = ::error** [static, inherited]

FLTK calls `Fl::error()` to output a normal error message.

The default version on Windows displays the error message in a MessageBox window.

The default version on all other platforms prints the error message to stderr.

You can override the behavior by setting the function pointer to your own routine.

[Fl::error\(\)](#) means there is a recoverable error such as the inability to read an image file. The default implementation returns after displaying the message.

29.10.2.2 `void(* Fl::fatal)(const char *format,...) (const char *, ...) = ::fatal` [static, inherited]

FLTK calls [Fl::fatal\(\)](#) to output a fatal error message.

The default version on Windows displays the error message in a MessageBox window.

The default version on all other platforms prints the error message to stderr.

You can override the behavior by setting the function pointer to your own routine.

[Fl::fatal\(\)](#) must not return, as FLTK is in an unusable state, however your version may be able to use longjmp or an exception to continue, as long as it does not call FLTK again. The default implementation exits with status 1 after displaying the message.

29.10.2.3 `void(* Fl::warning)(const char *format,...) (const char *, ...) = ::warning` [static, inherited]

FLTK calls [Fl::warning\(\)](#) to output a warning message.

The default version on Windows returns *without* printing a warning message, because Windows programs normally don't have stderr (a console window) enabled.

The default version on all other platforms prints the warning message to stderr.

You can override the behavior by setting the function pointer to your own routine.

[Fl::warning\(\)](#) means that there was a recoverable problem, the display may be messed up, but the user can probably keep working - all X protocol errors call this, for example. The default implementation returns after displaying the message.

29.11 File names and URI utility functions

Defines

- `#define FL_PATH_MAX 256`
all path buffers should use this length

Typedefs

- `typedef int(FL_File_Sort_F)(struct dirent **, struct dirent **)`
File sorting function.

Functions

- `FL_EXPORT int fl_filename_absolute (char *to, int tolen, const char *from)`
Makes a filename absolute from a relative filename.
- `FL_EXPORT int fl_filename_expand (char *to, int tolen, const char *from)`
Expands a filename coontaining shell variables.
- `FL_EXPORT const char * fl_filename_ext (const char *)`
Gets the extensions of a filename.
- `FL_EXPORT int fl_filename_isdir (const char *name)`
Determines if a file exists and is a directory from its filename.
- `FL_EXPORT int fl_filename_match (const char *name, const char *pattern)`
*Checks if a string *s* matches a pattern *p*.*
- `FL_EXPORT const char * fl_filename_name (const char *filename)`
Gets the file name from a path.
- `FL_EXPORT int fl_filename_relative (char *to, int tolen, const char *from)`
Makes a filename relative to the current working directory.
- `FL_EXPORT char * fl_filename_setext (char *to, int tolen, const char *ext)`
*Replaces the extension in *buf* of *max*.*

29.11.1 Typedef Documentation

29.11.1.1 `typedef int(FL_File_Sort_F)(struct dirent **, struct dirent **)`

File sorting function.

See also:

`fl_filename_list()`

29.11.2 Function Documentation

29.11.2.1 FL_EXPORT int fl_filename_absolute (char * *to*, int *tolen*, const char * *from*)

Makes a filename absolute from a relative filename.

Parameters:

- *to* resulting absolute filename
- ← *tolen* size of the absolute filename buffer
- ← *from* relative filename

Returns:

0 if no change, non zero otherwise

29.11.2.2 FL_EXPORT int fl_filename_expand (char * *to*, int *tolen*, const char * *from*)

Expands a filename coontaining shell variables.

Parameters:

- *to* resulting expanded filename
- ← *tolen* size of the expanded filename buffer
- ← *from* filename containing shell variables

Returns:

0 if no change, non zero otherwise

29.11.2.3 FL_EXPORT const char* fl_filename_ext (const char * *buf*)

Gets the extensions of a filename.

Parameters:

- ← *buf* the filename to be parsed

Returns:

a pointer to the extension (including '.') if any or NULL otherwise

29.11.2.4 FL_EXPORT int fl_filename_isdir (const char * *n*)

Determines if a file exists and is a directory from its filename.

Parameters:

- ← *n* the filename to parse

Returns:

non zero if file exists and is a directory, zero otherwise

29.11.2.5 FL_EXPORT int fl_filename_match (const char * *s*, const char * *p*)

Checks if a string *s* matches a pattern *p*.

The following syntax is used for the pattern:

- `*` matches any sequence of 0 or more characters.
- `?` matches any single character.
- `[set]` matches any character in the set. Set can contain any single characters, or a-z to represent a range. To match `]` or `-` they must be the first characters. To match `^` or `!` they must not be the first characters.
- `[^set]` or `[!set]` matches any character not in the set.
- `{X|Y|Z}` or `{X,Y,Z}` matches any one of the subexpressions literally.
- `\x` quotes the character *x* so it has no special meaning.
- *x* all other characters must be matched exactly.

Parameters:

← *s* the string to check for a match

← *p* the string pattern

Returns:

non zero if the string matches the pattern

29.11.2.6 FL_EXPORT const char* fl_filename_name (const char * *filename*)

Gets the file name from a path.

Returns:

a pointer to the char after the last slash, or to *filename* if there is none.

29.11.2.7 FL_EXPORT int fl_filename_relative (char * *to*, int *tolen*, const char * *from*)

Makes a filename relative to the current working directory.

Parameters:

→ *to* resulting relative filename

← *tolen* size of the relative filename buffer

← *from* absolute filename

Returns:

0 if no change, non zero otherwise

29.11.2.8 FL_EXPORT char* fl_filename_setext (char * *buf*, int *buflen*, const char * *ext*)

Replaces the extension in `buf` of max.
size `buflen` with the extension in `ext`.

Returns:

`buf` itself for calling convenience.

Chapter 30

Class Documentation

30.1 Fl Class Reference

The **Fl** is the FLTK global (static) containing state information and global methods for the current application.

```
#include <Fl.H>
```

Static Public Member Functions

- static int **add_awake_handler_** (Fl_Awake_Handler, void *)
Adds an awake handler for use in [awake\(\)](#).
- static void **add_check** (Fl_Timeout_Handler, void *=0)
FLTK will call this callback just before it flushes the display and waits for events.
- static void **add_fd** (int fd, void(*cb)(int, void *), void *=0)
*See void [add_fd](#)(int fd, int when, void (*cb)(int,void*),void* =0).*
- static void **add_fd** (int fd, int when, void(*cb)(int, void *), void *=0)
Adds file descriptor fd to listen to.
- static void **add_handler** (int(*h)(int))
Install a function to parse unrecognized events.
- static void **add_idle** (void(*cb)(void *), void *data=0)
Adds a callback function that is called every time by [Fl::wait\(\)](#) and also makes it act as though the timeout is zero (this makes [Fl::wait\(\)](#) return immediately, so if it is in a loop it is called repeatedly, and thus the idle function is called repeatedly).
- static void **add_timeout** (double t, Fl_Timeout_Handler, void *=0)
Adds a one-shot timeout callback.
- static int **arg** (int, char **, int &)
Consume a single switch from argv, starting at word i.

- static void [args](#) (int, char **)

*See [Fl::args](#)(int argc, char **argv, int& i, int (*cb)(int, char**, int&)).*
- static int [args](#) (int, char **, int &, int(*) (int, char **, int &)=0)

Consume all switches from argv.
- static int [awake](#) (Fl_Awake_Handler cb, void *message=0)

See void [awake](#)(void message=0).*
- static void [awake](#) (void *message=0)

The [awake\(\)](#) method sends a message pointer to the main thread, causing any pending [Fl::wait\(\)](#) call to terminate so that the main thread can retrieve the message and any pending redraws can be processed.
- static void [background](#) (uchar, uchar, uchar)

Changes [fl_color](#)(FL_BACKGROUND_COLOR) to the given color, and changes the gray ramp from 32 to 56 to black to white.
- static void [background2](#) (uchar, uchar, uchar)

Changes the alternative background color.
- static void [belowmouse](#) (Fl_Widget *)

Sets the widget that is below the mouse.
- static Fl_Widget * [belowmouse](#) ()

Gets the widget that is below the mouse.
- static int [box_dh](#) (Fl_Boxtype)

Returns the height offset for the given boxtype.
- static int [box_dw](#) (Fl_Boxtype)

Returns the width offset for the given boxtype.
- static int [box_dx](#) (Fl_Boxtype)

Returns the X offset for the given boxtype.
- static int [box_dy](#) (Fl_Boxtype)

Returns the Y offset for the given boxtype.
- static bool [cairo_autolink_context](#) ()

Gets the current autolink mode for cairo support.
- static void [cairo_autolink_context](#) (bool alink)

when HAVE_CAIRO is defined and [cairo_autolink_context\(\)](#) is true, any current window dc is linked to a current context.
- static void [cairo_cc](#) (cairo_t *c, bool own=false)

Sets the current cairo context to c.
- static cairo_t * [cairo_cc](#) ()

Gets the current cairo context linked with a fltk window.

- static cairo_t * **cairo_make_current** (FL_Window *w)
- static int **check** ()
Same as FL::wait(0).
- static void **clear_widget_pointer** (FL_Widget const *w)
Clears a widget pointer in the watch list.
- static int **compose** (int &del)
Any text editing widget should call this for each FL_KEYBOARD event.
- static void **compose_reset** ()
If the user moves the cursor, be sure to call FL::compose_reset().
- static void **copy** (const char *stuff, int len, int clipboard=0)
Copies the data pointed to by stuff to the selection (0) or primary (1) clipboard.
- static int **damage** ()
If true then flush() will do something.
- static void **damage** (int d)
If true then flush() will do something.
- static void **default_atclose** (FL_Window *, void *)
Default callback for window widgets.
- static void **delete_widget** (FL_Widget *w)
Schedules a widget for deletion at the next call to the event loop.
- static void **display** (const char *)
Sets the X display to use for all windows.
- static int **dnd** ()
Initiate a Drag And Drop operation.
- static int **dnd_text_ops** ()
Gets or sets whether drag and drop text operations are supported.
- static void **dnd_text_ops** (int v)
Gets or sets whether drag and drop text operations are supported.
- static void **do_widget_deletion** ()
Deletes widgets previously scheduled for deletion.
- static int **draw_box_active** ()
Determines if the current draw box is active or inactive.
- static int **event** ()
Returns the last event that was processed.

- static int [event_alt](#) ()
Returns non-zero if the Alt key is pressed.
- static int [event_button](#) ()
Gets which particular mouse button caused the current event.
- static int [event_button1](#) ()
Returns non-zero if mouse button 1 is currently held down.
- static int [event_button2](#) ()
Returns non-zero if button 2 is currently held down.
- static int [event_button3](#) ()
Returns non-zero if button 3 is currently held down.
- static int [event_buttons](#) ()
Returns the mouse buttons state bits; if non-zero, then at least one button is pressed now.
- static void [event_clicks](#) (int i)
Manually sets the number returned by [Fl::event_clicks\(\)](#).
- static int [event_clicks](#) ()
Returns non zero if we had a double click event.
- static int [event_ctrl](#) ()
Returns non-zero if the Control key is pressed.
- static int [event_dx](#) ()
Returns the current horizontal mouse scrolling associated with the [FL_MOUSEWHEEL](#) event.
- static int [event_dy](#) ()
Returns the current vertical mouse scrolling associated with the [FL_MOUSEWHEEL](#) event.
- static int [event_inside](#) (const [Fl_Widget](#) *)
Returns whether or not the mouse event is inside the given widget.
- static int [event_inside](#) (int, int, int, int)
Returns whether or not the mouse event is inside the given rectangle.
- static void [event_is_click](#) (int i)
Only i=0 works! See int [event_is_click\(\)](#).
- static int [event_is_click](#) ()
The first form returns non-zero if the mouse has not moved far enough and not enough time has passed since the last [FL_PUSH](#) or [FL_KEYBOARD](#) event for it to be considered a "drag" rather than a "click".
- static int [event_key](#) (int key)
Returns true if the given `key` was held down (or pressed) during the last event.
- static int [event_key](#) ()

Gets which key on the keyboard was last pushed.

- static int [event_length](#) ()
Returns the length of the text in [Fl::event_text\(\)](#).
- static int [event_original_key](#) ()
Returns the keycode of the last key event, regardless of the NumLock state.
- static int [event_shift](#) ()
Returns non-zero if the Shift key is pressed.
- static int [event_state](#) (int i)
See int [event_state\(\)](#).
- static int [event_state](#) ()
This is a bitfield of what shift states were on and what mouse buttons were held down during the most recent event.
- static const char * [event_text](#) ()
Returns the text associated with the current [FL_PASTE](#) or [FL_DND_RELEASE](#) event.
- static int [event_x](#) ()
Returns the mouse position of the event relative to the [Fl_Window](#) it was passed to.
- static int [event_x_root](#) ()
Returns the mouse position on the screen of the event.
- static int [event_y](#) ()
Returns the mouse position of the event relative to the [Fl_Window](#) it was passed to.
- static int [event_y_root](#) ()
Returns the mouse position on the screen of the event.
- static void [first_window](#) ([Fl_Window](#) *)
See [Fl_Window](#) [first_window\(\)](#).*
- static [Fl_Window](#) * [first_window](#) ()
Returns the first top-level window in the list of [shown\(\)](#) windows.
- static void [flush](#) ()
Causes all the windows that need it to be redrawn and graphics forced out through the pipes.
- static void [focus](#) ([Fl_Widget](#) *)
Sets the widget that will receive [FL_KEYBOARD](#) events.
- static [Fl_Widget](#) * [focus](#) ()
Gets the current [Fl::focus\(\)](#) widget.
- static void [foreground](#) (uchar, uchar, uchar)
Changes [fl_color\(FL_FOREGROUND_COLOR\)](#).

- static void `free_color` (`FL_Color`, int overlay=0)
Frees the specified color from the colormap, if applicable.
- static int `get_awake_handler` (`FL_Awake_Handler` &, void *&)
Gets the last stored awake handler for use in `awake()`.
- static `FL_Box_Draw_F` * `get_boxtype` (`FL_Boxtype`)
Gets the current box drawing function for the specified box type.
- static void `get_color` (`FL_Color`, `uchar` &, `uchar` &, `uchar` &)
See unsigned `get_color(FL_Color c)`.
- static unsigned `get_color` (`FL_Color`)
Returns the RGB value(s) for the given FLTK color index.
- static const char * `get_font` (`FL_Font`)
Gets the string for this face.
- static const char * `get_font_name` (`FL_Font`, int *attributes=0)
Get a human-readable string describing the family of this face.
- static int `get_font_sizes` (`FL_Font`, int *&sizep)
Return an array of sizes in `sizep`.
- static int `get_key` (int key)
Returns true if the given `key` is held down now.
- static void `get_mouse` (int &, int &)
Return where the mouse is on the screen by doing a round-trip query to the server.
- static void `get_system_colors` ()
Read the user preference colors from the system and use them to call `Fl::foreground()`, `Fl::background()`, and `Fl::background2()`.
- static int `gl_visual` (int, int *alist=0)
This does the same thing as `Fl::visual(int)` but also requires OpenGL drawing to work.
- static void `grab` (`FL_Window` &win)
See `FL_Window` `grab()`.*
- static void `grab` (`FL_Window` *)
Selects the window to grab.
- static `FL_Window` * `grab` ()
This is used when pop-up menu systems are active.
- static int `h` ()
Returns the height of the screen in pixels.

- static int `handle` (int, `FL_Window *`)
Sends the event to a window for processing.
- static int `has_check` (`FL_Timeout_Handler`, void `*=0`)
Returns 1 if the check exists and has not been called yet, 0 otherwise.
- static int `has_idle` (void(`*cb`)(void `*`), void `*data=0`)
Returns true if the specified idle callback is currently installed.
- static int `has_timeout` (`FL_Timeout_Handler`, void `*=0`)
Returns true if the timeout exists and has not been called yet.
- static void `lock` ()
The `lock()` method blocks the current thread until it can safely access FLTK widgets and data.
- static `FL_Window *` `modal` ()
Returns the top-most `modal()` window currently shown.
- static `FL_Window *` `next_window` (const `FL_Window *`)
Returns the next top-level window in the list of `shown()` windows.
- static void `own_colormap` ()
Makes FLTK use its own colormap.
- static void `paste` (`FL_Widget &receiver`)
Backward compatibility only: Set things up so the receiver widget will be called with an `FL_PASTE` event some time in the future for the specified clipboard.
- static void `paste` (`FL_Widget &receiver`, int clipboard)
Pastes the data from the selection (0) or primary (1) clipboard into receiver.
- static void `pushed` (`FL_Widget *`)
Sets the widget that is being pushed.
- static `FL_Widget *` `pushed` ()
Gets the widget that is being pushed.
- static `FL_Widget *` `readqueue` ()
All `FL_Widgets` that don't have a callback defined use a default callback that puts a pointer to the widget in this queue, and this method reads the oldest widget out of this queue.
- static int `ready` ()
This is similar to `Fl::check()` except this does not call `Fl::flush()` or any callbacks, which is useful if your program is in a state where such callbacks are illegal.
- static void `redraw` ()
Redraws all widgets.
- static void `release` ()
Releases the current grabbed window, equals `grab(0)`.

- static void [release_widget_pointer](#) (FL_Widget *&w)
Releases a widget pointer from the watch list.
- static int [reload_scheme](#) ()
Called by scheme according to scheme name.
- static void [remove_check](#) (FL_Timeout_Handler, void *=0)
Removes a check callback.
- static void [remove_fd](#) (int)
Removes a file descriptor handler.
- static void [remove_fd](#) (int, int when)
Removes a file descriptor handler.
- static void [remove_handler](#) (int(*h)(int))
Removes a previously added event handler.
- static void [remove_idle](#) (void(*cb)(void *), void *data=0)
Removes the specified idle callback, if it is installed.
- static void [remove_timeout](#) (FL_Timeout_Handler, void *=0)
Removes a timeout callback.
- static void [repeat_timeout](#) (double t, FL_Timeout_Handler, void *=0)
Repeats a timeout callback from the expiration of the previous timeout, allowing for more accurate timing.
- static int [run](#) ()
As long as any windows are displayed this calls [Fl::wait\(\)](#) repeatedly.
- static const char * [scheme](#) ()
*See void [scheme\(const char *name\)](#).*
- static int [scheme](#) (const char *)
Gets or sets the current widget scheme.
- static int [screen_count](#) ()
Gets the number of available screens.
- static void [screen_xywh](#) (int &X, int &Y, int &W, int &H, int n)
Gets the screen bounding rect for the given screen.
- static void [screen_xywh](#) (int &X, int &Y, int &W, int &H, int mx, int my)
Gets the bounding box of a screen that contains the specified screen position mx, my.
- static void [screen_xywh](#) (int &X, int &Y, int &W, int &H)
Gets the bounding box of a screen that contains the mouse pointer.
- static void [scrollbar_size](#) (int W)

Sets the default scrollbar size that is used by the [Fl_Browser_](#), [Fl_Help_View](#), [Fl_Scroll](#), and [Fl_Text_Display](#) widgets.

- static int [scrollbar_size](#) ()
Gets the default scrollbar size that is used by the [Fl_Browser_](#), [Fl_Help_View](#), [Fl_Scroll](#), and [Fl_Text_Display](#) widgets.
- static void [selection](#) ([Fl_Widget](#) &owner, const char *, int len)
Changes the current selection.
- static void [selection_owner](#) ([Fl_Widget](#) *)
Back-compatibility only: The single-argument call can be used to move the selection to another widget or to set the owner to NULL, without changing the actual text of the selection.
- static [Fl_Widget](#) * [selection_owner](#) ()
back-compatibility only: Gets the widget owning the current selection
- static void [set_abort](#) (void(*f)(const char *,...))
For back compatibility, sets the void [Fl::fatal](#) handler callback.
- static void [set_atclose](#) (void(*f)([Fl_Window](#) *, void *))
For back compatibility, sets the [Fl::atclose](#) handler callback.
- static void [set_boxtype](#) ([Fl_Boxtype](#), [Fl_Boxtype](#) from)
Copies the from boxtype.
- static void [set_boxtype](#) ([Fl_Boxtype](#), [Fl_Box_Draw_F](#) *, [uchar](#), [uchar](#), [uchar](#), [uchar](#))
Sets the function to call to draw a specific boxtype.
- static void [set_color](#) ([Fl_Color](#), unsigned)
Sets an entry in the fl_color index table.
- static void [set_color](#) ([Fl_Color](#), [uchar](#), [uchar](#), [uchar](#))
Sets an entry in the fl_color index table.
- static void [set_font](#) ([Fl_Font](#), [Fl_Font](#))
Copies one face to another.
- static void [set_font](#) ([Fl_Font](#), const char *)
Changes a face.
- static [Fl_Font](#) [set_fonts](#) (const char *==0)
FLTK will open the display, and add every fonts on the server to the face table.
- static void [set_idle](#) (void(*cb)())
Sets an idle callback.
- static void [set_labeltype](#) ([Fl_Labeltype](#), [Fl_Labeltype](#) from)
Sets the functions to call to draw and measure a specific labeltype.

- static void [set_labeltype](#) ([Fl_Labeltype](#), [Fl_Label_Draw_F](#) *, [Fl_Label_Measure_F](#) *)
Sets the functions to call to draw and measure a specific labeltype.
- static int [test_shortcut](#) (int)
Test the current event, which must be an [FL_KEYBOARD](#) or [FL_SHORTCUT](#), against a shortcut value (described in [Fl_Button](#)).
- static void * [thread_message](#) ()
The [thread_message\(\)](#) method returns the last message that was sent from a child by the [awake\(\)](#) method.
- static void [unlock](#) ()
The [unlock\(\)](#) method releases the lock that was set using the [lock\(\)](#) method.
- static double [version](#) ()
Returns the compiled-in value of the [FL_VERSION](#) constant.
- static int [visible_focus](#) ()
Gets or sets the visible keyboard focus on buttons and other non-text widgets.
- static void [visible_focus](#) (int v)
Gets or sets the visible keyboard focus on buttons and other non-text widgets.
- static int [visual](#) (int)
Selects a visual so that your graphics are drawn correctly.
- static int [w](#) ()
Returns the width of the screen in pixels.
- static double [wait](#) (double time)
See int [wait\(\)](#).
- static int [wait](#) ()
Waits until "something happens" and then returns.
- static void [watch_widget_pointer](#) ([Fl_Widget](#) *&w)
Adds a widget pointer to the widget watch list.
- static int [x](#) ()
Returns the origin of the current screen, where 0 indicates the left side of the screen.
- static int [y](#) ()
Returns the origin of the current screen, where 0 indicates the top edge of the screen.

Static Public Attributes

- static void(* [atclose](#))([Fl_Window](#) *, void *) = [default_atclose](#)
Back compatibility: default window callback handler.
- static void(* [error](#))(const char *,...) = ::[error](#)

FLTK calls `Fl::error()` to output a normal error message.

- static void(* `fatal`)(const char *,...) = ::fatal

FLTK calls `Fl::fatal()` to output a fatal error message.

- static const char *const `help` = helpmsg+13

Usage string displayed if `Fl::args()` detects an invalid argument.

- static void(* `idle`)()

The currently executing idle callback function: DO NOT USE THIS DIRECTLY!

- static void(* `warning`)(const char *,...) = ::warning

FLTK calls `Fl::warning()` to output a warning message.

30.1.1 Detailed Description

The `Fl` is the FLTK global (static) containing state information and global methods for the current application.

30.1.2 Member Function Documentation

30.1.2.1 int Fl::add_aware_handler_ (Fl_Awake_Handler func, void *data) [static]

Adds an aware handler for use in `awake()`.

30.1.2.2 void Fl::add_check (Fl_Timeout_Handler cb, void *argp = 0) [static]

FLTK will call this callback just before it flushes the display and waits for events.

This is different than an idle callback because it is only called once, then FLTK calls the system and tells it not to return until an event happens.

This can be used by code that wants to monitor the application's state, such as to keep a display up to date. The advantage of using a check callback is that it is called only when no events are pending. If events are coming in quickly, whole blocks of them will be processed before this is called once. This can save significant time and avoid the application falling behind the events.

Sample code:

```
bool state_changed; // anything that changes the display turns this on

void callback(void*) {
    if (!state_changed) return;
    state_changed = false;
    do_expensive_calculation();
    widget->redraw();
}

main() {
    Fl::add_check(callback);
    return Fl::run();
}
```

30.1.2.3 static void Fl::add_fd (int *fd*, int *when*, void(*)(int, void *) *cb*, void * = 0) [static]

Adds file descriptor *fd* to listen to.

When the *fd* becomes ready for reading [Fl::wait\(\)](#) will call the callback and then return. The callback is passed the *fd* and the arbitrary *void** argument.

The second version takes a *when* bitfield, with the bits `FL_READ`, `FL_WRITE`, and `FL_EXCEPT` defined, to indicate when the callback should be done.

There can only be one callback of each type for a file descriptor. [Fl::remove_fd\(\)](#) gets rid of *all* the callbacks for a given file descriptor.

Under UNIX *any* file descriptor can be monitored (files, devices, pipes, sockets, etc.). Due to limitations in Microsoft Windows, WIN32 applications can only monitor sockets.

30.1.2.4 void Fl::add_idle (void(*)(void *) *cb*, void * *data* = 0) [static]

Adds a callback function that is called every time by [Fl::wait\(\)](#) and also makes it act as though the timeout is zero (this makes [Fl::wait\(\)](#) return immediately, so if it is in a loop it is called repeatedly, and thus the idle function is called repeatedly).

The idle function can be used to get background processing done.

You can have multiple idle callbacks. To remove an idle callback use [Fl::remove_idle\(\)](#).

[Fl::wait\(\)](#) and [Fl::check\(\)](#) call idle callbacks, but [Fl::ready\(\)](#) does not.

The idle callback can call any FLTK functions, including [Fl::wait\(\)](#), [Fl::check\(\)](#), and [Fl::ready\(\)](#).

FLTK will not recursively call the idle callback.

30.1.2.5 void Fl::add_timeout (double *t*, FL_Timeout_Handler *cb*, void * *argp* = 0) [static]

Adds a one-shot timeout callback.

The function will be called by [Fl::wait\(\)](#) at *t* seconds after this function is called. The optional *void** argument is passed to the callback.

You can have multiple timeout callbacks. To remove a timeout callback use [Fl::remove_timeout\(\)](#).

If you need more accurate, repeated timeouts, use [Fl::repeat_timeout\(\)](#) to reschedule the subsequent timeouts.

The following code will print "TICK" each second on stdout with a fair degree of accuracy:

```
void callback(void*) {
    puts("TICK");
    Fl::repeat_timeout(1.0, callback);
}

int main() {
    Fl::add_timeout(1.0, callback);
    return Fl::run();
}
```

30.1.2.6 int Fl::arg (int *argc*, char ** *argv*, int & *i*) [static]

Consume a single switch from *argv*, starting at word *i*.

Returns the number of words eaten (1 or 2, or 0 if it is not recognized) and adds the same value to *i*. You can use this function if you prefer to control the incrementing through the arguments yourself.

30.1.2.7 `int Fl::args (int argc, char ** argv, int & i, int(*) (int, char **, int &) cb = 0) [static]`

Consume all switches from *argv*.

Returns number of words eaten Returns zero on error. '*i*' will either point at first word that does not start with '-', at the error word, or after a '-', or at *argc*. If your program does not take any word arguments you can report an error if *i* < *argc*.

FLTK provides an *entirely optional* command-line switch parser. You don't have to call it if you don't like them! Everything it can do can be done with other calls to FLTK.

To use the switch parser, call `Fl::args(...)` near the start of your program. This does *not* open the display, instead switches that need the display open are stashed into static variables. Then you *must* display your first window by calling `window->show(argc,argv)`, which will do anything stored in the static variables.

callback lets you define your own switches. It is called with the same *argc* and *argv*, and with *i* the index of each word. The callback should return zero if the switch is unrecognized, and not change *i*. It should return non-zero if the switch is recognized, and add at least 1 to *i* (it can add more to consume words after the switch). This function is called *before* any other tests, so *you can override any FLTK switch* (this is why FLTK can use very short switches instead of the long ones all other toolkits force you to use).

On return *i* is set to the index of the first non-switch. This is either:

- The first word that does not start with '-'.
- The word '-' (used by many programs to name stdin as a file)
- The first unrecognized switch (return value is 0).
- *argc*

The return value is *i* unless an unrecognized switch is found, in which case it is zero. If your program takes no arguments other than switches you should produce an error if the return value is less than *argc*.

All switches except -bg2 may be abbreviated one letter and case is ignored:

- -bg color or -background color
Sets the background color using `Fl::background()`.
- -bg2 color or -background2 color
Sets the secondary background color using `Fl::background2()`.
- -display host:n.n
Sets the X display to use; this option is silently ignored under WIN32 and MacOS.
- -dnd and -nodnd
Enables or disables drag and drop text operations using `Fl::dnd_text_ops()`.
- -fg color or -foreground color
Sets the foreground color using `Fl::foreground()`.
- -geometry WxH+X+Y
Sets the initial window position and size according the the standard X geometry string.

- `-iconic`
Iconifies the window using [Fl_Window::iconize\(\)](#).
- `-kbd` and `-nokbd`
Enables or disables visible keyboard focus for non-text widgets using [Fl::visible_focus\(\)](#).
- `-name string`
Sets the window class using [Fl_Window::xclass\(\)](#).
- `-scheme string`
Sets the widget scheme using [Fl::scheme\(\)](#).
- `-title string`
Sets the window title using [Fl_Window::label\(\)](#).
- `-tooltips` and `-notooltips`
Enables or disables tooltips using [Fl_Tooltip::enable\(\)](#).

The second form of [Fl::args\(\)](#) is useful if your program does not have command line switches of its own. It parses all the switches, and if any are not recognized it calls `Fl::abort(Fl::help)`.

A usage string is displayed if [Fl::args\(\)](#) detects an invalid argument on the command-line. You can change the message by setting the [Fl::help](#) pointer.

30.1.2.8 `void Fl::background (uchar r, uchar g, uchar b)` [static]

Changes `fl_color(FL_BACKGROUND_COLOR)` to the given color, and changes the gray ramp from 32 to 56 to black to white.

These are the colors used as backgrounds by almost all widgets and used to draw the edges of all the boxtypes.

30.1.2.9 `void Fl::background2 (uchar r, uchar g, uchar b)` [static]

Changes the alternative background color.

This color is used as a background by [Fl_Input](#) and other text widgets.

This call may change `fl_color(FL_FOREGROUND_COLOR)` if it does not provide sufficient contrast to `FL_BACKGROUND2_COLOR`.

30.1.2.10 `int Fl::box_dh (Fl_Boxtype t)` [static]

Returns the height offset for the given boxtype.

See also:

[box_dy\(\)](#).

30.1.2.11 `int Fl::box_dw (Fl_Boxtype t)` [static]

Returns the width offset for the given boxtype.

See also:

[box_dy\(\)](#).

30.1.2.12 `int Fl::box_dx (Fl_Boxtype t)` [static]

Returns the X offset for the given boxtype.

See also:

[box_dy\(\)](#)

30.1.2.13 `int Fl::box_dy (Fl_Boxtype t)` [static]

Returns the Y offset for the given boxtype.

These functions return the offset values necessary for a given boxtype, useful for computing the area inside a box's borders, to prevent overdrawing the borders.

For instance, in the case of a boxtype like FL_DOWN_BOX where the border width might be 2 pixels all around, the above functions would return 2, 2, 4, and 4 for box_dx, box_dy, box_dw, and box_dh respectively.

An example to compute the area inside a widget's box():

```
int X = yourwidget->x() + Fl::box_dx(yourwidget->box());
int Y = yourwidget->y() + Fl::box_dy(yourwidget->box());
int W = yourwidget->w() - Fl::box_dw(yourwidget->box());
int H = yourwidget->h() - Fl::box_dh(yourwidget->box());
```

These functions are mainly useful in the draw() code for deriving custom widgets, where one wants to avoid drawing over the widget's own border box().

30.1.2.14 `int Fl::check ()` [static]

Same as Fl::wait(0).

Calling this during a big calculation will keep the screen up to date and the interface responsive:

```
while (!calculation_done()) {
    calculate();
    Fl::check();
    if (user_hit_abort_button()) break;
}
```

The returns non-zero if any windows are displayed, and 0 if no windows are displayed (this is likely to change in future versions of FLTK).

30.1.2.15 `static int Fl::damage ()` [inline, static]

If true then [flush\(\)](#) will do something.

30.1.2.16 void Fl::display (const char * *d*) [static]

Sets the X display to use for all windows.

Actually this just sets the environment variable \$DISPLAY to the passed string, so this only works before you show() the first window or otherwise open the display, and does nothing useful under WIN32.

30.1.2.17 static int Fl::dnd_text_ops () [inline, static]

Gets or sets whether drag and drop text operations are supported.

This specifically affects whether selected text can be dragged from text fields or dragged within a text field as a cut/paste shortcut.

30.1.2.18 static void Fl::dnd_text_ops (int *v*) [inline, static]

Gets or sets whether drag and drop text operations are supported.

This specifically affects whether selected text can be dragged from text fields or dragged within a text field as a cut/paste shortcut.

30.1.2.19 int Fl::draw_box_active () [static]

Determines if the current draw box is active or inactive.

If inactive, the box color is changed by the inactive color.

30.1.2.20 void Fl::flush () [static]

Causes all the windows that need it to be redrawn and graphics forced out through the pipes.

This is what [wait\(\)](#) does before looking for events.

30.1.2.21 void Fl::foreground (uchar *r*, uchar *g*, uchar *b*) [static]

Changes fl_color(FL_FOREGROUND_COLOR).

30.1.2.22 int Fl::get_awake_handler_ (Fl_Awake_Handler & *func*, void *& *data*) [static]

Gets the last stored awake handler for use in [awake\(\)](#).

30.1.2.23 Fl_Box_Draw_F * Fl::get_boxtype (Fl_Boxtype *t*) [static]

Gets the current box drawing function for the specified box type.

30.1.2.24 void Fl::get_system_colors () [static]

Read the user preference colors from the system and use them to call [Fl::foreground\(\)](#), [Fl::background\(\)](#), and [Fl::background2\(\)](#).

This is done by Fl_Window::show(argc,argv) before applying the -fg and -bg switches.

On X this reads some common values from the Xdefaults database. KDE users can set these values by running the "krdp" program, and newer versions of KDE set this automatically if you check the "apply style to other X programs" switch in their control panel.

30.1.2.25 `int Fl::gl_visual (int mode, int * alist = 0)` [static]

This does the same thing as [Fl::visual\(int\)](#) but also requires OpenGL drawing to work.

This *must* be done if you want to draw in normal windows with OpenGL with `gl_start()` and `gl_end()`. It may be useful to call this so your X windows use the same visual as an [Fl_Gl_Window](#), which on some servers will reduce colormap flashing.

See [Fl_Gl_Window](#) for a list of additional values for the argument.

30.1.2.26 `void Fl::own_colormap ()` [static]

Makes FLTK use its own colormap.

This may make FLTK display better and will reduce conflicts with other programs that want lots of colors. However the colors may flash as you move the cursor between windows.

This does nothing if the current visual is not colormapped.

30.1.2.27 `int Fl::ready ()` [static]

This is similar to [Fl::check\(\)](#) except this does *not* call [Fl::flush\(\)](#) or any callbacks, which is useful if your program is in a state where such callbacks are illegal.

This returns true if [Fl::check\(\)](#) would do anything (it will continue to return true until you call [Fl::check\(\)](#) or [Fl::wait\(\)](#)).

```
while (!calculation_done()) {
    calculate();
    if (Fl::ready()) {
        do_expensive_cleanup();
        Fl::check();
        if (user_hit_abort_button()) break;
    }
}
```

30.1.2.28 `static void Fl::release ()` [inline, static]

Releases the current grabbed window, equals `grab(0)`.

Deprecated

Use `Fl::grab(0)` instead.

See also:

`Fl_Window*` [grab\(\)](#)

30.1.2.29 `int Fl::reload_scheme ()` [static]

Called by scheme according to scheme name.

Loads or reloads the current scheme selection. See void [scheme\(const char *name\)](#)

30.1.2.30 `void Fl::remove_check (Fl_Timeout_Handler cb, void *argp = 0)` [static]

Removes a check callback.

It is harmless to remove a check callback that no longer exists.

30.1.2.31 `static void Fl::remove_fd (int)` [static]

Removes a file descriptor handler.

30.1.2.32 `static void Fl::remove_fd (int, int when)` [static]

Removes a file descriptor handler.

30.1.2.33 `void Fl::remove_timeout (Fl_Timeout_Handler cb, void *argp = 0)` [static]

Removes a timeout callback.

It is harmless to remove a timeout callback that no longer exists.

30.1.2.34 `void Fl::repeat_timeout (double t, Fl_Timeout_Handler cb, void *argp = 0)`
[static]

Repeats a timeout callback from the expiration of the previous timeout, allowing for more accurate timing.

You may only call this method inside a timeout callback.

The following code will print "TICK" each second on stdout with a fair degree of accuracy:

```
void callback(void*) {
    puts("TICK");
    Fl::repeat_timeout(1.0, callback);
}

int main() {
    Fl::add_timeout(1.0, callback);
    return Fl::run();
}
```

30.1.2.35 `int Fl::run ()` [static]

As long as any windows are displayed this calls [Fl::wait\(\)](#) repeatedly.

When all the windows are closed it returns zero (supposedly it would return non-zero on any errors, but FLTK calls exit directly for these). A normal program will end main() with return [Fl::run\(\)](#);

30.1.2.36 int Fl::scheme (const char *s) [static]

Gets or sets the current widget scheme.

NULL will use the scheme defined in the FLTK_SCHEME environment variable or the scheme resource under X11. Otherwise, any of the following schemes can be used:

- "none" - This is the default look-n-feel which resembles old Windows (95/98/Me/NT/2000) and old GTK/KDE
- "plastic" - This scheme is inspired by the Aqua user interface on Mac OS X
- "gtk+" - This scheme is inspired by the Red Hat Bluecurve theme

30.1.2.37 void Fl::set_boxtype (Fl_Boxtype to, Fl_Boxtype from) [static]

Copies the from boxtype.

30.1.2.38 void Fl::set_boxtype (Fl_Boxtype t, Fl_Box_Draw_F *f, uchar a, uchar b, uchar c, uchar d) [static]

Sets the function to call to draw a specific boxtype.

30.1.2.39 static void Fl::set_idle (void(*)() cb) [inline, static]

Sets an idle callback.

Deprecated

This method is obsolete - use the [add_idle\(\)](#) method instead.

30.1.2.40 static void Fl::set_labeltype (Fl_Labeltype, Fl_Labeltype from) [static]

Sets the functions to call to draw and measure a specific labeltype.

30.1.2.41 void Fl::set_labeltype (Fl_Labeltype t, Fl_Label_Draw_F *f, Fl_Label_Measure_F *m) [static]

Sets the functions to call to draw and measure a specific labeltype.

30.1.2.42 double Fl::version () [static]

Returns the compiled-in value of the FL_VERSION constant.

This is useful for checking the version of a shared library.

30.1.2.43 static int Fl::visible_focus () [inline, static]

Gets or sets the visible keyboard focus on buttons and other non-text widgets.

The default mode is to enable keyboard focus for all widgets.

30.1.2.44 static void Fl::visible_focus (int v) [inline, static]

Gets or sets the visible keyboard focus on buttons and other non-text widgets.

The default mode is to enable keyboard focus for all widgets.

30.1.2.45 int Fl::visual (int flags) [static]

Selects a visual so that your graphics are drawn correctly.

This is only allowed before you call show() on any windows. This does nothing if the default visual satisfies the capabilities, or if no visual satisfies the capabilities, or on systems that don't have such brain-dead notions.

Only the following combinations do anything useful:

- Fl::visual(FL_RGB)
Full/true color (if there are several depths FLTK chooses the largest). Do this if you use fl_draw_image for much better (non-dithered) output.
- Fl::visual(FL_RGB8)
Full color with at least 24 bits of color. FL_RGB will always pick this if available, but if not it will happily return a less-than-24 bit deep visual. This call fails if 24 bits are not available.
- [Fl::visual](#)(FL_DOUBLE|FL_INDEX)
Hardware double buffering. Call this if you are going to use [Fl_Double_Window](#).
- [Fl::visual](#)(FL_DOUBLE|FL_RGB)
- [Fl::visual](#)(FL_DOUBLE|FL_RGB8)
Hardware double buffering and full color.

This returns true if the system has the capabilities by default or FLTK succeeded in turning them on. Your program will still work even if this returns false (it just won't look as good).

30.1.2.46 int Fl::wait () [static]

Waits until "something happens" and then returns.

Call this repeatedly to "run" your program. You can also check what happened each time after this returns, which is quite useful for managing program state.

What this really does is call all idle callbacks, all elapsed timeouts, call [Fl::flush\(\)](#) to get the screen to update, and then wait some time (zero if there are idle callbacks, the shortest of all pending timeouts, or infinity), for any events from the user or any [Fl::add_fd\(\)](#) callbacks. It then handles the events and calls the callbacks and then returns.

The return value of the first form is non-zero if there are any visible windows - this may change in future versions of FLTK.

The second form waits a maximum of *time* seconds. *It can return much sooner if something happens.*

The return value is positive if an event or fd happens before the time elapsed. It is zero if nothing happens (on Win32 this will only return zero if *time* is zero). It is negative if an error occurs (this will happen on UNIX if a signal happens).

30.1.3 Member Data Documentation

30.1.3.1 `const char *const Fl::help = helpmsg+13` [static]

Usage string displayed if [Fl::args\(\)](#) detects an invalid argument.

This may be changed to point to customized text at run-time.

30.1.3.2 `void(* Fl::idle)() ()` [static]

The currently executing idle callback function: DO NOT USE THIS DIRECTLY!

This is now used as part of a higher level system allowing multiple idle callback functions to be called.

See also:

[add_idle\(\)](#), [remove_idle\(\)](#)

The documentation for this class was generated from the following files:

- Fl.H
- Fl.cxx
- Fl_abort.cxx
- Fl_add_idle.cxx
- Fl_arg.cxx
- [fl_boxtype.cxx](#)
- Fl_Browser_.cxx
- [fl_color.cxx](#)
- fl_color_mac.cxx
- fl_color_win32.cxx
- Fl_compose.cxx
- Fl_display.cxx
- fl_dnd_mac.cxx
- fl_dnd_win32.cxx
- fl_dnd_x.cxx
- Fl_get_key.cxx
- Fl_get_key_mac.cxx
- Fl_get_key_win32.cxx
- Fl_get_system_colors.cxx
- Fl_grab.cxx
- fl_labeltype.cxx
- Fl_lock.cxx
- Fl_own_colormap.cxx
- fl_set_font.cxx
- fl_set_fonts_mac.cxx
- fl_set_fonts_win32.cxx
- fl_set_fonts_x.cxx

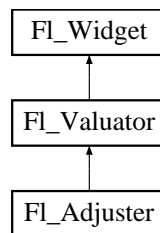
- `fl_set_fonts_xft.cxx`
- `fl_shortcut.cxx`
- `Fl_visual.cxx`
- `Fl_Widget.cxx`
- `Fl_Window.cxx`
- `gl_start.cxx`
- `screen_xywh.cxx`

30.2 Fl_Adjuster Class Reference

The [Fl_Adjuster](#) widget was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range.

```
#include <Fl_Adjuster.H>
```

Inheritance diagram for Fl_Adjuster::



Public Member Functions

- [Fl_Adjuster](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Adjuster](#) widget using the given position, size, and label string.
- int [soft](#) () const
If "soft" is turned on, the user is allowed to drag the value outside the range.
- void [soft](#) (int s)
If "soft" is turned on, the user is allowed to drag the value outside the range.

Protected Member Functions

- void [draw](#) ()
Draws the widget.
- int [handle](#) (int)
Handles the specified event.
- void [value_damage](#) ()
Asks for partial redraw.

30.2.1 Detailed Description

The [Fl_Adjuster](#) widget was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range.

Figure 30.1: Fl_Adjuster

When you press a button and drag to the right the value increases. When you drag to the left it decreases. The largest button adjusts by $100 * \text{step}()$, the next by $10 * \text{step}()$ and that smallest button by $\text{step}()$. Clicking on the buttons increments by 10 times the amount dragging by a pixel does. Shift + click decrements by 10 times the amount.

30.2.2 Constructor & Destructor Documentation

30.2.2.1 Fl_Adjuster::Fl_Adjuster (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [Fl_Adjuster](#) widget using the given position, size, and label string.

It looks best if one of the dimensions is 3 times the other.

Inherited destructor destroys the Valuator.

30.2.3 Member Function Documentation

30.2.3.1 void Fl_Adjuster::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.2.3.2 int Fl_Adjuster::handle (int *event*) [protected, virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

30.2.3.3 int FL_Adjuster::soft () const [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

30.2.3.4 void FL_Adjuster::soft (int s) [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

The documentation for this class was generated from the following files:

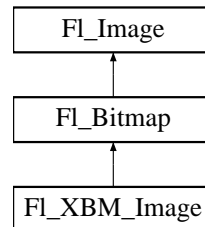
- FL_Adjuster.H
- FL_Adjuster.cxx

30.3 FL_Bitmap Class Reference

The [FL_Bitmap](#) class supports caching and drawing of mono-color (bitmap) images.

```
#include <FL_Bitmap.H>
```

Inheritance diagram for FL_Bitmap::



Public Member Functions

- [FL_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [FL_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx=0, int cy=0)
The [draw\(\)](#) methods draw the image.
- [FL_Bitmap](#) (const char *bits, int W, int H)
The constructors create a new bitmap from the specified bitmap data.
- [FL_Bitmap](#) (const [uchar](#) *bits, int W, int H)
The constructors create a new bitmap from the specified bitmap data.
- virtual void [label](#) ([FL_Menu_Item](#) *m)
The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void [label](#) ([FL_Widget](#) *w)
The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void [uncache](#) ()
If the image has been cached for display, delete the cache data.
- virtual [~FL_Bitmap](#) ()
The destructor free all memory and server resources that are used by the bitmap.

Public Attributes

- int `alloc_array`
Non-zero if array points to bitmap data allocated internally.
- const uchar * `array`
pointer to raw bitmap data
- unsigned `id`
for internal use

30.3.1 Detailed Description

The `FL_Bitmap` class supports caching and drawing of mono-color (bitmap) images. Images are drawn using the current color.

30.3.2 Constructor & Destructor Documentation

30.3.2.1 `FL_Bitmap::FL_Bitmap (const uchar * bits, int W, int H)` [inline]

The constructors create a new bitmap from the specified bitmap data.

30.3.2.2 `FL_Bitmap::FL_Bitmap (const char * array, int W, int H)` [inline]

The constructors create a new bitmap from the specified bitmap data.

30.3.3 Member Function Documentation

30.3.3.1 `FL_Image* FL_Bitmap::copy ()` [inline]

The `copy()` method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of `FL_Shared_Image`, released) when you are done with it.

Reimplemented from `FL_Image`.

30.3.3.2 `FL_Image * FL_Bitmap::copy (int W, int H)` [virtual]

The `copy()` method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of `FL_Shared_Image`, released) when you are done with it.

Reimplemented from `FL_Image`.

30.3.3.3 void Fl_Bitmap::draw (int X, int Y) [inline]

The [draw\(\)](#) methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented from [Fl_Image](#).

30.3.3.4 void Fl_Bitmap::draw (int X, int Y, int W, int H, int cx = 0, int cy = 0) [virtual]

The [draw\(\)](#) methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the cx and cy arguments.

Reimplemented from [Fl_Image](#).

30.3.3.5 void Fl_Bitmap::label (Fl_Menu_Item * m) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.

Use the [image\(\)](#) or [deimage\(\)](#) methods of the [Fl_Widget](#) and [Fl_Menu_Item](#) classes instead.

Reimplemented from [Fl_Image](#).

30.3.3.6 void Fl_Bitmap::label (Fl_Widget * widget) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.

Use the [image\(\)](#) or [deimage\(\)](#) methods of the [Fl_Widget](#) and [Fl_Menu_Item](#) classes instead.

Reimplemented from [Fl_Image](#).

30.3.3.7 void Fl_Bitmap::uncache () [virtual]

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from [Fl_Image](#).

The documentation for this class was generated from the following files:

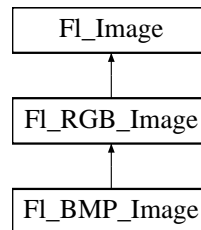
- [Fl_Bitmap.H](#)
- [Fl_Bitmap.cxx](#)

30.4 Fl_BMP_Image Class Reference

The [Fl_BMP_Image](#) class supports loading, caching, and drawing of Windows Bitmap (BMP) image files.

```
#include <Fl_BMP_Image.H>
```

Inheritance diagram for Fl_BMP_Image::



Public Member Functions

- [Fl_BMP_Image](#) (const char *filename)

The constructor loads the named BMP image from the given bmp filename.

30.4.1 Detailed Description

The [Fl_BMP_Image](#) class supports loading, caching, and drawing of Windows Bitmap (BMP) image files.

30.4.2 Constructor & Destructor Documentation

30.4.2.1 Fl_BMP_Image::Fl_BMP_Image (const char * bmp)

The constructor loads the named BMP image from the given bmp filename.

The inherited destructor free all memory and server resources that are used by the image.

The destructor free all memory and server resources that are used by the image

The documentation for this class was generated from the following files:

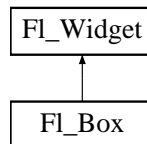
- Fl_BMP_Image.H
- Fl_BMP_Image.cxx

30.5 Fl_Box Class Reference

This widget simply draws its box, and possibly it's label.

```
#include <Fl_Box.H>
```

Inheritance diagram for Fl_Box::



Public Member Functions

- [Fl_Box](#) ([Fl_Boxtype](#) b, int X, int Y, int W, int H, const char *l)
*See [Fl_Box::Fl_Box\(int x, int y, int w, int h, const char * = 0\)](#).*
- [Fl_Box](#) (int X, int Y, int W, int H, const char *l=0)
 - The first constructor sets [box\(\)](#) to `FL_NO_BOX`, which means it is invisible.
- virtual int [handle](#) (int)
Handles the specified event.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.5.1 Detailed Description

This widget simply draws its box, and possibly it's label.

Putting it before some other widgets and making it big enough to surround them will let you draw a frame around them.

30.5.2 Constructor & Destructor Documentation

30.5.2.1 [Fl_Box::Fl_Box \(int X, int Y, int W, int H, const char *l = 0\)](#) [inline]

- The first constructor sets [box\(\)](#) to `FL_NO_BOX`, which means it is invisible.

However such widgets are useful as placeholders or [Fl_Group::resizable\(\)](#) values. To change the box to something visible, use [box\(n\)](#).

- The second form of the constructor sets the box to the specified box type.

The destructor removes the box.

30.5.3 Member Function Documentation

30.5.3.1 void Fl_Box::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.5.3.2 int Fl_Box::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

The documentation for this class was generated from the following files:

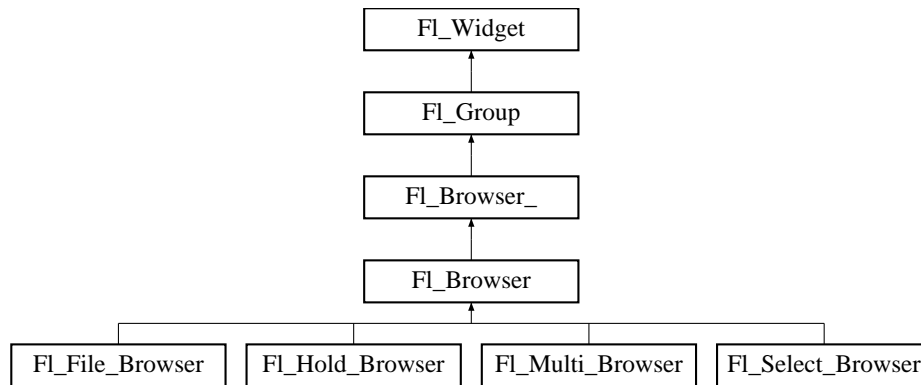
- `Fl_Box.H`
- `Fl_Box.cxx`

30.6 FL_Browser Class Reference

The [FL_Browser](#) widget displays a scrolling list of text lines, and manages all the storage for the text.

```
#include <Fl_Browser.H>
```

Inheritance diagram for FL_Browser::



Public Types

- enum [FL_Line_Position](#) { **TOP**, **BOTTOM**, **MIDDLE** }
For internal use only?

Public Member Functions

- void [add](#) (const char *newtext, void *d=0)
Adds a new line to the end of the browser.
- void [bottomline](#) (int line)
Scrolls the browser so the bottom item in the browser is showing the specified line.
- void [clear](#) ()
Removes all the lines in the browser.
- void [column_char](#) (char c)
Sets the column separator to c.
- char [column_char](#) () const
Gets the current column separator character.
- void [column_widths](#) (const int *arr)
Sets the current array to arr.
- const int * [column_widths](#) () const
Gets the current column width array.

- void [data](#) (int line, void *d)
Sets the user data for specified line to d.
- void * [data](#) (int line) const
Returns the user [data\(\)](#) for specified line.
- void [display](#) (int line, int val=1)
For back compatibility.
- int [displayed](#) (int line) const
Returns non-zero if line is visible.
- [Fl_Browser](#) (int X, int Y, int W, int H, const char *L=0)
The constructor makes an empty browser.
- void [format_char](#) (char c)
Sets the current format code prefix character to c.
- char [format_char](#) () const
Gets the current format code prefix character, which by default is '@'.
- void [hide](#) ()
Hides the entire [Fl_Browser](#) widget – opposite of [show\(\)](#).
- void [hide](#) (int line)
Makes line invisible, preventing selection by the user.
- void [insert](#) (int line, const char *newtext, void *d=0)
Insert a new entry whose label is newtext above given line, optional data d.
- void [lineposition](#) (int line, [Fl_Line_Position](#) pos)
Updates the browser so that line is shown at position pos.
- int [load](#) (const char *filename)
Clears the browser and reads the file, adding each line from the file to the browser.
- void [make_visible](#) (int line)
Make the item at the specified line [visible\(\)](#).
- void [middleline](#) (int line)
Scrolls the browser so the middle item in the browser is showing the specified line.
- void [move](#) (int to, int from)
Line from is removed and reinserted at to.
- void [remove](#) (int line)
Remove entry for given line number, making the browser one line shorter.
- void [replace](#) (int a, const char *b)
For back compatibility only.

- `int select (int line, int val=1)`
Sets the selection state of the item at `line` to the value `val`.
- `int selected (int line) const`
Returns 1 if specified `line` is selected, 0 if not.
- `void show ()`
Shows the entire `Fl_Browser` widget – opposite of `hide()`.
- `void show (int line)`
Makes `line` visible, and available for selection by user.
- `void size (int W, int H)`
Changes the size of the widget.
- `int size () const`
Returns how many lines are in the browser.
- `void swap (int a, int b)`
Swaps two browser lines `a` and `b`.
- `void text (int line, const char *newtext)`
Sets the text for the specified `line` to `newtext`.
- `const char * text (int line) const`
Returns the label text for the specified `line`.
- `void topline (int line)`
Scrolls the browser so the top item in the browser is showing the specified `line`.
- `int topline () const`
Returns the line that is currently visible at the top of the browser.
- `void value (int line)`
Sets the browser's `value()`, which selects the specified `line`.
- `int value () const`
Returns the line number of the currently selected line, or 0 if none.
- `int visible (int line) const`
Returns non-zero if the specified `line` is visible, 0 if hidden.
- `~Fl_Browser ()`
The destructor deletes all list items and destroys the browser.

Protected Member Functions

- `FL_BLINE * _remove (int line)`
Removes the item at the specified line.
- `FL_BLINE * find_line (int line) const`
Returns the item for specified line.
- `int full_height () const`
*The height of the entire list of all *visible()* items in pixels.*
- `int incr_height () const`
The default height of items (including spacing in-between) in pixels.
- `void insert (int line, FL_BLINE *item)`
Insert specified item above line.
- `void * item_at (int line) const`
Return the item at specified line.
- `void item_draw (void *item, int X, int Y, int W, int H) const`
Draws item at the position specified by X Y W H.
- `void * item_first () const`
Returns the very first item in the list.
- `int item_height (void *item) const`
Returns height of item in pixels.
- `void * item_last () const`
Returns the very last item in the list.
- `void * item_next (void *item) const`
Returns the next item after item.
- `void * item_prev (void *item) const`
Returns the previous item before item.
- `void item_select (void *item, int val)`
Change the selection state of item to the value val.
- `int item_selected (void *item) const`
See if item is selected.
- `void item_swap (void *a, void *b)`
Swap the items a and b.
- `const char * item_text (void *item) const`
Returns the label text for item.

- int [item_width](#) (void *item) const
Returns width of item in pixels.
- int [lineno](#) (void *item) const
Returns line number corresponding to item, or zero if not found.
- void [swap](#) (FL_BLINE *a, FL_BLINE *b)
Swap the two items a and b.

30.6.1 Detailed Description

The [FL_Browser](#) widget displays a scrolling list of text lines, and manages all the storage for the text.

This is not a text editor or spreadsheet! But it is useful for showing a vertical list of named objects to the user.

Each line in the browser is identified by number. *The numbers start at one* (this is so that zero can be reserved for "no line" in the selective browsers). *Unless otherwise noted, the methods do not check to see if the passed line number is in range and legal. It must always be greater than zero and \leq [size\(\)](#).*

Each line contains a null-terminated string of text and a void * data pointer. The text string is displayed, the void * pointer can be used by the callbacks to reference the object the text describes.

The base class does nothing when the user clicks on it. The subclasses [FL_Select_Browser](#), [FL_Hold_Browser](#), and [FL_Multi_Browser](#) react to user clicks to select lines in the browser and do callbacks.

The base class [FL_Browser_](#) provides the scrolling and selection mechanisms of this and all the subclasses, but the dimensions and appearance of each item are determined by the subclass. You can use [FL_Browser_](#) to display information other than text, or text that is dynamically produced from your own data structures. If you find that loading the browser is a lot of work or is inefficient, you may want to make a subclass of [FL_Browser_](#).

Some common coding patterns used for working with [FL_Browser](#):

```
// How to loop through all the items in the browser
for ( int t=1; t<=browser->size(); t++ ) {           // index 1 based..!
    printf("item #%d, label='%s'\n", t, browser->text(t));
}
```

Note: If you are *subclassing* [FL_Browser](#), it's more efficient to use the protected methods [item_first\(\)](#) and [item_next\(\)](#), since [FL_Browser](#) internally uses linked lists to manage the browser's items. For more info, see [find_item\(int\)](#).

30.6.2 Constructor & Destructor Documentation

30.6.2.1 [FL_Browser::FL_Browser](#) (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

The constructor makes an empty browser.

Parameters:

- ← *X,Y,W,H* position and size.
- ← *L* label string, may be NULL.

30.6.2.2 FL_Browser::~~FL_Browser () [inline]

The destructor deletes all list items and destroys the browser.

30.6.3 Member Function Documentation

30.6.3.1 FL_BLINE * FL_Browser::_remove (int *line*) [protected]

Removes the item at the specified *line*.

Caveat: See efficiency note in [find_line\(\)](#). You must call [redraw\(\)](#) to make any changes visible.

Parameters:

← *line* The line number to be removed. (1 based) Must be in range!

Returns:

Pointer to browser item that was removed (and is no longer valid).

30.6.3.2 void FL_Browser::add (const char * *newtext*, void * *d* = 0)

Adds a new line to the end of the browser.

The text string *newtext* may contain format characters; see [format_char\(\)](#) for details. *newtext* is copied using the `strdup()` function, and can be NULL to make a blank line.

The optional void * argument *d* will be the [data\(\)](#) for the new item.

Parameters:

← *newtext* The label text used for the added item

← *d* Optional user [data\(\)](#) for the item (0 if unspecified)

30.6.3.3 void FL_Browser::bottomline (int *line*) [inline]

Scrolls the browser so the bottom item in the browser is showing the specified *line*.

30.6.3.4 void FL_Browser::column_char (char *c*) [inline]

Sets the column separator to *c*.

This will only have an effect if you also set [column_widths\(\)](#). The default is `'\t'` (tab).

30.6.3.5 char FL_Browser::column_char () const [inline]

Gets the current column separator character.

The default is `'\t'` (tab).

30.6.3.6 void Fl_Browser::column_widths (const int * *arr*) [inline]

Sets the current array to *arr*.

Make sure the last entry is zero.

See also:

```
const int *Fl_Browsercolumn_widths() const
```

30.6.3.7 const int* Fl_Browser::column_widths () const [inline]

Gets the current column width array.

This array is zero-terminated and specifies the widths in pixels of each column. The text is split at each [column_char\(\)](#) and each part is formatted into it's own column. After the last column any remaining text is formatted into the space between the last column and the right edge of the browser, even if the text contains instances of [column_char\(\)](#). The default value is a one-element array of just a zero, which means there are no columns.

Example:

```
Fl_Browser *b = new Fl_Browser(..);
int widths[] = { 50, 50, 50, 70, 70, 40, 40, 70, 70, 50, 0 }; // widths for each column
b->column_widths(widths); // assign array to widget
b->column_char('\t');      // use tab as the column character
b->add("USER\tPID\tCPU\tMEM\tVSZ\tRSS\tTTY\tSTAT\tSTART\tTIME\tCOMMAND");
b->add("root\t2888\t0.0\t0.0\t1352\t0\ttty3\tSW\tAug15\t0:00\tb@f/sbin/mingetty tty3");
b->add("root\t13115\t0.0\t0.0\t1352\t0\ttty2\tSW\tAug30\t0:00\tb@f/sbin/mingetty tty2");
[...]
```

30.6.3.8 void Fl_Browser::data (int *line*, void * *d*)

Sets the user data for specified *line* to *d*.

Does nothing if *line* is out of range.

Parameters:

← *line* The line of the item whose [data\(\)](#) is to be changed. (1 based)

← *d* The new data to be assigned to the item. (can be NULL)

30.6.3.9 void * Fl_Browser::data (int *line*) const

Returns the user [data\(\)](#) for specified *line*.

Return value can be NULL if *line* is out of range or no user [data\(\)](#) was defined. The parameter *line* is 1 based (1 will be the first item in the list).

Parameters:

← *line* The line number of the item whose [data\(\)](#) is returned. (1 based)

Returns:

The user data pointer (can be NULL)

30.6.3.10 void FL_Browser::display (int *line*, int *val* = 1)

For back compatibility.

This calls show(*line*) if *val* is true, and hide(*line*) otherwise. If *val* is not specified, the default is 1 (makes the line visible).

See also:

[show\(int line\)](#), [hide\(int line\)](#)

30.6.3.11 FL_BLINK * FL_Browser::find_line (int *line*) const [protected]

Returns the item for specified *line*.

Note: This call is slow. It's fine for e.g. responding to user clicks, but slow if called often, such as in a tight sorting loop. Finding an item 'by line' involves a linear lookup on the internal linked list. The performance hit can be significant if the browser's contents is large, and the method is called often (e.g. during a sort). If you're writing a subclass, use the protected methods [item_first\(\)](#), [item_next\(\)](#), etc. to access the internal linked list more efficiently.

Parameters:

← *line* The line number of the item to return. (1 based)

Returns:

The returned item.

30.6.3.12 void FL_Browser::format_char (char *c*) [inline]

Sets the current format code prefix character to *c*.

The default prefix is '@'. Set the prefix to 0 to disable formatting.

See also:

[uchar FL_Browser::format_char\(\) const](#)

30.6.3.13 char FL_Browser::format_char () const [inline]

Gets the current format code prefix character, which by default is '@'.

A string of formatting codes at the start of each column are stripped off and used to modify how the rest of the line is printed:

- '@.' Print rest of line, don't look for more '@' signs
- '@@' Print rest of line starting with '@'
- '@l' Use a LARGE (24 point) font
- '@m' Use a medium large (18 point) font
- '@s' Use a small (11 point) font

- '@b' Use a **bold** font (adds FL_BOLD to font)
- '@i' Use an *italic* font (adds FL_ITALIC to font)
- '@f' or '@t' Use a fixed-pitch font (sets font to FL_COURIER)
- '@c' Center the line horizontally
- '@r' Right-justify the text
- '@B0', '@B1', ... '@B255' Fill the background with fl_color(n)
- '@C0', '@C1', ... '@C255' Use fl_color(n) to draw the text
- '@F0', '@F1', ... Use fl_font(n) to draw the text
- '@S1', '@S2', ... Use point size n to draw the text
- '@u' or '@_' Underline the text.
- '@-' draw an engraved line through the middle.

Notice that the '@.' command can be used to reliably terminate the parsing. To print a random string in a random color, use `printf("\nC%d\@.%s", color, string)` and it will work even if the string starts with a digit or has the format character in it.

30.6.3.14 int FL_Browser::full_height() const [protected, virtual]

The height of the entire list of all [visible\(\)](#) items in pixels.

This returns the accumulated height of *all* the items in the browser that are not hidden with [hide\(\)](#), including items scrolled off screen.

Returns:

The accumulated size of all the visible items in pixels.

Reimplemented from [FL_Browser_](#).

30.6.3.15 void FL_Browser::hide() [inline, virtual]

Hides the entire [FL_Browser](#) widget – opposite of [show\(\)](#).

Reimplemented from [FL_Widget](#).

30.6.3.16 void FL_Browser::hide(int line)

Makes `line` invisible, preventing selection by the user.

The line can still be selected under program control. This changes the [full_height\(\)](#) if the state was changed. When a line is made invisible, lines below it are moved up in the display. [redraw\(\)](#) is called automatically if a change occurred.

Parameters:

← *line* The line to be hidden. (1 based)

30.6.3.17 `int FL_Browser::incr_height () const` [protected, virtual]

The default height of items (including spacing in-between) in pixels.

This currently returns `textsize() + 2`.

Returns:

The value in pixels.

Reimplemented from [FL_Browser_](#).

30.6.3.18 `void FL_Browser::insert (int line, const char * newtext, void * d = 0)`

Insert a new entry whose label is *newtext* *above* given *line*, optional data *d*.

Text may contain format characters; see [format_char\(\)](#) for details. *newtext* is copied using the `strdup()` function, and can be NULL to make a blank line.

The optional void * argument *d* will be the [data\(\)](#) of the new item.

Parameters:

- ← *line* Line position for insert. (1 based)
If *line* > [size\(\)](#), the entry will be added at the end.
- ← *newtext* The label text for the new line.
- ← *d* Optional pointer to user data to be associated with the new line.

30.6.3.19 `void FL_Browser::insert (int line, FL_BLINE * item)` [protected]

Insert specified *item* above *line*.

If *line* > [size\(\)](#) then the line is added to the end.

Caveat: See efficiency note in [find_line\(\)](#).

Parameters:

- ← *line* The new line will be inserted above this line (1 based).
- ← *item* The item to be added.

30.6.3.20 `void* FL_Browser::item_at (int line) const` [inline, protected, virtual]

Return the item at specified *line*.

Parameters:

- ← *line* The line of the item to return. (1 based)

Returns:

The item, or NULL if line out of range.

Reimplemented from [FL_Browser_](#).

30.6.3.21 `void Fl_Browser::item_draw (void * item, int X, int Y, int W, int H) const` [protected, virtual]

Draws *item* at the position specified by *X Y W H*.

The *W* and *H* values are used for clipping. Should only be called within the context of an FLTK [draw\(\)](#).

Parameters:

- ← *item* The item to be drawn
- ← *X,Y,W,H* position and size.

Implements [Fl_Browser_](#).

30.6.3.22 `void * Fl_Browser::item_first () const` [protected, virtual]

Returns the very first item in the list.

Example of use:

```
// Walk the browser from beginning to end
for ( void *i=item_first(); i; i=item_next(i) ) {
    printf("item label='%s'\n", item_text(i));
}
```

Returns:

The first item, or NULL if list is empty.

Implements [Fl_Browser_](#).

30.6.3.23 `int Fl_Browser::item_height (void * item) const` [protected, virtual]

Returns height of *item* in pixels.

This takes into account embedded @ codes within the [text\(\)](#) label.

Parameters:

- ← *item* The item whose height is returned.

Returns:

The height of the item in pixels.

Implements [Fl_Browser_](#).

30.6.3.24 `void * Fl_Browser::item_last () const` [protected, virtual]

Returns the very last item in the list.

Example of use:

```
// Walk the browser in reverse, from end to start
for ( void *i=item_last(); i; i=item_prev(i) ) {
    printf("item label='%s'\n", item_text(i));
}
```


Returns:

The last item, or NULL if list is empty.

Reimplemented from [Fl_Browser_](#).

30.6.3.25 void * Fl_Browser::item_next (void * *item*) const [protected, virtual]

Returns the next item after *item*.

Parameters:

← *item* The 'current' item

Returns:

The next item after *item*, or NULL if there are none after this one.

See also:

[item_first\(\)](#)

Implements [Fl_Browser_](#).

30.6.3.26 void * Fl_Browser::item_prev (void * *item*) const [protected, virtual]

Returns the previous item before *item*.

Parameters:

← *item* The 'current' item

Returns:

The previous item before *item*, or NULL if there none before this one.

See also:

[item_last\(\)](#)

Implements [Fl_Browser_](#).

30.6.3.27 void Fl_Browser::item_select (void * *item*, int *val*) [protected, virtual]

Change the selection state of *item* to the value *val*.

Parameters:

← *item* The item to be changed.

← *val* The new selection state: 1 selects, 0 de-selects.

Reimplemented from [Fl_Browser_](#).

30.6.3.28 `int Fl_Browser::item_selected (void * item) const` [protected, virtual]

See if `item` is selected.

Parameters:

← *item* The item whose selection state is to be checked.

Returns:

1 if selected, 0 if not.

Reimplemented from [Fl_Browser_](#).

30.6.3.29 `void Fl_Browser::item_swap (void * a, void * b)` [inline, protected, virtual]

Swap the items `a` and `b`.

You must call [redraw\(\)](#) to make any changes visible.

Parameters:

← *a, b* the items to be swapped.

Reimplemented from [Fl_Browser_](#).

30.6.3.30 `const char * Fl_Browser::item_text (void * item) const` [protected, virtual]

Returns the label text for `item`.

Parameters:

← *item* The item whose label text is returned.

Returns:

The item's text string. (Can be NULL)

Reimplemented from [Fl_Browser_](#).

30.6.3.31 `int Fl_Browser::item_width (void * item) const` [protected, virtual]

Returns width of `item` in pixels.

This takes into account embedded @ codes within the [text\(\)](#) label.

Parameters:

← *item* The item whose width is returned.

Returns:

The width of the item in pixels.

Implements [Fl_Browser_](#).

30.6.3.32 `int FL_Browser::lineno (void * item) const` `[protected]`

Returns line number corresponding to `item`, or zero if not found.

Caveat: See efficiency note in [find_line\(\)](#).

Parameters:

← *item* The item to be found

Returns:

The line number of the item, or 0 if not found.

30.6.3.33 `void FL_Browser::lineposition (int line, FL_Line_Position pos)`

Updates the browser so that `line` is shown at position `pos`.

Parameters:

← *line* line number. (1 based)

← *pos* position.

30.6.3.34 `int FL_Browser::load (const char *filename)`

Clears the browser and reads the file, adding each line from the file to the browser.

If the filename is NULL or a zero-length string then this just clears the browser. This returns zero if there was any error in opening or reading the file, in which case `errno` is set to the system error. The [data\(\)](#) of each line is set to NULL.

30.6.3.35 `void FL_Browser::make_visible (int line)` `[inline]`

Make the item at the specified `line` [visible\(\)](#).

Functionally similar to [show\(int line\)](#). If `line` is out of range, redisplay top or bottom of list as appropriate.

Parameters:

← *line* The line to be made visible.

See also:

[show\(int\)](#)

30.6.3.36 `void FL_Browser::middleline (int line)` `[inline]`

Scrolls the browser so the middle item in the browser is showing the specified `line`.

30.6.3.37 void Fl_Browser::move (int *to*, int *from*)

Line *from* is removed and reinserted at *to*.

Note: *to* is calculated *after* line *from* gets removed.

Parameters:

- ← *to* Destination line number (calculated *after* line *from* is removed)
- ← *from* Line number of item to be moved

30.6.3.38 void Fl_Browser::remove (int *line*)

Remove entry for given *line* number, making the browser one line shorter.

You must call [redraw\(\)](#) to make any changes visible.

Parameters:

- ← *line* Line to be removed. (1 based)
- If *line* is out of range, no action is taken.

30.6.3.39 void Fl_Browser::replace (int *a*, const char * *b*) [inline]

For back compatibility only.

30.6.3.40 int Fl_Browser::select (int *line*, int *val* = 1)

Sets the selection state of the item at *line* to the value *val*.

If *val* is not specified, the default is 1 (selects the item).

Parameters:

- ← *line* The line number of the item to be changed. (1 based)
- ← *val* The new selection state (1=select, 0=de-select).

Returns:

1 if the state changed, 0 if not.

30.6.3.41 int Fl_Browser::selected (int *line*) const

Returns 1 if specified *line* is selected, 0 if not.

Parameters:

- ← *line* The line being checked (1 based)

Returns:

1 if item selected, 0 if not.

30.6.3.42 void FL_Browser::show () [inline, virtual]

Shows the entire [FL_Browser](#) widget – opposite of [hide\(\)](#).

Reimplemented from [FL_Widget](#).

30.6.3.43 void FL_Browser::show (int line)

Makes `line` visible, and available for selection by user.

Opposite of [hide\(int\)](#). This changes the [full_height\(\)](#) if the state was changed. [redraw\(\)](#) is called automatically if a change occurred.

Parameters:

← *line* The line to be shown. (1 based)

30.6.3.44 void FL_Browser::size (int W, int H) [inline]

Changes the size of the widget.

`size(W, H)` is a shortcut for `resize(x(), y(), W, H)`.

Parameters:

← *W,H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented from [FL_Widget](#).

30.6.3.45 int FL_Browser::size () const [inline]

Returns how many lines are in the browser.

The last line number is equal to this. Returns 0 if browser is empty.

30.6.3.46 void FL_Browser::swap (int a, int b)

Swaps two browser lines `a` and `b`.

You must call [redraw\(\)](#) to make any changes visible.

Parameters:

← *a,b* The two lines to be swapped. (both 1 based)

30.6.3.47 void FL_Browser::swap (FL_BLINE *a, FL_BLINE *b) [protected]

Swap the two items `a` and `b`.

You must call [redraw\(\)](#) to make any changes visible.

Parameters:

← *a,b* The two items to be swapped.

30.6.3.48 void FL_Browser::text (int *line*, const char * *newtext*)

Sets the text for the specified *line* to *newtext*.

Text may contain format characters; see [format_char\(\)](#) for details. *newtext* is copied using the `strdup()` function, and can be NULL to make a blank line.

Does nothing if *line* is out of range.

Parameters:

← *line* The line of the item whose text will be changed. (1 based)

← *newtext* The new string to be assigned to the item.

30.6.3.49 const char * FL_Browser::text (int *line*) const

Returns the label text for the specified *line*.

Return value can be NULL if *line* is out of range or unset. The parameter *line* is 1 based.

Parameters:

← *line* The line number of the item whose text is returned. (1 based)

Returns:

The text string (can be NULL)

30.6.3.50 void FL_Browser::topline (int *line*) [inline]

Scrolls the browser so the top item in the browser is showing the specified *line*.

30.6.3.51 int FL_Browser::topline () const

Returns the line that is currently visible at the top of the browser.

If there is no vertical scrollbar then this will always return 1.

Returns:

The [lineno\(\)](#) of the [top\(\)](#) of the browser.

30.6.3.52 void FL_Browser::value (int *line*) [inline]

Sets the browser's [value\(\)](#), which selects the specified *line*.

This is the same as calling `select(line)`.

30.6.3.53 int Fl_Browser::value () const

Returns the line number of the currently selected line, or 0 if none.

Returns:

The line number of current selection, or 0 if none selected.

30.6.3.54 int Fl_Browser::visible (int *line*) const

Returns non-zero if the specified `line` is visible, 0 if hidden.

Use [show\(int\)](#), [hide\(int\)](#), or [make_visible\(int\)](#) to change an item's visible state.

Parameters:

← *line* The line in the browser to be tested. (1 based)

The documentation for this class was generated from the following files:

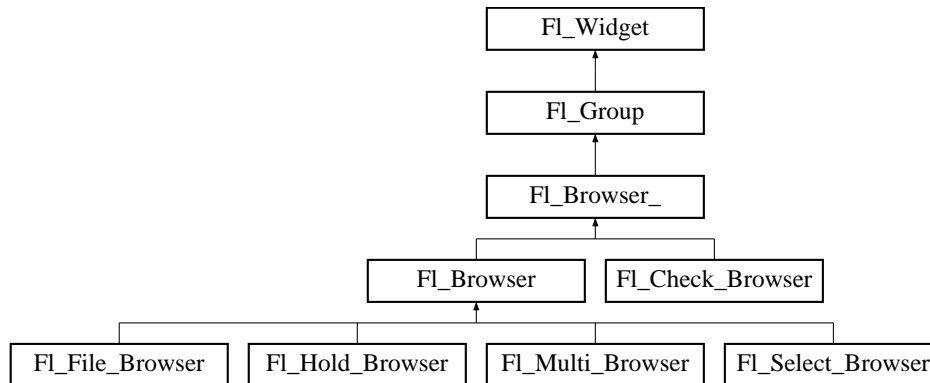
- Fl_Browser.H
- Fl_Browser.cxx
- Fl_Browser_load.cxx

30.7 Fl_Browser_ Class Reference

This is the base for browsers.

```
#include <Fl_Browser_.H>
```

Inheritance diagram for Fl_Browser_:



Public Types

- enum {
HORIZONTAL = 1, **VERTICAL** = 2, **BOTH** = 3, **ALWAYS_ON** = 4,
HORIZONTAL_ALWAYS = 5, **VERTICAL_ALWAYS** = 6, **BOTH_ALWAYS** = 7 }

Public Member Functions

- int **deselect** (int docallbacks=0)
Deselects all items in the list and returns 1 if the state changed or 0 if it did not.
- void **display** (void *)
Displays item p, scrolling the list as necessary.
- int **handle** (int)
Handles an event within the normal widget bounding box.
- void **has_scrollbar** (uchar i)
By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the widget.
- uchar **has_scrollbar** () const
See Fl_Browser_::has_scrollbar(uchar).
- void **hposition** (int)
Gets or sets the horizontal scrolling position of the list, which is the pixel offset of the list items within the list area.
- int **hposition** () const

Gets or sets the horizontal scrolling position of the list, which is the pixel offset of the list items within the list area.

- void [position](#) (int)

Gets or sets the vertical scrolling position of the list, which is the pixel offset of the list items within the list area.

- int [position](#) () const

Gets or sets the vertical scrolling position of the list, which is the pixel offset of the list items within the list area.

- void [resize](#) (int, int, int, int)

Repositions and/or resizes the browser.

- void [scrollbar_left](#) ()

Moves the vertical scrollbar to the lefthand side of the list.

- void [scrollbar_right](#) ()

Moves the vertical scrollbar to the righthand side of the list.

- int [select](#) (void *, int=1, int docallbacks=0)

Sets the selection state of item p to s and returns 1 if the state changed or 0 if it did not.

- int [select_only](#) (void *, int docallbacks=0)

Selects item p and returns 1 if the state changed or 0 if it did not.

- void [sort](#) (int flags=0)

Sort the items in the browser.

- void [textcolor](#) (unsigned n)

Sets the default text color to color n.

- [FL_Color textcolor](#) () const

Gets the default text color for the lines in the browser.

- void [textfont](#) ([FL_Font](#) s)

Sets the default text font to font s.

- [FL_Font textfont](#) () const

Gets the default text font for the lines in the browser.

- void [textsize](#) ([FL_Fontsize](#) s)

Sets the default text size to size s.

- [FL_Fontsize textsize](#) () const

Gets the default text size for the lines in the browser.

Static Public Member Functions

- static void [scrollbar_width](#) (int b)
Sets the width of any scrollbars that are used to b.
- static int [scrollbar_width](#) ()
Gets the width of any scrollbars that are used.

Public Attributes

- [Fl_Scrollbar hscrollbar](#)
Horizontal scrollbar.
- [Fl_Scrollbar scrollbar](#)
Vertical scrollbar.

Protected Member Functions

- void [bbox](#) (int &, int &, int &, int &) const
This method returns the bounding box for the interior of the list, inside the scrollbars.
- void [deleting](#) (void *a)
This method should be used when an item is deleted from the list.
- int [displayed](#) (void *) const
This method returns non-zero if item p is currently visible in the list.
- void [draw](#) ()
Draws the list within the normal widget bounding box.
- void [draw](#) (int, int, int, int)
Draws the contents of the browser within the specified bounding box.
- void * [find_item](#) (int my)
This method returns the item under mouse at my.
- [Fl_Browser_](#) (int, int, int, int, const char * = 0)
The constructor makes an empty browser.
- virtual int [full_height](#) () const
This method may be provided by the subclass to indicate the full height of the item list in pixels.
- virtual int [full_width](#) () const
This method may be provided by the subclass to indicate the full width of the item list in pixels.
- virtual int [incr_height](#) () const
This method may be provided to return the average height of all items, to be used for scrolling.

- void `inserting` (void *a, void *b)
This method should be used when an item is added to the list.
- virtual void * `item_at` (int) const
Return the item a specified index.
- virtual void `item_draw` (void *, int, int, int, int) const =0
This method must be provided by the subclass to draw the item p in the area indicated by x, y, w, and h.
- virtual void * `item_first` () const =0
This method must be provided by the subclass to return the first item in the list.
- virtual int `item_height` (void *) const =0
This method must be provided by the subclass to return the height of the item p in pixels.
- virtual void * `item_last` () const
This method can be provided by the subclass to return the ilast item in the list.
- virtual void * `item_next` (void *) const =0
This method must be provided by the subclass to return the item in the list after p.
- virtual void * `item_prev` (void *) const =0
This method must be provided by the subclass to return the item in the list before p.
- virtual int `item_quick_height` (void *) const
This method may be provided by the subclass to return the height of the item p in pixels.
- virtual void `item_select` (void *, int=1)
This method must be implemented by the subclass if it supports multiple selections in the browser.
- virtual int `item_selected` (void *) const
This method must be implemented by the subclass if it supports multiple selections in the browser.
- virtual void `item_swap` (void *, void *)
This optional function is required for sorting browser items.
- virtual const char * `item_text` (void *item) const
This optional function returns a string that may be used for sorting.
- virtual int `item_width` (void *) const =0
This method must be provided by the subclass to return the width of the item p in pixels.
- int `leftedge` () const
This method returns the X position of the left edge of the list area after adjusting for the scrollbar and border, if any.
- void `new_list` ()
This method should be called when the list data is completely replaced or cleared.

- void [redraw_line](#) (void *)

This method should be called when the contents of an item have changed but not changed the height of the item.

- void [redraw_lines](#) ()

This method will cause the entire list to be redrawn.

- void [replacing](#) (void *a, void *b)

This method should be used when an item is replaced in the list.

- void * [selection](#) () const

Returns the item currently selected, or NULL if there is no selection.

- void [swapping](#) (void *a, void *b)

Exchange pointers a and b.

- void * [top](#) () const

Returns the item the appears at the top of the list.

30.7.1 Detailed Description

This is the base for browsers.

To be useful it must be subclassed and several virtual functions defined. The Forms-compatible browser and the file chooser's browser are subclassed off of this.

This has been designed so that the subclass has complete control over the storage of the data, although because next() and prev() functions are used to index, it works best as a linked list or as a large block of characters in which the line breaks must be searched for.

A great deal of work has been done so that the "height" of a data object does not need to be determined until it is drawn. This is useful if actually figuring out the size of an object requires accessing image data or doing stat() on a file or doing some other slow operation.

30.7.2 Constructor & Destructor Documentation

30.7.2.1 [Fl_Browser::Fl_Browser_](#) (int X, int Y, int W, int H, const char * l = 0) [protected]

The constructor makes an empty browser.

30.7.3 Member Function Documentation

30.7.3.1 void [Fl_Browser::deleting](#) (void * l) [protected]

This method should be used when an item is deleted from the list.

It allows the [Fl_Browser_](#) to discard any cached data it has on the item.

30.7.3.2 `int Fl_Browser_::deselect (int docallbacks = 0)`

Deselects all items in the list and returns 1 if the state changed or 0 if it did not.

If `docb` is non-zero, `deselect` tries to call the callback function for the widget.

30.7.3.3 `void Fl_Browser_::display (void *p)`

Displays item `p`, scrolling the list as necessary.

30.7.3.4 `void Fl_Browser_::draw (int, int, int, int) [protected]`

Draws the contents of the browser within the specified bounding box.

Todo

Find the implementation, if any, and document it there!

30.7.3.5 `void * Fl_Browser_::find_item (int my) [protected]`

This method returns the item under mouse at `my`.

If no item is displayed at that position then `NULL` is returned.

30.7.3.6 `int Fl_Browser_::full_height () const [protected, virtual]`

This method may be provided by the subclass to indicate the full height of the item list in pixels.

The default implementation computes the full height from the item heights.

Reimplemented in [Fl_Browser](#).

30.7.3.7 `int Fl_Browser_::full_width () const [protected, virtual]`

This method may be provided by the subclass to indicate the full width of the item list in pixels.

The default implementation computes the full width from the item widths.

30.7.3.8 `int Fl_Browser_::handle (int event) [virtual]`

Handles an event within the normal widget bounding box.

Reimplemented from [Fl_Group](#).

Reimplemented in [Fl_Check_Browser](#).

30.7.3.9 `void Fl_Browser_::has_scrollbar (uchar i) [inline]`

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the widget.

[has_scrollbar\(\)](#) changes this based on the value of `h`:

- 0 - No scrollbars.

- `Fl_Browser_::HORIZONTAL` - Only a horizontal scrollbar.
- `Fl_Browser_::VERTICAL` - Only a vertical scrollbar.
- `Fl_Browser_::BOTH` - The default is both scrollbars.
- `Fl_Browser_::HORIZONTAL_ALWAYS` - Horizontal scrollbar always on, vertical always off.
- `Fl_Browser_::VERTICAL_ALWAYS` - Vertical scrollbar always on, horizontal always off.
- `Fl_Browser_::BOTH_ALWAYS` - Both always on.

30.7.3.10 `int Fl_Browser_::incr_height () const` [protected, virtual]

This method may be provided to return the average height of all items, to be used for scrolling.

The default implementation uses the height of the first item.

Reimplemented in [Fl_Browser](#).

30.7.3.11 `void Fl_Browser_::inserting (void *a, void *b)` [protected]

This method should be used when an item is added to the list.

It allows the [Fl_Browser_](#) to update its cache data as needed.

30.7.3.12 `virtual void* Fl_Browser_::item_first () const` [protected, pure virtual]

This method must be provided by the subclass to return the first item in the list.

Implemented in [Fl_Browser](#).

30.7.3.13 `virtual int Fl_Browser_::item_height (void *) const` [protected, pure virtual]

This method must be provided by the subclass to return the height of the item p in pixels.

Allow for two additional pixels for the list selection box.

Implemented in [Fl_Browser](#).

30.7.3.14 `virtual void* Fl_Browser_::item_last () const` [inline, protected, virtual]

This method can be provided by the subclass to return the ilast item in the list.

Reimplemented in [Fl_Browser](#).

30.7.3.15 `virtual void* Fl_Browser_::item_next (void *) const` [protected, pure virtual]

This method must be provided by the subclass to return the item in the list after p.

Implemented in [Fl_Browser](#).

30.7.3.16 `virtual void* Fl_Browser_::item_prev (void *) const` [protected, pure virtual]

This method must be provided by the subclass to return the item in the list before `p`.

Implemented in [Fl_Browser](#).

30.7.3.17 `int Fl_Browser_::item_quick_height (void * l) const` [protected, virtual]

This method may be provided by the subclass to return the height of the item `p` in pixels.

Allow for two additional pixels for the list selection box. This method differs from `item_height` in that it is only called for selection and scrolling operations. The default implementation calls `item_height`.

30.7.3.18 `void Fl_Browser_::item_select (void *, int = 1)` [protected, virtual]

This method must be implemented by the subclass if it supports multiple selections in the browser.

The `s` argument specifies the selection state for item `p`: 0 = off, 1 = on.

Reimplemented in [Fl_Browser](#).

30.7.3.19 `int Fl_Browser_::item_selected (void * l) const` [protected, virtual]

This method must be implemented by the subclass if it supports multiple selections in the browser.

The method should return 1 if `p` is selected and 0 otherwise.

Reimplemented in [Fl_Browser](#).

30.7.3.20 `virtual int Fl_Browser_::item_width (void *) const` [protected, pure virtual]

This method must be provided by the subclass to return the width of the item `p` in pixels.

Allow for two additional pixels for the list selection box.

Implemented in [Fl_Browser](#).

30.7.3.21 `void Fl_Browser_::new_list ()` [protected]

This method should be called when the list data is completely replaced or cleared.

It informs the [Fl_Browser_](#) widget that any cached information it has concerning the items is invalid.

30.7.3.22 `void Fl_Browser_::redraw_lines ()` [inline, protected]

This method will cause the entire list to be redrawn.

30.7.3.23 `void Fl_Browser_::replacing (void * a, void * b)` [protected]

This method should be used when an item is replaced in the list.

It allows the [Fl_Browser_](#) to update its cache data as needed.

30.7.3.24 void FL_Browser_::resize (int *X*, int *Y*, int *W*, int *H*) [virtual]

Repositions and/or resizes the browser.

Reimplemented from [FL_Group](#).

30.7.3.25 void FL_Browser_::scrollbar_left () [inline]

Moves the vertical scrollbar to the lefthand side of the list.

For back compatibility.

30.7.3.26 void FL_Browser_::scrollbar_right () [inline]

Moves the vertical scrollbar to the righthand side of the list.

For back compatibility.

30.7.3.27 int FL_Browser_::select (void **l*, int *i* = 1, int *docallbacks* = 0)

Sets the selection state of item *p* to *s* and returns 1 if the state changed or 0 if it did not.

If *docb* is non-zero, select tries to call the callback function for the widget.

30.7.3.28 int FL_Browser_::select_only (void **l*, int *docallbacks* = 0)

Selects item *p* and returns 1 if the state changed or 0 if it did not.

Any other items in the list are deselected.

If *docb* is non-zero, select_only tries to call the callback function for the widget.

30.7.3.29 void* FL_Browser_::selection () const [inline, protected]

Returns the item currently selected, or NULL if there is no selection.

For multiple selection browsers this call returns the currently focused item, even if it is not selected. To find all selected items, call

[FL_Multi_Browser::selected\(\)](#) for every item in question.

30.7.3.30 void FL_Browser_::sort (int *flags* = 0)

Sort the items in the browser.

[item_swap\(void*, void*\)](#) and [item_text\(void*\)](#) must be implemented for this call.

Parameters:

← *flags* no flags were defined yet. Sorting in descending order and sorting while ignoring case come to mind.

30.7.3.31 void* Fl_Browser_::top () const [inline, protected]

Returns the item the appears at the top of the list.

30.7.4 Member Data Documentation**30.7.4.1 Fl_Scrollbar Fl_Browser_::scrollbar**

Vertical scrollbar.

The documentation for this class was generated from the following files:

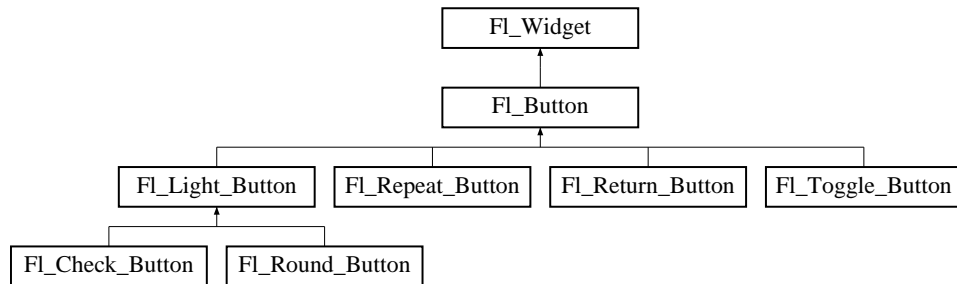
- Fl_Browser_.H
- Fl_Browser_.cxx

30.8 FL_Button Class Reference

Buttons generate callbacks when they are clicked by the user.

```
#include <Fl_Button.H>
```

Inheritance diagram for FL_Button::



Public Member Functions

- `int clear ()`
Same as `value (0)`.
- `void down_box (FL_Boxtype b)`
Sets the down box type.
- `FL_Boxtype down_box () const`
Returns the current down box type, which is drawn when `value()` is non-zero.
- `void down_color (unsigned c)`
(for backwards compatibility)
- `FL_Color down_color () const`
(for backwards compatibility)
- `FL_Button (int X, int Y, int W, int H, const char *L=0)`
The constructor creates the button using the given position, size and label.
- `virtual int handle (int)`
Handles the specified event.
- `int set ()`
Same as `value (1)`.
- `void setonly ()`
Turns on this button and turns off all other radio buttons in the group (calling `value (1)` or `set ()` does not do this).
- `void shortcut (const char *s)`
(for backwards compatibility)

- void `shortcut` (int s)
Sets the shortcut key to s.
- int `shortcut` () const
Returns the current shortcut key for the button.
- char `value` () const
Returns the current value of the button (0 or 1).
- int `value` (int v)
Sets the current value of the button.

Protected Member Functions

- virtual void `draw` ()
Draws the widget.

30.8.1 Detailed Description

Buttons generate callbacks when they are clicked by the user.

You control exactly when and how by changing the values for `type()` and `when()`. Buttons can also generate callbacks in response to `FL_SHORTCUT` events. The button can either have an explicit `shortcut(int s)` value or a letter shortcut can be indicated in the `label()` with an `'&'` character before it. For the label shortcut it does not matter if *Alt* is held down, but if you have an input field in the same window, the user will have to hold down the *Alt* key so that the input field does not eat the event first as an `FL_KEYBOARD` event.

Todo

Refactor the doxygen comments for `FL_Button type()` documentation.

For an `FL_Button` object, the `type()` call returns one of:

- `FL_NORMAL_BUTTON` (0): `value()` remains unchanged after button press.
- `FL_TOGGLE_BUTTON`: `value()` is inverted after button press.
- `FL_RADIO_BUTTON`: `value()` is set to 1 after button press, and all other buttons in the current group with `type() == FL_RADIO_BUTTON` are set to zero.

Todo

Refactor the doxygen comments for `FL_Button when()` documentation.

For an `FL_Button` object, the following `when()` values are useful, the default being `FL_WHEN_RELEASE`:

- 0: The callback is not done, instead `changed()` is turned on.
- `FL_WHEN_RELEASE`: The callback is done after the user successfully clicks the button, or when a shortcut is typed.
- `FL_WHEN_CHANGED`: The callback is done each time the `value()` changes (when the user pushes and releases the button, and as the mouse is dragged around in and out of the button).

30.8.2 Constructor & Destructor Documentation

30.8.2.1 `Fl_Button::Fl_Button (int X, int Y, int W, int H, const char * L = 0)`

The constructor creates the button using the given position, size and label.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.8.3 Member Function Documentation

30.8.3.1 `int Fl_Button::clear ()` `[inline]`

Same as `value(0)`.

See also:

[value\(int v\)](#)

30.8.3.2 `void Fl_Button::down_box (Fl_Boxtype b)` `[inline]`

Sets the down box type.

The default value of 0 causes FLTK to figure out the correct matching down version of [box\(\)](#).

Parameters:

- ← *b* down box type

30.8.3.3 `Fl_Boxtype Fl_Button::down_box () const` `[inline]`

Returns the current down box type, which is drawn when [value\(\)](#) is non-zero.

Return values:

Fl_Boxtype

30.8.3.4 `void Fl_Button::draw ()` `[protected, virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

Reimplemented in [Fl_Light_Button](#), and [Fl_Return_Button](#).

30.8.3.5 int Fl_Button::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

Reimplemented in [Fl_Light_Button](#), [Fl_Repeat_Button](#), and [Fl_Return_Button](#).

30.8.3.6 int Fl_Button::set () [inline]

Same as `value(1)`.

See also:

[value\(int v\)](#)

30.8.3.7 void Fl_Button::shortcut (int s) [inline]

Sets the shortcut key to *s*.

Setting this overrides the use of '&' in the [label\(\)](#). The value is a bitwise OR of a key and a set of shift flags, for example: `FL_ALT | 'a'`, or `FL_ALT | (FL_F + 10)`, or just `'a'`. A value of 0 disables the shortcut.

The key can be any value returned by [Fl::event_key\(\)](#), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by [Fl::event_state\(\)](#). If the bit is on, that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

Parameters:

← *s* bitwise OR of key and shift flags

30.8.3.8 int Fl_Button::shortcut () const [inline]

Returns the current shortcut key for the button.

Return values:

int

30.8.3.9 int Fl_Button::value (int v)

Sets the current value of the button.

A non-zero value sets the button to 1 (ON), and zero sets it to 0 (OFF).

Parameters:

← *v* button value.

See also:

[set\(\)](#), [clear\(\)](#)

The documentation for this class was generated from the following files:

- Fl_Button.H
- Fl_Button.cxx

30.9 Fl_Cairo_State Class Reference

Contains all the necessary info on the current cairo context.

```
#include <Fl_Cairo.H>
```

Public Member Functions

- void [autolink](#) (bool b)
Sets the autolink option, only available with `--enable-cairoext`.
- bool [autolink](#) () const
Gets the autolink option. See [Fl::cairo_autolink_context\(bool\)](#).
- void [cc](#) (cairo_t *c, bool own=true)
Sets the current cairo context, own indicates cc deletion is handle externally by user.
- cairo_t * [cc](#) () const
Gets the current cairo context.
- void * [gc](#) () const
Gets the last gc attached to a cc.
- void [gc](#) (void *c)
Sets the gc c to keep track on.
- void * [window](#) () const
Gets the last window attached to a cc.
- void [window](#) (void *w)
Sets the window w to keep track on.

30.9.1 Detailed Description

Contains all the necessary info on the current cairo context.

A private internal & unique corresponding object is created to permit cairo context state handling while keeping it opaque. For internal use only.

Note:

Only available when configure has the `--enable-cairo` option

The documentation for this class was generated from the following file:

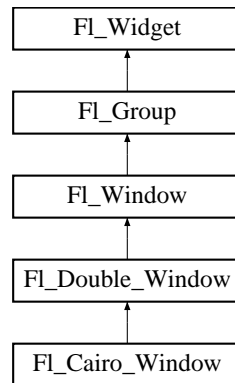
- Fl_Cairo.H

30.10 FL_Cairo_Window Class Reference

This defines a pre-configured cairo fltk window.

```
#include <Fl_Cairo_Window.H>
```

Inheritance diagram for FL_Cairo_Window::



Public Types

- typedef void(* [cairo_draw_cb](#))(FL_Cairo_Window *self, cairo_t *def)
This defines the cairo draw callback prototype that you must further.

Public Member Functions

- **FL_Cairo_Window** (int w, int h)
- void [set_draw_cb](#) ([cairo_draw_cb](#) cb)
You must provide a draw callback which will implement your cairo rendering.

Protected Member Functions

- void [draw](#) ()
Overloaded to provide cairo callback support.

30.10.1 Detailed Description

This defines a pre-configured cairo fltk window.

This class overloads the virtual [draw\(\)](#) method for you, so that the only thing you have to do is to provide your cairo code. All cairo context handling is achieved transparently.

Note:

You can alternatively define your custom cairo fltk window, and thus at least override the [draw\(\)](#) method to provide custom cairo support. In this case you will probably use `Fl::cairo_make_`

`current(Fl_Window*)` to attach a context to your window. You should do it only when your window is the current window.

See also:

[Fl_Window::current\(\)](#)

30.10.2 Member Function Documentation

30.10.2.1 `void Fl_Cairo_Window::set_draw_cb (cairo_draw_cb cb)` [inline]

You must provide a draw callback which will implement your cairo rendering.

This method will permit you to set your cairo callback to `cb`.

The documentation for this class was generated from the following file:

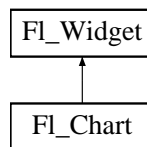
- `Fl_Cairo_Window.H`

30.11 FL_Chart Class Reference

[FL_Chart](#) displays simple charts.

```
#include <FL_Chart.H>
```

Inheritance diagram for `FL_Chart`:



Public Member Functions

- void [add](#) (double val, const char *str=0, unsigned col=0)
Add the data value val with optional label str and color col to the chart.
- void [autosize](#) (uchar n)
Set whether the chart will automatically adjust the bounds of the chart.
- [uchar autosize](#) () const
Get whether the chart will automatically adjust the bounds of the chart.
- void [bounds](#) (double a, double b)
Sets the lower and upper bounds of the chart values.
- void [bounds](#) (double *a, double *b) const
Gets the lower and upper bounds of the chart values.
- void [clear](#) ()
Removes all values from the chart.
- [FL_Chart](#) (int X, int Y, int W, int H, const char *L=0)
Create a new FL_Chart widget using the given position, size and label string.
- void [insert](#) (int ind, double val, const char *str=0, unsigned col=0)
Inserts a data value val at the given position ind.
- void [maxsize](#) (int m)
Set the maximum number of data values for a chart.
- int [maxsize](#) () const
Gets the maximum number of data values for a chart.
- void [replace](#) (int ind, double val, const char *str=0, unsigned col=0)
Replace a data value val at the given position ind.
- void [size](#) (int W, int H)

Changes the size of the widget.

- `int size () const`
Returns the number of data values in the chart.
- `void textcolor (unsigned n)`
gets the chart's text color to n.
- `FL_Color textcolor () const`
Gets the chart's text color.
- `void textfont (FL_Font s)`
Sets the chart's text font to s.
- `FL_Font textfont () const`
Gets the chart's text font.
- `void textsize (FL_Fontsize s)`
gets the chart's text size to s.
- `FL_Fontsize textsize () const`
Gets the chart's text size.
- `~FL_Chart ()`
Destroys the FL_Chart widget and all of its data.

Protected Member Functions

- `void draw ()`
Draws the widget.

30.11.1 Detailed Description

`FL_Chart` displays simple charts.

It is provided for Forms compatibility.

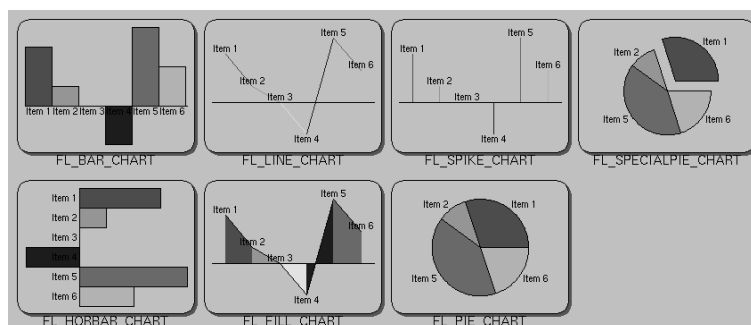


Figure 30.2: FL_Chart

Todo

Refactor `Fl_Chart::type()` information.

The type of an `Fl_Chart` object can be set using `type(uchar t)` to:

- `FL_BAR_CHART`: Each sample value is drawn as a vertical bar.
- `FL_FILLED_CHART`: The chart is filled from the bottom of the graph to the sample values.
- `FL_HORBAR_CHART`: Each sample value is drawn as a horizontal bar.
- `FL_LINE_CHART`: The chart is drawn as a polyline with vertices at each sample value.
- `FL_PIE_CHART`: A pie chart is drawn with each sample value being drawn as a proportionate slice in the circle.
- `FL_SPECIALPIE_CHART`: Like `FL_PIE_CHART`, but the first slice is separated from the pie.
- `FL_SPIKE_CHART`: Each sample value is drawn as a vertical line.

30.11.2 Constructor & Destructor Documentation

30.11.2.1 `Fl_Chart::Fl_Chart (int X, int Y, int W, int H, const char * L = 0)`

Create a new `Fl_Chart` widget using the given position, size and label string.

The default boxstyle is `FL_NO_BOX`.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.11.3 Member Function Documentation

30.11.3.1 `void Fl_Chart::add (double val, const char * str = 0, unsigned col = 0)`

Add the data value *val* with optional label *str* and color *col* to the chart.

Parameters:

- ← *val* data value
- ← *str* optional data label
- ← *col* optional data color

30.11.3.2 `void Fl_Chart::autosize (uchar n)` `[inline]`

Set whether the chart will automatically adjust the bounds of the chart.

Parameters:

- ← *n* non-zero to enable automatic resizing, zero to disable.

30.11.3.3 uchar Fl_Chart::autosize () const [inline]

Get whether the chart will automatically adjust the bounds of the chart.

Returns:

non-zero if auto-sizing is enabled and zero if disabled.

30.11.3.4 void Fl_Chart::bounds (double *a*, double *b*)

Sets the lower and upper bounds of the chart values.

Parameters:

← *a, b* are used to set lower, upper

30.11.3.5 void Fl_Chart::bounds (double **a*, double **b*) const [inline]

Gets the lower and upper bounds of the chart values.

Parameters:

→ *a, b* are set to lower, upper

30.11.3.6 void Fl_Chart::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to Fl_Widget* and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.11.3.7 void Fl_Chart::insert (int *ind*, double *val*, const char **str* = 0, unsigned *col* = 0)

Inserts a data value *val* at the given position *ind*.

Position 1 is the first data value.

Parameters:

← *ind* insertion position
 ← *val* data value
 ← *str* optional data label
 ← *col* optional data color

30.11.3.8 void FL_Chart::maxsize (int *m*)

Set the maximum number of data values for a chart.

If you do not call this method then the chart will be allowed to grow to any size depending on available memory.

Parameters:

← *m* maximum number of data values allowed.

30.11.3.9 void FL_Chart::replace (int *ind*, double *val*, const char * *str* = 0, unsigned *col* = 0)

Replace a data value *val* at the given position *ind*.

Position 1 is the first data value.

Parameters:

← *ind* insertion position

← *val* data value

← *str* optional data label

← *col* optional data color

30.11.3.10 void FL_Chart::size (int *W*, int *H*) [inline]

Changes the size of the widget.

size(*W*, *H*) is a shortcut for `resize(x(), y(), W, H)`.

Parameters:

← *W,H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented from [FL_Widget](#).

30.11.3.11 void FL_Chart::textcolor (unsigned *n*) [inline]

gets the chart's text color to *n*.

30.11.3.12 void FL_Chart::textfont (FL_Font *s*) [inline]

Sets the chart's text font to *s*.

30.11.3.13 void Fl_Chart::textsize (Fl_Fontsize *s*) `[inline]`

gets the chart's text size to *s*.

The documentation for this class was generated from the following files:

- Fl_Chart.H
- Fl_Chart.cxx

30.12 FL_CHART_ENTRY Struct Reference

For internal use only.

```
#include <Fl_Chart.H>
```

Public Attributes

- unsigned [col](#)
For internal use only.
- char [str](#) [FL_CHART_LABEL_MAX+1]
For internal use only.
- float [val](#)
For internal use only.

30.12.1 Detailed Description

For internal use only.

30.12.2 Member Data Documentation

30.12.2.1 unsigned FL_CHART_ENTRY::col

For internal use only.

30.12.2.2 char FL_CHART_ENTRY::str[FL_CHART_LABEL_MAX+1]

For internal use only.

30.12.2.3 float FL_CHART_ENTRY::val

For internal use only.

The documentation for this struct was generated from the following file:

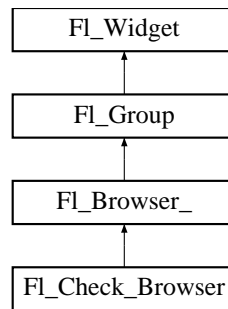
- Fl_Chart.H

30.13 Fl_Check_Browser Class Reference

The [Fl_Check_Browser](#) widget displays a scrolling list of text lines that may be selected and/or checked by the user.

```
#include <Fl_Check_Browser.H>
```

Inheritance diagram for Fl_Check_Browser::



Public Member Functions

- [int add](#) (const char *s, int b)
*See int [Fl_Check_Browser::add\(char *s\)](#).*
- [int add](#) (const char *s)
*See int [Fl_Check_Browser::add\(char *s\)](#).*
- [int add](#) (char *s, int b)
*See int [Fl_Check_Browser::add\(char *s\)](#).*
- [int add](#) (char *s)
Add a new unchecked line to the end of the browser.
- [void check_all](#) ()
Sets all the items checked.
- [void check_none](#) ()
Sets all the items unchecked.
- [void checked](#) (int item, int b)
Sets the check status of item item to b.
- [int checked](#) (int item) const
Gets the current status of item item.
- [void clear](#) ()
Remove every item from the browser.
- [Fl_Check_Browser](#) (int x, int y, int w, int h, const char *l=0)
The constructor makes an empty browser.

- `int nchecked () const`
Returns how many items are currently checked.
- `int nitems () const`
Returns how many lines are in the browser.
- `int remove (int item)`
*Remove line *n* and make the browser one line shorter.*
- `void set_checked (int item)`
Equivalent to `Fl_Check_Browser::checked(item, 1)`.
- `char * text (int item) const`
*Return a pointer to an internal buffer holding item *item*'s text.*
- `int value () const`
Returns the index of the currently selected item.
- `~Fl_Check_Browser ()`
The destructor deletes all list items and destroys the browser.

Protected Member Functions

- `int handle (int)`
Handles an event within the normal widget bounding box.

30.13.1 Detailed Description

The `Fl_Check_Browser` widget displays a scrolling list of text lines that may be selected and/or checked by the user.

30.13.2 Constructor & Destructor Documentation

30.13.2.1 `Fl_Check_Browser::Fl_Check_Browser (int X, int Y, int W, int H, const char * l = 0)`

The constructor makes an empty browser.

30.13.2.2 `Fl_Check_Browser::~~Fl_Check_Browser ()` `[inline]`

The destructor deletes all list items and destroys the browser.

30.13.3 Member Function Documentation

30.13.3.1 `int FL_Check_Browser::add (char * s)`

Add a new unchecked line to the end of the browser.

The text is copied using the `strdup()` function. It may also be `NULL` to make a blank line. The second form can set the item checked.

30.13.3.2 `void FL_Check_Browser::check_all ()`

Sets all the items checked.

30.13.3.3 `void FL_Check_Browser::check_none ()`

Sets all the items unchecked.

30.13.3.4 `void FL_Check_Browser::checked (int i, int b)`

Sets the check status of item `item` to `b`.

30.13.3.5 `int FL_Check_Browser::checked (int i) const`

Gets the current status of item `item`.

30.13.3.6 `void FL_Check_Browser::clear ()`

Remove every item from the browser.

Reimplemented from [FL_Group](#).

30.13.3.7 `int FL_Check_Browser::handle (int event)` `[protected, virtual]`

Handles an event within the normal widget bounding box.

Reimplemented from [FL_Browser_](#).

30.13.3.8 `int FL_Check_Browser::nchecked () const` `[inline]`

Returns how many items are currently checked.

30.13.3.9 `int FL_Check_Browser::nitems () const` `[inline]`

Returns how many lines are in the browser.

The last line number is equal to this.

30.13.3.10 int Fl_Check_Browser::remove (int *item*)

Remove line *n* and make the browser one line shorter.

Returns the number of lines left in the browser.

30.13.3.11 void Fl_Check_Browser::set_checked (int *item*) [inline]

Equivalent to `Fl_Check_Browser::checked(item, 1)`.

30.13.3.12 char * Fl_Check_Browser::text (int *i*) const

Return a pointer to an internal buffer holding item *item*'s text.

30.13.3.13 int Fl_Check_Browser::value () const

Returns the index of the currently selected item.

The documentation for this class was generated from the following files:

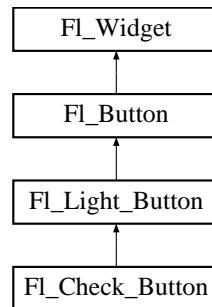
- `Fl_Check_Browser.H`
- `Fl_Check_Browser.cxx`

30.14 Fl_Check_Button Class Reference

A button with an "checkmark" to show its status.

```
#include <Fl_Check_Button.H>
```

Inheritance diagram for Fl_Check_Button::



Public Member Functions

- [Fl_Check_Button](#) (int X, int Y, int W, int H, const char *L=0)

Creates a new [Fl_Check_Button](#) widget using the given position, size and label string.

30.14.1 Detailed Description

A button with an "checkmark" to show its status.

Figure 30.3: Fl_Check_Button

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for [type\(\)](#) and [when\(\)](#).

The [Fl_Check_Button](#) subclass displays its "ON" state by showing a "checkmark" rather than drawing itself pushed in.

Todo

Refactor `Fl_Check_Button` doxygen comments (add `color()` info etc?)

Todo

Generate `Fl_Check_Button.gif` with visible checkmark.

30.14.2 Constructor & Destructor Documentation

30.14.2.1 `Fl_Check_Button::Fl_Check_Button (int X, int Y, int W, int H, const char * L = 0)`

Creates a new `Fl_Check_Button` widget using the given position, size and label string.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

The documentation for this class was generated from the following files:

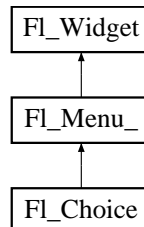
- `Fl_Check_Button.H`
- `Fl_Check_Button.cxx`

30.15 Fl_Choice Class Reference

A button that is used to pop up a menu.

```
#include <Fl_Choice.H>
```

Inheritance diagram for Fl_Choice::



Public Member Functions

- [Fl_Choice](#) (int X, int Y, int W, int H, const char *L=0)
Create a new [Fl_Choice](#) widget using the given position, size and label string.
- int [handle](#) (int)
Handles the specified event.
- int [value](#) (const [Fl_Menu_Item](#) *v)
Sets the currently selected value using a pointer to menu item.
- int [value](#) (int v)
Sets the currently selected value using the index into the menu item array.
- int [value](#) () const
Gets the index of the last item chosen by the user.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.15.1 Detailed Description

A button that is used to pop up a menu.

This is a button that, when pushed, pops up a menu (or hierarchy of menus) defined by an array of [Fl_Menu_Item](#) objects. Motif calls this an `OptionButton`.

The only difference between this and a [Fl_Menu_Button](#) is that the name of the most recent chosen menu item is displayed inside the box, while the label is displayed outside the box. However, since the use of this is most often to control a single variable rather than do individual callbacks, some of the [Fl_Menu_Button](#) methods are redescribed here in those terms.

When the user picks an item off the menu the `value()` is set to that item and then the item's callback is done with the menu_button as the `Fl_Widget*` argument. If the item does not have a callback the menu_button's callback is done instead.

All three mouse buttons pop up the menu. The Forms behavior of the first two buttons to increment/decrement the choice is not implemented. This could be added with a subclass, however.

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the `label()`. See `Fl_Button::shortcut(int s)` for a description of this.

Typing the `shortcut()` of any of the items will do exactly the same as when you pick the item with the mouse. The '&' character in item names are only looked at when the menu is popped up, however.

Figure 30.4: Fl_Choice

Todo

Refactor the doxygen comments for `Fl_Choice changed()` documentation.

- `int Fl_Widget::changed() const` This value is true the user picks a different value. *It is turned off by `value()` and just before doing a callback (the callback can turn it back on if desired).*
- `void Fl_Widget::set_changed()` This method sets the `changed()` flag.
- `void Fl_Widget::clear_changed()` This method clears the `changed()` flag.
- `Fl_Boxtype Fl_Choice::down_box() const` Gets the current down box, which is used when the menu is popped up. The default down box type is `FL_DOWN_BOX`.
- `void Fl_Choice::down_box(Fl_Boxtype b)` Sets the current down box type to `b`.

30.15.2 Constructor & Destructor Documentation

30.15.2.1 Fl_Choice::Fl_Choice (int X, int Y, int W, int H, const char * L = 0)

Create a new `Fl_Choice` widget using the given position, size and label string.

The default boxtype is `FL_UP_BOX`.

The constructor sets `menu()` to `NULL`. See `Fl_Menu_` for the methods to set or change the menu.

Parameters:

- ← `X,Y,W,H` position and size of the widget
- ← `L` widget label, default is no label

30.15.3 Member Function Documentation

30.15.3.1 void FL_Choice::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `FL_Widget*` and call the virtual [draw\(\)](#) method.

```
FL_Widget *s = &scroll;           // scroll is an embedded FL_Scrollbar
s->draw();                         // calls FL_Scrollbar::draw()
```

Implements [FL_Widget](#).

30.15.3.2 int FL_Choice::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

30.15.3.3 int FL_Choice::value (const FL_Menu_Item * v)

Sets the currently selected value using a pointer to menu item.

Changing the selected value causes a [redraw\(\)](#).

Parameters:

← *v* pointer to menu item in the menu item array.

Returns:

non-zero if the new value is different to the old one.

Reimplemented from [Fl_Menu_](#).

30.15.3.4 int Fl_Choice::value (int v)

Sets the currently selected value using the index into the menu item array.

Changing the selected value causes a [redraw\(\)](#).

Parameters:

← v index of value in the menu item array.

Returns:

non-zero if the new value is different to the old one.

Reimplemented from [Fl_Menu_](#).

30.15.3.5 int Fl_Choice::value () const [inline]

Gets the index of the last item chosen by the user.

The index is zero initially.

Reimplemented from [Fl_Menu_](#).

The documentation for this class was generated from the following files:

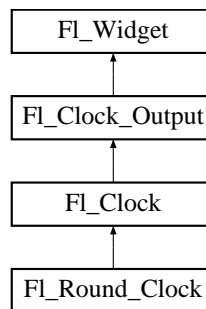
- [Fl_Choice.H](#)
- [Fl_Choice.cxx](#)

30.16 Fl_Clock Class Reference

This widget provides a round analog clock display.

```
#include <Fl_Clock.H>
```

Inheritance diagram for Fl_Clock::



Public Member Functions

- [Fl_Clock](#) (uchar t, int X, int Y, int W, int H, const char *L)

Create an [Fl_Clock](#) widget using the given boxtype, position, size, and label string.

- [Fl_Clock](#) (int X, int Y, int W, int H, const char *L=0)

Create an [Fl_Clock](#) widget using the given position, size, and label string.

- int [handle](#) (int)

Handles the specified event.

- void [update](#) ()

Undefined.

- [~Fl_Clock](#) ()

The destructor removes the clock.

30.16.1 Detailed Description

This widget provides a round analog clock display.

[Fl_Clock](#) is provided for Forms compatibility. It installs a 1-second timeout callback using [Fl::add_timeout\(\)](#). You can choose the rounded or square type of the clock with [type\(\)](#), see below.

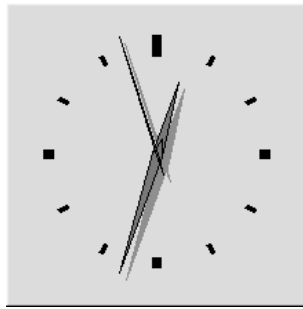


Figure 30.5: FL_SQUARE_CLOCK type

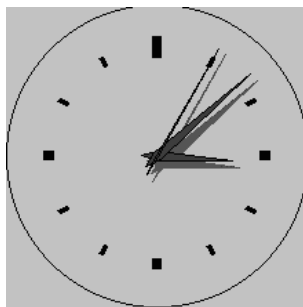


Figure 30.6: FL_ROUND_CLOCK type

30.16.2 Constructor & Destructor Documentation

30.16.2.1 `Fl_Clock::Fl_Clock (int X, int Y, int W, int H, const char * L = 0)`

Create an [Fl_Clock](#) widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.16.2.2 `Fl_Clock::Fl_Clock (uchar t, int X, int Y, int W, int H, const char * L)`

Create an [Fl_Clock](#) widget using the given boxtype, position, size, and label string.

Parameters:

- ← *t* boxtype
- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.16.3 Member Function Documentation

30.16.3.1 `int FL_Clock::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

30.16.3.2 `void FL_Clock::update ()`

Undefined.

Todo

Find [FL_Clock::update\(\)](#) implementation, if any, and document it.

The documentation for this class was generated from the following files:

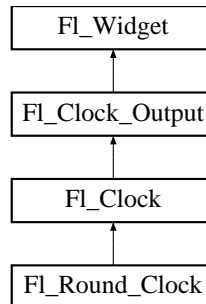
- [FL_Clock.H](#)
- [FL_Clock.cxx](#)

30.17 Fl_Clock_Output Class Reference

This widget can be used to display a program-supplied time.

```
#include <Fl_Clock.H>
```

Inheritance diagram for Fl_Clock_Output::



Public Member Functions

- **Fl_Clock_Output** (int X, int Y, int W, int H, const char *L=0)
*Create a new **Fl_Clock_Output** widget with the given position, size and label.*
- int **hour** () const
Returns the displayed hour (0 to 23).
- int **minute** () const
Returns the displayed minute (0 to 59).
- int **second** () const
Returns the displayed second (0 to 60, 60=leap second).
- **ulong value** () const
Returns the displayed time.
- void **value** (int H, int m, int s)
Set the displayed time.
- void **value** (**ulong** v)
Set the displayed time.

Protected Member Functions

- void **draw** (int X, int Y, int W, int H)
Draw clock with the given position and size.
- void **draw** ()
Draw clock with current position and size.

30.17.1 Detailed Description

This widget can be used to display a program-supplied time.

The time shown on the clock is not updated. To display the current time, use [FL_Clock](#) instead.

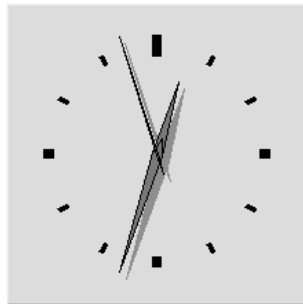


Figure 30.7: FL_SQUARE_CLOCK type

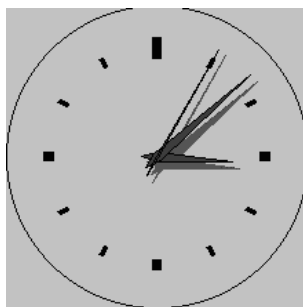


Figure 30.8: FL_ROUND_CLOCK type

30.17.2 Constructor & Destructor Documentation

30.17.2.1 FL_Clock_Output::FL_Clock_Output (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Create a new [FL_Clock_Output](#) widget with the given position, size and label.

The default boxtype is `FL_NO_BOX`.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.17.3 Member Function Documentation

30.17.3.1 void FL_Clock_Output::draw (int *X*, int *Y*, int *W*, int *H*) [protected]

Draw clock with the given position and size.

Parameters:

← *X,Y,W,H* position and size

30.17.3.2 int Fl_Clock_Output::hour () const [inline]

Returns the displayed hour (0 to 23).

See also:

[value\(\)](#), [minute\(\)](#), [second\(\)](#)

30.17.3.3 int Fl_Clock_Output::minute () const [inline]

Returns the displayed minute (0 to 59).

See also:

[value\(\)](#), [hour\(\)](#), [second\(\)](#)

30.17.3.4 int Fl_Clock_Output::second () const [inline]

Returns the displayed second (0 to 60, 60=leap second).

See also:

[value\(\)](#), [hour\(\)](#), [minute\(\)](#)

30.17.3.5 ulong Fl_Clock_Output::value () const [inline]

Returns the displayed time.

Returns the time in seconds since the UNIX epoch (January 1, 1970).

See also:

[value\(ulong\)](#)

30.17.3.6 void Fl_Clock_Output::value (int *H*, int *m*, int *s*)

Set the displayed time.

Set the time in hours, minutes, and seconds.

Parameters:

← *H,m,s* displayed time

See also:

[hour\(\)](#), [minute\(\)](#), [second\(\)](#)

30.17.3.7 void Fl_Clock_Output::value (ulong *v*)

Set the displayed time.

Set the time in seconds since the UNIX epoch (January 1, 1970).

Parameters:

← *v* seconds since epoch

See also:

[value\(\)](#)

The documentation for this class was generated from the following files:

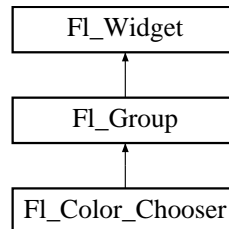
- Fl_Clock.H
- Fl_Clock.cxx

30.18 FL_Color_Chooser Class Reference

The [FL_Color_Chooser](#) widget provides a standard RGB color chooser.

```
#include <Fl_Color_Chooser.H>
```

Inheritance diagram for `FL_Color_Chooser`::



Public Member Functions

- `double b () const`
Returns the current blue value.
- `FL_Color_Chooser (int X, int Y, int W, int H, const char *L=0)`
Creates a new [FL_Color_Chooser](#) widget using the given position, size, and label string.
- `double g () const`
Returns the current green value.
- `int hsv (double H, double S, double V)`
Set the hsv values.
- `double hue () const`
Returns the current hue.
- `int mode ()`
Returns which [FL_Color_Chooser](#) variant is currently active.
- `double r () const`
Returns the current red value.
- `int rgb (double R, double G, double B)`
Sets the current rgb color values.
- `double saturation () const`
Returns the saturation.
- `double value () const`
Returns the value/brightness.

Static Public Member Functions

- static void `hsv2rgb` (double H, double S, double V, double &R, double &G, double &B)
This static method converts HSV colors to RGB colorspace.
- static void `rgb2hsv` (double R, double G, double B, double &H, double &S, double &V)
This static method converts RGB colors to HSV colorspace.

Related Functions

(Note that these are not member functions.)

- int `fl_color_chooser` (const char *name, uchar &r, uchar &g, uchar &b)
Pops up a window to let the user pick an arbitrary RGB color.
- int `fl_color_chooser` (const char *name, double &r, double &g, double &b)
Pops up a window to let the user pick an arbitrary RGB color.

30.18.1 Detailed Description

The `FL_Color_Chooser` widget provides a standard RGB color chooser.

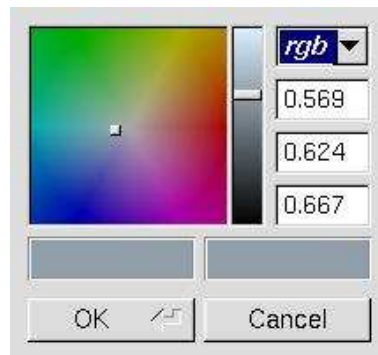


Figure 30.9: `fl_color_chooser()`

You can place any number of the widgets into a panel of your own design. The diagram shows the widget as part of a color chooser dialog created by the `fl_color_chooser()` function. The `FL_Color_Chooser` widget contains the hue box, value slider, and rgb input fields from the above diagram (it does not have the color chips or the Cancel or OK buttons). The callback is done every time the user changes the rgb value. It is not done if they move the hue control in a way that produces the *same* rgb value, such as when saturation or value is zero.

The `fl_color_chooser()` function pops up a window to let the user pick an arbitrary RGB color. They can pick the hue and saturation in the "hue box" on the left (hold down CTRL to just change the saturation), and the brightness using the vertical slider. Or they can type the 8-bit numbers into the RGB `FL_Value_Input` fields, or drag the mouse across them to adjust them. The pull-down menu lets the user set the input fields to show RGB, HSV, or 8-bit RGB (0 to 255).

`fl_color_chooser()` returns non-zero if the user picks ok, and updates the RGB values. If the user picks cancel or closes the window this returns zero and leaves RGB unchanged.

If you use the color chooser on an 8-bit screen, it will allocate all the available colors, leaving you no space to exactly represent the color the user picks! You can however use `fl_rectf()` to fill a region with a simulated color using dithering.

30.18.2 Constructor & Destructor Documentation

30.18.2.1 `Fl_Color_Chooser::Fl_Color_Chooser (int X, int Y, int W, int H, const char * L = 0)`

Creates a new `Fl_Color_Chooser` widget using the given position, size, and label string.

The recommended dimensions are 200x95. The color is initialized to black.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.18.3 Member Function Documentation

30.18.3.1 `double Fl_Color_Chooser::b () const` [inline]

Returns the current blue value.

$0 \leq b \leq 1$.

30.18.3.2 `double Fl_Color_Chooser::g () const` [inline]

Returns the current green value.

$0 \leq g \leq 1$.

30.18.3.3 `int Fl_Color_Chooser::hsv (double H, double S, double V)`

Set the hsv values.

The passed values are clamped (or for hue, modulus 6 is used) to get legal values. Does not do the call-back.

Parameters:

- ← *H,S,V* color components.

Returns:

- 1 if a new hsv value was set, 0 if the hsv value was the previous one.

30.18.3.4 `void Fl_Color_Chooser::hsv2rgb (double H, double S, double V, double & R, double & G, double & B)` [static]

This *static* method converts HSV colors to RGB colorspace.

Parameters:

← H, S, V color components

→ R, G, B color components

30.18.3.5 double Fl_Color_Chooser::hue () const [inline]

Returns the current hue.

$0 \leq \text{hue} < 6$. Zero is red, one is yellow, two is green, etc. *This value is convenient for the internal calculations - some other systems consider hue to run from zero to one, or from 0 to 360.*

30.18.3.6 double Fl_Color_Chooser::r () const [inline]

Returns the current red value.

$0 \leq r \leq 1$.

30.18.3.7 int Fl_Color_Chooser::rgb (double R, double G, double B)

Sets the current rgb color values.

Does not do the callback. Does not clamp (but out of range values will produce psychedelic effects in the hue selector).

Parameters:

← R, G, B color components.

Returns:

1 if a new rgb value was set, 0 if the rgb value was the previous one.

30.18.3.8 void Fl_Color_Chooser::rgb2hsv (double R, double G, double B, double &H, double &S, double &V) [static]

This *static* method converts RGB colors to HSV colorspace.

Parameters:

← R, G, B color components

→ H, S, V color components

30.18.3.9 double Fl_Color_Chooser::saturation () const [inline]

Returns the saturation.

$0 \leq \text{saturation} \leq 1$.

30.18.3.10 double Fl_Color_Chooser::value () const [inline]

Returns the value/brightness.

0 <= value <= 1.

The documentation for this class was generated from the following files:

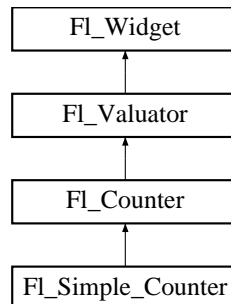
- [Fl_Color_Chooser.H](#)
- [Fl_Color_Chooser.cxx](#)

30.19 Fl_Counter Class Reference

Controls a single floating point value with button (or keyboard) arrows.

```
#include <Fl_Counter.H>
```

Inheritance diagram for Fl_Counter::



Public Member Functions

- **Fl_Counter** (int X, int Y, int W, int H, const char *L=0)
*Creates a new **Fl_Counter** widget using the given position, size, and label string.*
- int **handle** (int)
Handles the specified event.
- void **lstep** (double a)
Sets the increment for the large step buttons.
- double **step** ()
Returns the increment for normal step buttons.
- void **step** (double a)
Sets the increment for the normal step buttons.
- void **step** (double a, double b)
Sets the increments for the normal and large step buttons.
- void **textcolor** (unsigned s)
Sets the font color to s.
- **Fl_Color** **textcolor** () const
Gets the font color.
- void **textfont** (**Fl_Font** s)
Sets the text font to s.
- **Fl_Font** **textfont** () const
Gets the text font.

- void [textsize](#) ([Fl_Fontsize](#) s)
Sets the font size to s.
- [Fl_Fontsize textsize](#) () const
Gets the font size.
- [~Fl_Counter](#) ()
Destroys the valuator.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.19.1 Detailed Description

Controls a single floating point value with button (or keyboard) arrows.

Double arrows buttons achieve larger steps than simple arrows.

See also:

[Fl_Spinner](#) for [value](#) input with vertical [step](#) arrows.

Figure 30.10: [Fl_Counter](#)

Todo

Refactor the doxygen comments for [Fl_Counter type\(\)](#) documentation.

The type of an [Fl_Counter](#) object can be set using [type\(uchar t\)](#) to:

- `FL_NORMAL_COUNTER`: Displays a counter with 4 arrow buttons.
- `FL_SIMPLE_COUNTER`: Displays a counter with only 2 arrow buttons.

30.19.2 Constructor & Destructor Documentation

30.19.2.1 Fl_Counter::Fl_Counter (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates a new [Fl_Counter](#) widget using the given position, size, and label string.

The default type is FL_NORMAL_COUNTER.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.19.3 Member Function Documentation

30.19.3.1 void Fl_Counter::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.19.3.2 int Fl_Counter::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

- ← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [FL_Widget](#).

30.19.3.3 void FL_Counter::lstep (double *a*) [inline]

Sets the increment for the large step buttons.

The default value is 1.0.

Parameters:

← *a* large step increment.

30.19.3.4 void FL_Counter::step (double *a*) [inline]

Sets the increment for the normal step buttons.

Parameters:

← *a* normal step increment.

Reimplemented from [FL_Valuator](#).

30.19.3.5 void FL_Counter::step (double *a*, double *b*) [inline]

Sets the increments for the normal and large step buttons.

Parameters:

← *a,b* normal and large step increments.

The documentation for this class was generated from the following files:

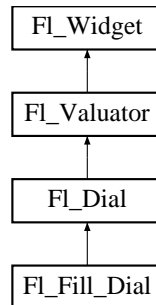
- FL_Counter.H
- FL_Counter.cxx

30.20 Fl_Dial Class Reference

The [Fl_Dial](#) widget provides a circular dial to control a single floating point value.

```
#include <Fl_Dial.H>
```

Inheritance diagram for Fl_Dial::



Public Member Functions

- void [angle1](#) (short a)
See short [angle1\(\)](#) const.
- short [angle1](#) () const
Sets Or gets the angles used for the minimum and maximum values.
- void [angle2](#) (short a)
See short [angle1\(\)](#) const.
- short [angle2](#) () const
See short [angle1\(\)](#) const.
- void [angles](#) (short a, short b)
See short [angle1\(\)](#) const.
- [Fl_Dial](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [Fl_Dial](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Allow subclasses to handle event based on current position and size.

Protected Member Functions

- void [draw](#) ()
Draws dial at current position and size.
- void [draw](#) (int X, int Y, int W, int H)
Draws dial at given position and size.

- `int handle (int event, int X, int Y, int W, int H)`

Allows subclasses to handle event based on given position and size.

30.20.1 Detailed Description

The `FL_Dial` widget provides a circular dial to control a single floating point value.

Figure 30.11: `FL_Dial`

Use `type()` to set the type of the dial to:

- `FL_NORMAL_DIAL` - Draws a normal dial with a knob.
- `FL_LINE_DIAL` - Draws a dial with a line.
- `FL_FILL_DIAL` - Draws a dial with a filled arc.

30.20.2 Constructor & Destructor Documentation

30.20.2.1 `FL_Dial::FL_Dial (int X, int Y, int W, int H, const char * l = 0)`

Creates a new `FL_Dial` widget using the given position, size, and label string.

Creates a new `FL_Dial` widget using the given position, size, and label string.

The default type is `FL_NORMAL_DIAL`.

30.20.3 Member Function Documentation

30.20.3.1 `short FL_Dial::angle1 () const` `[inline]`

Sets Or gets the angles used for the minimum and maximum values.

The default values are 45 and 315 (0 degrees is straight down and the angles progress clockwise). Normally `angle1` is less than `angle2`, but if you reverse them the dial moves counter-clockwise.

30.20.3.2 void Fl_Dial::draw (int *X*, int *Y*, int *W*, int *H*) [protected]

Draws dial at given position and size.

Parameters:

← *X,Y,W,H* position and size

30.20.3.3 int Fl_Dial::handle (int *event*, int *X*, int *Y*, int *W*, int *H*) [protected]

Allows subclasses to handle event based on given position and size.

Parameters:

← *event,X,Y,W,H* event to handle, related position and size.

The documentation for this class was generated from the following files:

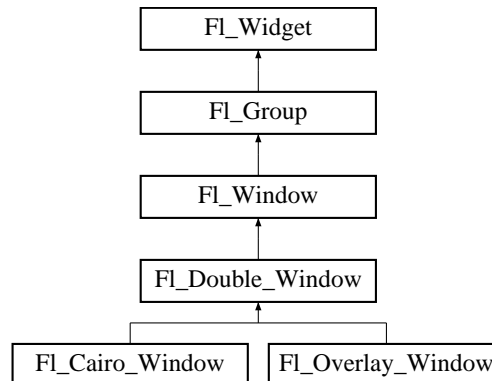
- Fl_Dial.H
- Fl_Dial.cxx

30.21 Fl_Double_Window Class Reference

The [Fl_Double_Window](#) provides a double-buffered window.

```
#include <Fl_Double_Window.H>
```

Inheritance diagram for `Fl_Double_Window`:



Public Member Functions

- [Fl_Double_Window](#) (int X, int Y, int W, int H, const char *l=0)
*See `Fl_Double_Window::Fl_Double_Window(int w, int h, const char *label = 0)`.*
- [Fl_Double_Window](#) (int W, int H, const char *l=0)
Creates a new `Fl_Double_Window` widget using the given position, size, and label (title) string.
- void [flush](#) ()
Forces the window to be redrawn.
- void [hide](#) ()
Remove the window from the screen.
- void [resize](#) (int, int, int, int)
Changes the size and position of the window.
- void [show](#) (int a, char **b)
See virtual void `Fl_Window::show()`.
- void [show](#) ()
Put the window on the screen.
- [~Fl_Double_Window](#) ()
The destructor also deletes all the children.

Protected Member Functions

- void [flush](#) (int eraseoverlay)

Forces the window to be redrawn.

Protected Attributes

- char [force_doublebuffering_](#)

Force double buffering, even if the OS already buffers windows (overlays need that on MacOS and Windows2000).

30.21.1 Detailed Description

The [FL_Double_Window](#) provides a double-buffered window.

If possible this will use the X double buffering extension (Xdbe). If not, it will draw the window data into an off-screen pixmap, and then copy it to the on-screen window.

It is highly recommended that you put the following code before the first [show\(\)](#) of *any* window in your program:

```
FL::visual (FL_DOUBLE|FL_INDEX)
```

This makes sure you can use Xdbe on servers where double buffering does not exist for every visual.

30.21.2 Constructor & Destructor Documentation

30.21.2.1 FL_Double_Window::~~FL_Double_Window ()

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code.

30.21.3 Member Function Documentation

30.21.3.1 void FL_Double_Window::flush (int *eraseoverlay*) [protected]

Forces the window to be redrawn.

Parameters:

← *eraseoverlay* non-zero to erase overlay, zero to ignore

[FL_Overlay_Window](#) relies on [flush\(1\)](#) copying the back buffer to the front everywhere, even if [damage\(\)](#) == 0, thus erasing the overlay, and leaving the clip region set to the entire window.

30.21.3.2 void FL_Double_Window::hide () [virtual]

Remove the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Overlay_Window](#).

30.21.3.3 void FL_Double_Window::resize (int, int, int, int) [virtual]

Changes the size and position of the window.

If [shown\(\)](#) is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If [shown\(\)](#) is false, the size and position are used when [show\(\)](#) is called. See [FL_Group](#) for the effect of resizing on the child widgets.

You can also call the [FL_Widget](#) methods [size\(x,y\)](#) and [position\(w,h\)](#), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Overlay_Window](#).

30.21.3.4 void FL_Double_Window::show () [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Overlay_Window](#).

The documentation for this class was generated from the following files:

- [FL_Double_Window.H](#)
- [FL_Double_Window.cxx](#)

30.22 Fl_End Class Reference

This is a dummy class that allows you to end a [Fl_Group](#) in a constructor list of a class:.

```
#include <Fl_Group.H>
```

Public Member Functions

- [Fl_End](#) ()

All it does is calling [Fl_Group::current\(\)->end\(\)](#).

30.22.1 Detailed Description

This is a dummy class that allows you to end a [Fl_Group](#) in a constructor list of a class:.

```
class MyClass {
    Fl_Group group;
    Fl_Button button_in_group;
    Fl_End end;
    Fl_Button button_outside_group;
    MyClass();
};
MyClass::MyClass() :
    group(10,10,100,100),
    button_in_group(20,20,60,30),
    end(),
    button_outside_group(10,120,60,30)
{ }
```

The documentation for this class was generated from the following file:

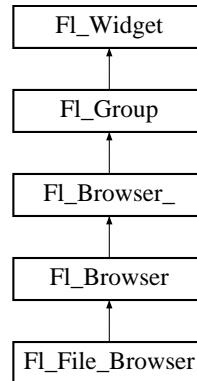
- [Fl_Group.H](#)

30.23 Fl_File_Browser Class Reference

The [Fl_File_Browser](#) widget displays a list of filenames, optionally with file-specific icons.

```
#include <Fl_File_Browser.H>
```

Inheritance diagram for Fl_File_Browser::



Public Types

- enum { **FILES**, **DIRECTORIES** }

Public Member Functions

- void [filetype](#) (int t)
Sets or gets the file browser type, FILES or DIRECTORIES.
- int [filetype](#) () const
Sets or gets the file browser type, FILES or DIRECTORIES.
- const char * [filter](#) () const
Sets or gets the filename filter.
- void [filter](#) (const char *pattern)
Sets or gets the filename filter.
- [Fl_File_Browser](#) (int, int, int, int, const char * = 0)
The constructor creates the [Fl_File_Browser](#) widget at the specified position and size.
- void [iconsize](#) (uchar s)
Sets or gets the size of the icons.
- uchar [iconsize](#) () const
Sets or gets the size of the icons.
- int [load](#) (const char *directory, [Fl_File_Sort_F](#) *sort = fl_numericsort)
Loads the specified directory into the browser.

- void [textsize](#) ([Fl_Fontsize](#) s)
Sets the default text size to size s.
- [Fl_Fontsize](#) [textsize](#) () const
Gets the default text size for the lines in the browser.

30.23.1 Detailed Description

The [Fl_File_Browser](#) widget displays a list of filenames, optionally with file-specific icons.

30.23.2 Constructor & Destructor Documentation

30.23.2.1 [Fl_File_Browser::Fl_File_Browser](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

The constructor creates the [Fl_File_Browser](#) widget at the specified position and size.

The destructor destroys the widget and frees all memory that has been allocated.

30.23.3 Member Function Documentation

30.23.3.1 void [Fl_File_Browser::filetype](#) (int *t*) [[inline](#)]

Sets or gets the file browser type, FILES or DIRECTORIES.

When set to FILES, both files and directories are shown. Otherwise only directories are shown.

30.23.3.2 int [Fl_File_Browser::filetype](#) () const [[inline](#)]

Sets or gets the file browser type, FILES or DIRECTORIES.

When set to FILES, both files and directories are shown. Otherwise only directories are shown.

30.23.3.3 const char* [Fl_File_Browser::filter](#) () const [[inline](#)]

Sets or gets the filename filter.

The pattern matching uses the [fl_filename_match\(\)](#) function in FLTK.

30.23.3.4 void [Fl_File_Browser::filter](#) (const char * *pattern*)

Sets or gets the filename filter.

The pattern matching uses the [fl_filename_match\(\)](#) function in FLTK.

30.23.3.5 void [Fl_File_Browser::iconsize](#) (uchar *s*) [[inline](#)]

Sets or gets the size of the icons.

The default size is 20 pixels.

30.23.3.6 `uchar Fl_File_Browser::iconsize () const` `[inline]`

Sets or gets the size of the icons.

The default size is 20 pixels.

30.23.3.7 `int Fl_File_Browser::load (const char * directory, Fl_File_Sort_F * sort = fl_numericsort)`

Loads the specified directory into the browser.

If icons have been loaded then the correct icon is associated with each file in the list.

The sort argument specifies a sort function to be used with `fl_filename_list()`.

The documentation for this class was generated from the following files:

- `Fl_File_Browser.H`
- `Fl_File_Browser.cxx`

30.24 FL_File_Chooser Class Reference

The [FL_File_Chooser](#) widget displays a standard file selection dialog that supports various selection modes.

Public Types

- enum { **SINGLE** = 0, **MULTI** = 1, **CREATE** = 2, **DIRECTORY** = 4 }

Public Member Functions

- [FL_Widget](#) * [add_extra](#) ([FL_Widget](#) *gr)
Adds extra widget at the bottom of [FL_File_Chooser](#) window.
- void [callback](#) (void(*cb)([FL_File_Chooser](#) *, void *), void *d=0)
Sets the file chooser callback cb and associated data d.
- [FL_Color](#) [color](#) ()
Sets or gets the background color of the [FL_File_Browser](#) list.
- void [color](#) ([FL_Color](#) c)
Sets or gets the background color of the [FL_File_Browser](#) list.
- int [count](#) ()
Returns the number of selected files.
- char * [directory](#) ()
Sets or gets the current directory.
- void [directory](#) (const char *d)
Sets or gets the current directory.
- const char * [filter](#) ()
*See void [filter](#)(const char *pattern).*
- void [filter](#) (const char *p)
Sets or gets the current filename filter patterns.
- void [filter_value](#) (int f)
Sets or gets the current filename filter selection.
- int [filter_value](#) ()
Sets or gets the current filename filter selection.
- [FL_File_Chooser](#) (const char *d, const char *p, int t, const char *title)
The constructor creates the [FL_File_Chooser](#) dialog shown.
- void [hide](#) ()
Hides the [FL_File_Chooser](#) window.

- `uchar iconsize ()`
Sets or gets the size of the icons in the [FL_File_Browser](#).
- `void iconsize (uchar s)`
Sets or gets the size of the icons in the [FL_File_Browser](#).
- `const char * label ()`
Sets or gets the title bar text for the [FL_File_Chooser](#).
- `void label (const char *l)`
Sets or gets the title bar text for the [FL_File_Chooser](#).
- `const char * ok_label ()`
Sets or gets the label for the "ok" button in the [FL_File_Chooser](#).
- `void ok_label (const char *l)`
Sets or gets the label for the "ok" button in the [FL_File_Chooser](#).
- `int preview () const`
Returns the current state of the preview box.
- `void preview (int e)`
Enable or disable the preview tile.
- `void rescan ()`
Reloads the current directory in the [FL_File_Browser](#).
- `void rescan_keep_filename ()`
Rescan the current directory without clearing the filename, then select the file if it is in the list.
- `void show ()`
Shows the [FL_File_Chooser](#) window.
- `int shown ()`
Returns non-zero if the file chooser main window [show\(\)](#) has been called (but not [hide\(\)](#) see [FL_Window::shown\(\)](#)).
- `FL_Color textcolor ()`
Sets or gets the current [FL_File_Browser](#) text color.
- `void textcolor (FL_Color c)`
Sets or gets the current [FL_File_Browser](#) text color.
- `FL_Font textfont ()`
Sets or gets the current [FL_File_Browser](#) text font.
- `void textfont (FL_Font f)`
Sets or gets the current [FL_File_Browser](#) text font.
- `FL_Fontsize textsize ()`

Sets or gets the current *FL_File_Browser* text size.

- void `textsize` (FL_Fontsize s)
Sets or gets the current *FL_File_Browser* text size.
- int `type` ()
Sets or gets the current type of *FL_File_Chooser*.
- void `type` (int t)
Sets or gets the current type of *FL_File_Chooser*.
- void `user_data` (void *d)
Sets the file chooser user data d.
- void * `user_data` () const
Gets the file chooser user data d.
- void `value` (const char *filename)
Sets or gets the current value of the selected file.
- const char * `value` (int f=1)
See const char *value(const char *pathname).
- int `visible` ()
Returns 1 if the *FL_File_Chooser* window is visible.
- ~*FL_File_Chooser* ()
Destroys the widget and frees all memory used by it.

Public Attributes

- *FL_Button* * `newButton`
The "new directory" button is exported so that application developers can control the appearance and use.
- *FL_Check_Button* * `previewButton`
The "preview" button is exported so that application developers can control the appearance and use.

Static Public Attributes

- static const char * `add_favorites_label` = "Add to Favorites"
[standard text may be customized at run-time]
- static const char * `all_files_label` = "All Files (*)"
[standard text may be customized at run-time]
- static const char * `custom_filter_label` = "Custom Filter"
[standard text may be customized at run-time]

- static const char * [existing_file_label](#) = "Please choose an existing file!"
[standard text may be customized at run-time]
- static const char * [favorites_label](#) = "Favorites"
[standard text may be customized at run-time]
- static const char * [filename_label](#) = "Filename:"
[standard text may be customized at run-time]
- static const char * [filesystems_label](#) = "File Systems"
[standard text may be customized at run-time]
- static const char * [manage_favorites_label](#) = "Manage Favorites"
[standard text may be customized at run-time]
- static const char * [new_directory_label](#) = "New Directory?"
[standard text may be customized at run-time]
- static const char * [new_directory_tooltip](#) = "Create a new directory."
[standard text may be customized at run-time]
- static const char * [preview_label](#) = "Preview"
[standard text may be customized at run-time]
- static const char * [save_label](#) = "Save"
[standard text may be customized at run-time]
- static const char * [show_label](#) = "Show:"
[standard text may be customized at run-time]
- static [Fl_File_Sort_F](#) * [sort](#) = fl_numericsort
the sort function that is used when loading the contents of a directory.

Related Functions

(Note that these are not member functions.)

- char * [fl_dir_chooser](#) (const char *message, const char *fname, int relative)
- char * [fl_file_chooser](#) (const char *message, const char *pat, const char *fname, int relative)
- void [fl_file_chooser_callback](#) (void(*cb)(const char *))
- void [fl_file_chooser_ok_label](#) (const char *l)

30.24.1 Detailed Description

The [Fl_File_Chooser](#) widget displays a standard file selection dialog that supports various selection modes.

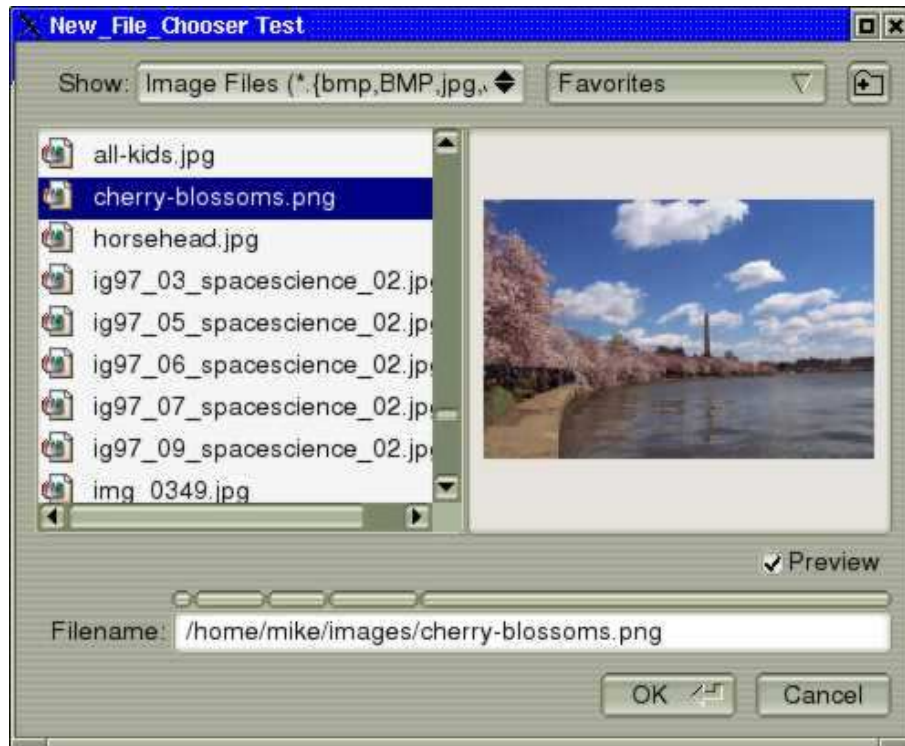


Figure 30.12: Fl_File_Chooser

The [Fl_File_Chooser](#) class also exports several static values that may be used to localize or customize the appearance of all file chooser dialogs:

Member	Default value
add_favorites_label	"Add to Favorites"
all_files_label	"All Files (*)"
custom_filter_label	"Custom Filter"
existing_file_label	"Please choose an existing file!"
favorites_label	"Favorites"
filename_label	"Filename:"
filesystems_label	"My Computer" (WIN32) "File Systems" (all others)
manage_favorites_label	"Manage Favorites"
new_directory_label	"New Directory?"
new_directory_tooltip	"Create a new directory."
preview_label	"Preview"
save_label	"Save"
show_label	"Show:"
sort	fl_numericsort

The [Fl_File_Chooser::sort](#) member specifies the sort function that is used when loading the contents of a directory and can be customized at run-time.

The [FL_File_Chooser](#) class also exports the [FL_File_Chooser::newButton](#) and [FL_File_Chooser::previewButton](#) widgets so that application developers can control their appearance and use. For more complex customization, consider copying the FLTK file chooser code and changing it accordingly.

30.24.2 Constructor & Destructor Documentation

30.24.2.1 [FL_File_Chooser::FL_File_Chooser](#) (const char * *pathname*, const char * *pattern*, int *type*, const char * *title*)

The constructor creates the [FL_File_Chooser](#) dialog shown.

The *pathname* argument can be a directory name or a complete file name (in which case the corresponding file is highlighted in the list and in the filename input field.)

The *pattern* argument can be a NULL string or "*" to list all files, or it can be a series of descriptions and filter strings separated by tab characters (\t). The format of filters is either "Description text (patterns)" or just "patterns". A file chooser that provides filters for HTML and image files might look like:

```
"HTML Files (*.html)\tImage Files (*.{bmp,gif,jpg,png})"
```

The file chooser will automatically add the "All Files (*)" pattern to the end of the string you pass if you do not provide one. The first filter in the string is the default filter.

See the FLTK documentation on [fl_filename_match\(\)](#) for the kinds of pattern strings that are supported.

The *type* argument can be one of the following:

- SINGLE - allows the user to select a single, existing file.
- MULTI - allows the user to select one or more existing files.
- CREATE - allows the user to select a single, existing file or specify a new filename.
- DIRECTORY - allows the user to select a single, existing directory.

The *title* argument is used to set the title bar text for the [FL_File_Chooser](#) window.

30.24.2.2 [FL_File_Chooser::~~FL_File_Chooser](#) ()

Destroys the widget and frees all memory used by it.

30.24.3 Member Function Documentation

30.24.3.1 [FL_Widget * FL_File_Chooser::add_extra](#) (FL_Widget * *gr*)

Adds extra widget at the bottom of [FL_File_Chooser](#) window.

Returns pointer for previous extra widget or NULL if not set previously. If argument is NULL only remove previous extra widget.

Note:

[FL_File_Chooser](#) does **not** delete extra widget in destructor! To prevent memory leakage, don't forget to delete unused extra widgets

30.24.3.2 `Fl_Color Fl_File_Chooser::color ()`

Sets or gets the background color of the [Fl_File_Browser](#) list.

30.24.3.3 `void Fl_File_Chooser::color (Fl_Color c)`

Sets or gets the background color of the [Fl_File_Browser](#) list.

30.24.3.4 `int Fl_File_Chooser::count ()`

Returns the number of selected files.

30.24.3.5 `const char * Fl_File_Chooser::directory ()`

Sets or gets the current directory.

30.24.3.6 `void Fl_File_Chooser::directory (const char * pathname)`

Sets or gets the current directory.

30.24.3.7 `void Fl_File_Chooser::filter (const char * pattern)`

Sets or gets the current filename filter patterns.

The filter patterns use [fl_filename_match\(\)](#). Multiple patterns can be used by separating them with tabs, like `"*.jpg\t*.png\t*.gif\t*"`. In addition, you can provide human-readable labels with the patterns inside parenthesis, like `"JPEG Files (*.jpg)\tPNG Files (*.png)\tGIF Files (*.gif)\tAll Files (*)"` .

Use `filter(NULL)` to show all files.

30.24.3.8 `void Fl_File_Chooser::filter_value (int f)`

Sets or gets the current filename filter selection.

30.24.3.9 `int Fl_File_Chooser::filter_value ()`

Sets or gets the current filename filter selection.

30.24.3.10 `void Fl_File_Chooser::hide ()`

Hides the [Fl_File_Chooser](#) window.

30.24.3.11 `uchar Fl_File_Chooser::iconsize ()`

Sets or gets the size of the icons in the [Fl_File_Browser](#).

By default the icon size is set to 1.5 times the `textsize()`.

30.24.3.12 void Fl_File_Chooser::iconsize (uchar *s*)

Sets or gets the size of the icons in the [Fl_File_Browser](#).

By default the icon size is set to 1.5 times the [textsize\(\)](#).

30.24.3.13 const char * Fl_File_Chooser::label ()

Sets or gets the title bar text for the [Fl_File_Chooser](#).

30.24.3.14 void Fl_File_Chooser::label (const char * *l*)

Sets or gets the title bar text for the [Fl_File_Chooser](#).

30.24.3.15 int Fl_File_Chooser::preview () const [inline]

Returns the current state of the preview box.

30.24.3.16 void Fl_File_Chooser::preview (int *e*)

Enable or disable the preview tile.

1 = enable preview, 0 = disable preview.

30.24.3.17 void Fl_File_Chooser::rescan ()

Reloads the current directory in the [Fl_File_Browser](#).

30.24.3.18 void Fl_File_Chooser::show ()

Shows the [Fl_File_Chooser](#) window.

30.24.3.19 Fl_Color Fl_File_Chooser::textcolor ()

Sets or gets the current [Fl_File_Browser](#) text color.

30.24.3.20 void Fl_File_Chooser::textcolor (Fl_Color *c*)

Sets or gets the current [Fl_File_Browser](#) text color.

30.24.3.21 Fl_Font Fl_File_Chooser::textfont ()

Sets or gets the current [Fl_File_Browser](#) text font.

30.24.3.22 void Fl_File_Chooser::textfont (Fl_Font *f*)

Sets or gets the current [Fl_File_Browser](#) text font.

30.24.3.23 `Fl_Fontsize Fl_File_Chooser::textsize ()`

Sets or gets the current [Fl_File_Browser](#) text size.

30.24.3.24 `void Fl_File_Chooser::textsize (Fl_Fontsize s)`

Sets or gets the current [Fl_File_Browser](#) text size.

30.24.3.25 `int Fl_File_Chooser::type ()`

Sets or gets the current type of [Fl_File_Chooser](#).

30.24.3.26 `void Fl_File_Chooser::type (int t)`

Sets or gets the current type of [Fl_File_Chooser](#).

30.24.3.27 `void Fl_File_Chooser::value (const char * pathname)`

Sets or gets the current value of the selected file.

In the second form, *file* is a 1-based index into a list of file names. The number of selected files is returned by [Fl_File_Chooser::count\(\)](#).

This sample code loops through all selected files:

```
// Get list of filenames user selected from a MULTI chooser
for ( int t=1; t<=chooser->count(); t++ ) {
    const char *filename = chooser->value(t);
    ...
}
```

30.24.3.28 `int Fl_File_Chooser::visible ()`

Returns 1 if the [Fl_File_Chooser](#) window is visible.

The documentation for this class was generated from the following files:

- `Fl_File_Chooser.H`
- `Fl_File_Chooser.cxx`
- `Fl_File_Chooser2.cxx`
- `fl_file_dir.cxx`

30.25 FL_File_Icon Class Reference

The [FL_File_Icon](#) class manages icon images that can be used as labels in other widgets and as icons in the FileBrowser widget.

```
#include <FL_File_Icon.H>
```

Public Types

- enum {
 ANY, PLAIN, FIFO, DEVICE,
 LINK, DIRECTORY }
- enum {
 END, COLOR, LINE, CLOSEDLINE,
 POLYGON, OUTLINEPOLYGON, VERTEX }

Public Member Functions

- short * [add](#) (short d)
Adds a keyword value to the icon array, returning a pointer to it.
- short * [add_color](#) ([FL_Color](#) c)
Adds a color value to the icon array, returning a pointer to it.
- short * [add_vertex](#) (float x, float y)
Adds a vertex value to the icon array, returning a pointer to it.
- short * [add_vertex](#) (int x, int y)
Adds a vertex value to the icon array, returning a pointer to it.
- void [clear](#) ()
Clears all icon data from the icon.
- void [draw](#) (int x, int y, int w, int h, [FL_Color](#) ic, int active=1)
Draws an icon in the indicated area.
- [FL_File_Icon](#) (const char *p, int t, int nd=0, short *d=0)
Creates a new [FL_File_Icon](#) with the specified information.
- void [label](#) ([FL_Widget](#) *w)
Applies the icon to the widget, registering the [FL_File_Icon](#) label type as needed.
- void [load](#) (const char *f)
Loads the specified icon image.
- int [load_fti](#) (const char *fti)
Loads an SGI icon file.
- int [load_image](#) (const char *i)

Load an image icon file from an image filename.

- [Fl_File_Icon * next \(\)](#)
Returns next file icon object.
- `const char * pattern ()`
Returns the filename matching pattern for the icon.
- `int size ()`
Returns the number of words of data used by the icon.
- `int type ()`
Returns the filetype associated with the icon, which can be one of the following:.
- `short * value ()`
Returns the data array for the icon.
- `~Fl_File_Icon ()`
The destructor destroys the icon and frees all memory that has been allocated for it.

Static Public Member Functions

- `static Fl_File_Icon * find (const char *filename, int filetype=ANY)`
Finds an icon that matches the given filename and file type.
- `static Fl_File_Icon * first ()`
Returns a pointer to the first icon in the list.
- `static void labeltype (const Fl_Label *o, int x, int y, int w, int h, Fl_Align a)`
Draw the icon label.
- `static void load_system_icons (void)`
Loads all system-defined icons.

30.25.1 Detailed Description

The [Fl_File_Icon](#) class manages icon images that can be used as labels in other widgets and as icons in the FileBrowser widget.

30.25.2 Constructor & Destructor Documentation

30.25.2.1 `Fl_File_Icon::Fl_File_Icon (const char *p, int t, int nd = 0, short *d = 0)`

Creates a new [Fl_File_Icon](#) with the specified information.

Parameters:

← *p* filename pattern

← *t* file type

← *nd* number of data values

← *d* data values

30.25.3 Member Function Documentation

30.25.3.1 `short * Fl_File_Icon::add (short d)`

Adds a keyword value to the icon array, returning a pointer to it.

Parameters:

← *d* data value

30.25.3.2 `short* Fl_File_Icon::add_color (Fl_Color c) [inline]`

Adds a color value to the icon array, returning a pointer to it.

Parameters:

← *c* color value

30.25.3.3 `short* Fl_File_Icon::add_vertex (float x, float y) [inline]`

Adds a vertex value to the icon array, returning a pointer to it.

The floating point version goes from 0.0 to 1.0. The origin (0.0) is in the lower-lefthand corner of the icon.

Parameters:

← *x,y* vertex coordinates

30.25.3.4 `short* Fl_File_Icon::add_vertex (int x, int y) [inline]`

Adds a vertex value to the icon array, returning a pointer to it.

The integer version accepts coordinates from 0 to 10000. The origin (0.0) is in the lower-lefthand corner of the icon.

Parameters:

← *x,y* vertex coordinates

30.25.3.5 `void Fl_File_Icon::clear () [inline]`

Clears all icon data from the icon.

30.25.3.6 void Fl_File_Icon::draw (int *x*, int *y*, int *w*, int *h*, Fl_Color *ic*, int *active* = 1)

Draws an icon in the indicated area.

Parameters:

- ← *x,y,w,h* position and size
- ← *ic* icon color
- ← *active* status, default is active [non-zero]

30.25.3.7 Fl_File_Icon * Fl_File_Icon::find (const char * *filename*, int *filetype* = ANY) [static]

Finds an icon that matches the given filename and file type.

Parameters:

- ← *filename* name of file
- ← *filetype* enumerated file type

Returns:

matching file icon or NULL

30.25.3.8 static Fl_File_Icon* Fl_File_Icon::first () [inline, static]

Returns a pointer to the first icon in the list.

30.25.3.9 void Fl_File_Icon::label (Fl_Widget * *w*)

Applies the icon to the widget, registering the [Fl_File_Icon](#) label type as needed.

Parameters:

- ← *w* widget for which this icon will become the label

30.25.3.10 void Fl_File_Icon::labeltype (const Fl_Label * *o*, int *x*, int *y*, int *w*, int *h*, Fl_Align *a*) [static]

Draw the icon label.

Parameters:

- ← *o* label data
- ← *x,y,w,h* position and size of label
- ← *a* label alignment [not used]

30.25.3.11 void Fl_File_Icon::load (const char **f*)

Loads the specified icon image.

The format is deduced from the filename.

Parameters:

← *f* filename

30.25.3.12 int Fl_File_Icon::load_fti (const char **fti*)

Loads an SGI icon file.

Parameters:

← *fti* icon filename

Returns:

0 on success, non-zero on error

30.25.3.13 int Fl_File_Icon::load_image (const char **ifile*)

Load an image icon file from an image filename.

Parameters:

← *ifile* image filename

Returns:

0 on success, non-zero on error

30.25.3.14 void Fl_File_Icon::load_system_icons (void) [static]

Loads all system-defined icons.

This call is useful when using the FileChooser widget and should be used when the application starts:

```
Fl_File_Icon::load_system_icons();
```

30.25.3.15 Fl_File_Icon* Fl_File_Icon::next () [inline]

Returns next file icon object.

See [Fl_File_Icon::first\(\)](#)

30.25.3.16 const char* Fl_File_Icon::pattern () [inline]

Returns the filename matching pattern for the icon.

30.25.3.17 `int Fl_File_Icon::size () [inline]`

Returns the number of words of data used by the icon.

30.25.3.18 `int Fl_File_Icon::type () [inline]`

Returns the filetype associated with the icon, which can be one of the following:.

- `Fl_File_Icon::ANY`, any kind of file.
- `Fl_File_Icon::PLAIN`, plain files.
- `Fl_File_Icon::FIFO`, named pipes.
- `Fl_File_Icon::DEVICE`, character and block devices.
- `Fl_File_Icon::LINK`, symbolic links.
- `Fl_File_Icon::DIRECTORY`, directories.

30.25.3.19 `short* Fl_File_Icon::value () [inline]`

Returns the data array for the icon.

The documentation for this class was generated from the following files:

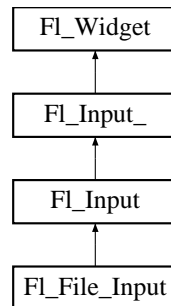
- `Fl_File_Icon.H`
- `Fl_File_Icon.cxx`
- `Fl_File_Icon2.cxx`

30.26 Fl_File_Input Class Reference

This widget displays a pathname in a text input field.

```
#include <Fl_File_Input.H>
```

Inheritance diagram for Fl_File_Input::



Public Member Functions

- void [down_box](#) ([Fl_Boxtype](#) b)
Sets the box type to use for the navigation bar.
- [Fl_Boxtype](#) [down_box](#) () const
Gets the box type used for the navigation bar.
- void [errorcolor](#) ([Fl_Color](#) c)
Sets the current error color to c.
- [Fl_Color](#) [errorcolor](#) () const
Gets the current error color.
- [Fl_File_Input](#) (int X, int Y, int W, int H, const char *L=0)
Creates a new [Fl_File_Input](#) widget using the given position, size, and label string.
- virtual int [handle](#) (int event)
Handle events in the widget.
- const char * [value](#) ()
Returns the current value, which is a pointer to an internal buffer and is valid only until the next event is handled.
- int [value](#) (const char *str, int len)
Sets the value of the widget given a new string value and its length.
- int [value](#) (const char *str)
Sets the value of the widget given a new string value.

Protected Member Functions

- virtual void [draw](#) ()

Draws the file input widget.

30.26.1 Detailed Description

This widget displays a pathname in a text input field.

A navigation bar located above the input field allows the user to navigate upward in the directory tree. You may want to handle FL_WHEN_CHANGED events for tracking text changes and also FL_WHEN_RELEASE for button release when changing to parent dir. FL_WHEN_RELEASE callback won't be called if the directory clicked is the same that the current one.

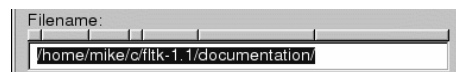


Figure 30.13: Fl_File_Input

Note:

As all [Fl_Input](#) derived objects, [Fl_File_Input](#) may call its callback when loosing focus (see [FL_UNFOCUS](#)) to update its state like its cursor shape. One resulting side effect is that you should call [clear_changed\(\)](#) early in your callback to avoid reentrant calls if you plan to show another window or dialog box in the callback.

30.26.2 Constructor & Destructor Documentation

30.26.2.1 Fl_File_Input::Fl_File_Input (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates a new [Fl_File_Input](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Parameters:

- ← *X,Y,W,H* position and size of the widget
- ← *L* widget label, default is no label

30.26.3 Member Function Documentation

30.26.3.1 void Fl_File_Input::down_box (Fl_Boxtype *b*) [inline]

Sets the box type to use for the navigation bar.

30.26.3.2 Fl_Boxtype Fl_File_Input::down_box () const [inline]

Gets the box type used for the navigation bar.

30.26.3.3 `Fl_Color Fl_File_Input::errorcolor () const` `[inline]`

Gets the current error color.

Todo

Better docs for `Fl_File_Input::errorcolor()` - is it even used?

30.26.3.4 `int Fl_File_Input::handle (int event)` `[virtual]`

Handle events in the widget.

Return non zero if event is handled.

Parameters:

← *event*

Reimplemented from `Fl_Input`.

30.26.3.5 `int Fl_File_Input::value (const char * str, int len)`

Sets the value of the widget given a new string value and its length.

Returns non 0 on success.

Parameters:

← *str* new string value

← *len* length of value

Reimplemented from `Fl_Input_`.

30.26.3.6 `int Fl_File_Input::value (const char * str)`

Sets the value of the widget given a new string value.

Returns non 0 on success.

Parameters:

← *str* new string value

Reimplemented from `Fl_Input_`.

The documentation for this class was generated from the following files:

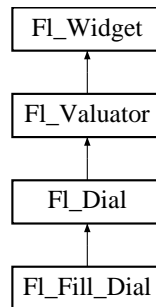
- `Fl_File_Input.H`
- `Fl_File_Input.cxx`

30.27 Fl_Fill_Dial Class Reference

Draws a dial with a filled arc.

```
#include <Fl_Fill_Dial.H>
```

Inheritance diagram for Fl_Fill_Dial::



Public Member Functions

- [Fl_Fill_Dial](#) (int *x*, int *y*, int *w*, int *h*, const char **l*=0)
Creates a filled dial, also setting its type to FL_FILL_DIAL.

30.27.1 Detailed Description

Draws a dial with a filled arc.

30.27.2 Constructor & Destructor Documentation

30.27.2.1 Fl_Fill_Dial::Fl_Fill_Dial (int *x*, int *y*, int *w*, int *h*, const char **l* = 0) [inline]

Creates a filled dial, also setting its type to FL_FILL_DIAL.

The documentation for this class was generated from the following file:

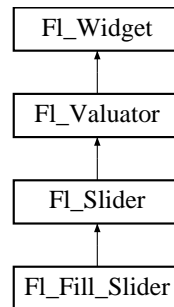
- Fl_Fill_Dial.H

30.28 FL_Fill_Slider Class Reference

Widget that draws a filled horizontal slider, useful as a progress or value meter.

```
#include <FL_Fill_Slider.H>
```

Inheritance diagram for FL_Fill_Slider::



Public Member Functions

- [FL_Fill_Slider](#) (int x, int y, int w, int h, const char *l=0)
Creates the slider from its position,size and optional title.

30.28.1 Detailed Description

Widget that draws a filled horizontal slider, useful as a progress or value meter.

30.28.2 Constructor & Destructor Documentation

30.28.2.1 FL_Fill_Slider::FL_Fill_Slider (int x, int y, int w, int h, const char *l = 0) [inline]

Creates the slider from its position,size and optional title.

The documentation for this class was generated from the following file:

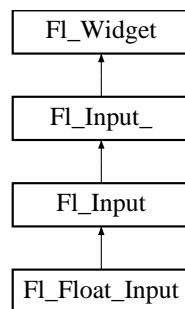
- FL_Fill_Slider.H

30.29 Fl_Float_Input Class Reference

The [Fl_Float_Input](#) class is a subclass of [Fl_Input](#) that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits).

```
#include <Fl_Float_Input.H>
```

Inheritance diagram for Fl_Float_Input::



Public Member Functions

- [Fl_Float_Input](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [Fl_Float_Input](#) widget using the given position, size, and label string.

30.29.1 Detailed Description

The [Fl_Float_Input](#) class is a subclass of [Fl_Input](#) that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits).

30.29.2 Constructor & Destructor Documentation

30.29.2.1 [Fl_Float_Input::Fl_Float_Input](#) (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [Fl_Float_Input](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

Inherited destructor destroys the widget and any value associated with it

The documentation for this class was generated from the following file:

- `Fl_Float_Input.H`

30.30 FL_Font_Descriptor Class Reference

This a structure for an actual system font, with junk to help choose it and info on character sizes.

```
#include <FL_Font.H>
```

Public Attributes

- [FL_Font_Descriptor * next](#)
linked list for this FL_Fontdesc

30.30.1 Detailed Description

This a structure for an actual system font, with junk to help choose it and info on character sizes.

Each FL_Fontdesc has a linked list of these. These are created the first time each system font/size combination is used.

The documentation for this class was generated from the following file:

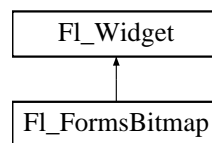
- FL_Font.H

30.31 FL_FormsBitmap Class Reference

Forms compatibility Bitmap Image Widget.

```
#include <FL_FormsBitmap.H>
```

Inheritance diagram for FL_FormsBitmap::



Public Member Functions

- `FL_Bitmap * bitmap () const`
Gets a the current associated [FL_Bitmap](#) objects.
- `void bitmap (FL_Bitmap *B)`
Sets a new bitmap.
- `FL_FormsBitmap (FL_Boxtype, int, int, int, int, const char *==0)`
Creates a bitmap widget from a box type, position, size and optional label specification.
- `void set (int W, int H, const uchar *bits)`
Sets a new bitmap bits with size W,H.

Protected Member Functions

- `void draw ()`
Draws the bitmap and its associated box.

30.31.1 Detailed Description

Forms compatibility Bitmap Image Widget.

30.31.2 Member Function Documentation

30.31.2.1 `FL_Bitmap* FL_FormsBitmap::bitmap () const` [inline]

Gets a the current associated [FL_Bitmap](#) objects.

30.31.2.2 `void FL_FormsBitmap::bitmap (FL_Bitmap * B)` [inline]

Sets a new bitmap.

30.31.2.3 void Fl_FormsBitmap::draw (void) [protected, virtual]

Draws the bitmap and its associated box.

Implements [Fl_Widget](#).

30.31.2.4 void Fl_FormsBitmap::set (int *W*, int *H*, const uchar * *bits*)

Sets a new bitmap bits with size W,H.

Deletes the previous one.

The documentation for this class was generated from the following files:

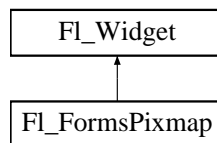
- Fl_FormsBitmap.H
- forms_bitmap.cxx

30.32 FL_FormsPixmap Class Reference

Forms pixmap drawing routines.

```
#include <FL_FormsPixmap.H>
```

Inheritance diagram for FL_FormsPixmap::



Public Member Functions

- [FL_FormsPixmap](#) ([FL_Boxtype](#) t, int X, int Y, int W, int H, const char *L=0)
Creates a new [FL_FormsPixmap](#) widget using the given box type, position, size and label string.
- [FL_Pixmap](#) * [Pixmap](#) () const
Get the internal pixmap pointer.
- void [Pixmap](#) ([FL_Pixmap](#) *B)
Set the internal pixmap pointer to an existing pixmap.
- void [set](#) (char *const *bits)
Set/create the internal pixmap using raw data.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.32.1 Detailed Description

Forms pixmap drawing routines.

30.32.2 Constructor & Destructor Documentation

30.32.2.1 FL_FormsPixmap::FL_FormsPixmap (FL_Boxtype t, int X, int Y, int W, int H, const char * L = 0)

Creates a new [FL_FormsPixmap](#) widget using the given box type, position, size and label string.

Parameters:

← *t* box type

- ← *X,Y,W,H* position and size
- ← *L* widget label, default is no label

30.32.3 Member Function Documentation

30.32.3.1 void Fl_FormsPixmap::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.32.3.2 Fl_Pixmap* Fl_FormsPixmap::Pixmap () const [inline]

Get the internal pixmap pointer.

30.32.3.3 void Fl_FormsPixmap::Pixmap (Fl_Pixmap * *B*) [inline]

Set the internal pixmap pointer to an existing pixmap.

Parameters:

- ← *B* existing pixmap

30.32.3.4 void Fl_FormsPixmap::set (char *const * *bits*)

Set/create the internal pixmap using raw data.

Parameters:

- ← *bits* raw data

The documentation for this class was generated from the following files:

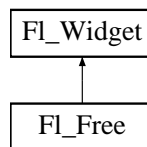
- `Fl_FormsPixmap.H`
- `forms_pixmap.cxx`

30.33 Fl_Free Class Reference

Emulation of the Forms "free" widget.

```
#include <Fl_Free.H>
```

Inheritance diagram for Fl_Free::



Public Member Functions

- [Fl_Free](#) ([uchar](#) t, int X, int Y, int W, int H, const char *L, [FL_HANDLEPTR](#) hdl)
Create a new [Fl_Free](#) widget with type, position, size, label and handler.
- int [handle](#) (int e)
Handles the specified event.
- [~Fl_Free](#) ()
The destructor will call the handle function with the event [FL_FREE_MEM](#).

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.33.1 Detailed Description

Emulation of the Forms "free" widget.

This emulation allows the free demo to run, and appears to be useful for porting programs written in Forms which use the free widget or make subclasses of the Forms widgets.

There are five types of free, which determine when the handle function is called:

- [FL_NORMAL_FREE](#) normal event handling.
- [FL_SLEEPING_FREE](#) deactivates event handling (widget is inactive).
- [FL_INPUT_FREE](#) accepts [FL_FOCUS](#) events.
- [FL_CONTINUOUS_FREE](#) sets a timeout callback 100 times a second and provides an [FL_STEP](#) event. This has obvious detrimental effects on machine performance.
- [FL_ALL_FREE](#) same as [FL_INPUT_FREE](#) and [FL_CONTINUOUS_FREE](#).

30.33.2 Constructor & Destructor Documentation

30.33.2.1 `Fl_Free::Fl_Free (uchar t, int X, int Y, int W, int H, const char * L, FL_HANDLEPTR hdl)`

Create a new `Fl_Free` widget with type, position, size, label and handler.

Parameters:

- ← *t* type
- ← *X,Y,W,H* position and size
- ← *L* widget label
- ← *hdl* handler function

The constructor takes both the type and the handle function. The handle function should be declared as follows:

```
int handle_function(Fl_Widget *w,
                   int      event,
                   float    event_x,
                   float    event_y,
                   char      key)
```

This function is called from the `handle()` method in response to most events, and is called by the `draw()` method.

The event argument contains the event type:

```
// old event names for compatibility:
#define FL_MOUSE      FL_DRAG
#define FL_DRAW       0
#define FL_STEP       9
#define FL_FREEMEM    12
#define FL_FREEZE     FL_UNMAP
#define FL_THAW       FL_MAP
```

30.33.3 Member Function Documentation

30.33.3.1 `void Fl_Free::draw ()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call `redraw()` instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual `draw()` method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                          // calls Fl_Scrollbar::draw()
```

Implements `Fl_Widget`.

30.33.3.2 int Fl_Free::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

The documentation for this class was generated from the following files:

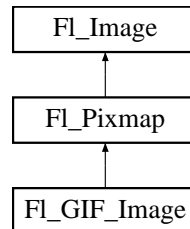
- Fl_Free.H
- forms_free.cxx

30.34 FL_GIF_Image Class Reference

The [FL_GIF_Image](#) class supports loading, caching, and drawing of Compuserve GIFSM images.

```
#include <Fl_GIF_Image.H>
```

Inheritance diagram for FL_GIF_Image::



Public Member Functions

- [FL_GIF_Image](#) (const char *filename)
The constructor loads the named GIF image.

30.34.1 Detailed Description

The [FL_GIF_Image](#) class supports loading, caching, and drawing of Compuserve GIFSM images.

The class loads the first image and supports transparency.

30.34.2 Constructor & Destructor Documentation

30.34.2.1 FL_GIF_Image::FL_GIF_Image (const char * *iname*)

The constructor loads the named GIF image.

The inherited destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

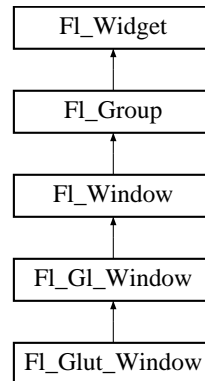
- FL_GIF_Image.H
- FL_GIF_Image.cxx

30.35 FL_Gl_Window Class Reference

The [FL_Gl_Window](#) widget sets things up so OpenGL works.

```
#include <Fl_Gl_Window.H>
```

Inheritance diagram for FL_Gl_Window::



Public Member Functions

- `int can_do ()`
Returns non-zero if the hardware supports the given or current OpenGL mode.
- `int can_do_overlay ()`
Returns true if the hardware overlay is possible.
- `void context (void *, int destroy_flag=0)`
Returns or sets a pointer to the GLContext that this window is using.
- `void * context () const`
void See void [context\(void* v, int destroy_flag\)](#)
- `void context_valid (char v)`
See char [FL_Gl_Window::context_valid\(\)](#) const.
- `char context_valid () const`
Will only be set if the OpenGL context is created or recreated.
- `FL_Gl_Window (int X, int Y, int W, int H, const char *l=0)`
Creates a new [FL_Gl_Window](#) widget using the given position, size, and label string.
- `FL_Gl_Window (int W, int H, const char *l=0)`
Creates a new [FL_Gl_Window](#) widget using the given size, and label string.
- `void flush ()`
Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).
- `void hide ()`

Hides the window and destroys the OpenGL context.

- void [hide_overlay](#) ()
Hides the window if it is not this window, does nothing in WIN32.
- void [invalidate](#) ()
The [invalidate\(\)](#) method turns off [valid\(\)](#) and is equivalent to calling [value\(0\)](#).
- void [make_current](#) ()
The [make_current\(\)](#) method selects the OpenGL context for the widget.
- void [make_overlay_current](#) ()
The [make_overlay_current\(\)](#) method selects the OpenGL context for the widget's overlay.
- int [mode](#) (const int *a)
See [FL_Mode mode\(\)](#) const.
- int [mode](#) (int a)
See [FL_Mode mode\(\)](#) const.
- [FL_Mode mode](#) () const
Set or change the OpenGL capabilities of the window.
- void [ortho](#) ()
Sets the projection so 0,0 is in the lower left of the window and each pixel is 1 unit wide/tall.
- void [redraw_overlay](#) ()
This method causes [draw_overlay\(\)](#) to be called at a later time.
- void [resize](#) (int, int, int, int)
Changes the size and position of the window.
- void [show](#) (int a, char **b)
See virtual void [FL_Window::show\(\)](#).
- void [show](#) ()
Put the window on the screen.
- void [swap_buffers](#) ()
The [swap_buffers\(\)](#) method swaps the back and front buffers.
- void [valid](#) (char v)
See char [FL_Gl_Window::valid\(\)](#) const.
- char [valid](#) () const
Is turned off when FLTK creates a new context for this window or when the window resizes, and is turned on after [draw\(\)](#) is called.
- [~FL_Gl_Window](#) ()
The destructor removes the widget and destroys the OpenGL context associated with it.

Static Public Member Functions

- static int [can_do](#) (const int *m)
Returns non-zero if the hardware supports the given or current OpenGL mode.
- static int [can_do](#) (int m)
Returns non-zero if the hardware supports the given or current OpenGL mode.

Protected Member Functions

- virtual void [draw](#) ()
Draws the [FL_Gl_Window](#).

Friends

- class [_FL_Gl_Overlay](#)

30.35.1 Detailed Description

The [FL_Gl_Window](#) widget sets things up so OpenGL works.

It also keeps an OpenGL "context" for that window, so that changes to the lighting and projection may be reused between redraws. [FL_Gl_Window](#) also flushes the OpenGL streams and swaps buffers after [draw\(\)](#) returns.

OpenGL hardware typically provides some overlay bit planes, which are very useful for drawing UI controls atop your 3D graphics. If the overlay hardware is not provided, FLTK tries to simulate the overlay. This works pretty well if your graphics are double buffered, but not very well for single-buffered.

Please note that the FLTK drawing and clipping functions will not work inside an [FL_Gl_Window](#). All drawing should be done using OpenGL calls exclusively. Even though [FL_Gl_Window](#) is derived from [FL_Group](#), it is not useful to add other FLTK Widgets as children, unless those widgets are modified to draw using OpenGL calls.

30.35.2 Constructor & Destructor Documentation

30.35.2.1 [FL_Gl_Window::FL_Gl_Window](#) (int *W*, int *H*, const char **l* = 0) [inline]

Creates a new [FL_Gl_Window](#) widget using the given size, and label string.

The default boxtype is FL_NO_BOX. The default mode is FL_RGB|FL_DOUBLE|FL_DEPTH.

30.35.2.2 [FL_Gl_Window::FL_Gl_Window](#) (int *X*, int *Y*, int *W*, int *H*, const char **l* = 0) [inline]

Creates a new [FL_Gl_Window](#) widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX. The default mode is FL_RGB|FL_DOUBLE|FL_DEPTH.

30.35.3 Member Function Documentation

30.35.3.1 `int FL_GL_Window::can_do ()` [inline]

Returns non-zero if the hardware supports the given or current OpenGL mode.

30.35.3.2 `static int FL_GL_Window::can_do (const int * m)` [inline, static]

Returns non-zero if the hardware supports the given or current OpenGL mode.

30.35.3.3 `static int FL_GL_Window::can_do (int m)` [inline, static]

Returns non-zero if the hardware supports the given or current OpenGL mode.

30.35.3.4 `int FL_GL_Window::can_do_overlay ()`

Returns true if the hardware overlay is possible.

If this is false, FLTK will try to simulate the overlay, with significant loss of update speed. Calling this will cause FLTK to open the display.

30.35.3.5 `void FL_GL_Window::context (void * v, int destroy_flag = 0)`

Returns or sets a pointer to the GLContext that this window is using.

This is a system-dependent structure, but it is portable to copy the context from one window to another. You can also set it to NULL, which will force FLTK to recreate the context the next time [make_current\(\)](#) is called, this is useful for getting around bugs in OpenGL implementations.

If *destroy_flag* is true the context will be destroyed by fltk when the window is destroyed, or when the [mode\(\)](#) is changed, or the next time [context\(x\)](#) is called.

30.35.3.6 `char FL_GL_Window::context_valid () const` [inline]

Will only be set if the OpenGL context is created or recreated.

It differs from [FL_GL_Window::valid\(\)](#) which is also set whenever the context changes size.

30.35.3.7 `void FL_GL_Window::draw (void)` [protected, virtual]

Draws the [FL_GL_Window](#).

You **must** *subclass* [FL_GL_Window](#) and provide an implementation for [draw\(\)](#).

You **must** *override* the [draw\(\)](#) method.

You may also provide an implementation of [draw_overlay\(\)](#) if you want to draw into the overlay planes. You can avoid reinitializing the viewport and lights and other things by checking [valid\(\)](#) at the start of [draw\(\)](#) and only doing the initialization if it is false.

The [draw\(\)](#) method can *only* use OpenGL calls. Do not attempt to call X, any of the functions in [<FL/fl_-draw.H>](#), or glX directly. Do not call [gl_start\(\)](#) or [gl_finish\(\)](#).

If double-buffering is enabled in the window, the back and front buffers are swapped after this function is completed.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Glut_Window](#).

30.35.3.8 void FL_GI_Window::flush () [virtual]

Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).

Reimplemented from [FL_Window](#).

30.35.3.9 void FL_GI_Window::hide_overlay ()

Hides the window if it is not this window, does nothing in WIN32.

30.35.3.10 void FL_GI_Window::make_current ()

The [make_current\(\)](#) method selects the OpenGL context for the widget.

It is called automatically prior to the [draw\(\)](#) method being called and can also be used to implement feedback and/or selection within the [handle\(\)](#) method.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Glut_Window](#).

30.35.3.11 void FL_GI_Window::make_overlay_current ()

The [make_overlay_current\(\)](#) method selects the OpenGL context for the widget's overlay.

It is called automatically prior to the [draw_overlay\(\)](#) method being called and can also be used to implement feedback and/or selection within the [handle\(\)](#) method.

30.35.3.12 FL_Mode FL_GI_Window::mode () const [inline]

Set or change the OpenGL capabilities of the window.

The value can be any of the following OR'd together:

- `FL_RGB` - RGB color (not indexed)
- `FL_RGB8` - RGB color with at least 8 bits of each color
- `FL_INDEX` - Indexed mode
- `FL_SINGLE` - not double buffered
- `FL_DOUBLE` - double buffered
- `FL_ACCUM` - accumulation buffer
- `FL_ALPHA` - alpha channel in color
- `FL_DEPTH` - depth buffer

- `FL_STENCIL` - stencil buffer
- `FL_MULTISAMPLE` - multisample antialiasing

`FL_RGB` and `FL_SINGLE` have a value of zero, so they are "on" unless you give `FL_INDEX` or `FL_DOUBLE`.

If the desired combination cannot be done, FLTK will try turning off `FL_MULTISAMPLE`. If this also fails the `show()` will call `Fl::error()` and not show the window.

You can change the mode while the window is displayed. This is most useful for turning double-buffering on and off. Under X this will cause the old X window to be destroyed and a new one to be created. If this is a top-level window this will unfortunately also cause the window to blink, raise to the top, and be de-iconized, and the `xid()` will change, possibly breaking other code. It is best to make the GL window a child of another window if you wish to do this!

`mode()` must not be called within `draw()` since it changes the current context.

30.35.3.13 void Fl_Gl_Window::ortho ()

Sets the projection so 0,0 is in the lower left of the window and each pixel is 1 unit wide/tall.

If you are drawing 2D images, your `draw()` method may want to call this if `valid()` is false.

30.35.3.14 void Fl_Gl_Window::redraw_overlay ()

This method causes `draw_overlay()` to be called at a later time.

Initially the overlay is clear. If you want the window to display something in the overlay when it first appears, you must call this immediately after you `show()` your window.

30.35.3.15 void Fl_Gl_Window::resize (int, int, int, int) [virtual]

Changes the size and position of the window.

If `shown()` is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If `shown()` is false, the size and position are used when `show()` is called. See [Fl_Group](#) for the effect of resizing on the child widgets.

You can also call the [Fl_Widget](#) methods `size(x,y)` and `position(w,h)`, which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from [Fl_Window](#).

30.35.3.16 void Fl_Gl_Window::show () [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [Fl_Window](#).

30.35.3.17 void Fl_Gl_Window::swap_buffers ()

The [swap_buffers\(\)](#) method swaps the back and front buffers.

It is called automatically after the [draw\(\)](#) method is called.

30.35.3.18 char Fl_Gl_Window::valid () const [inline]

Is turned off when FLTK creates a new context for this window or when the window resizes, and is turned on *after* [draw\(\)](#) is called.

You can use this inside your [draw\(\)](#) method to avoid unnecessarily initializing the OpenGL context. Just do this:

```
void mywindow::draw() {
    if (!valid()) {
        glViewport(0,0,w(),h());
        glFrustum(...);
        ...other initialization...
    }
    if (!context_valid()) {
        ...load textures, etc. ...
    }
    ... draw your geometry here ...
}
```

You can turn [valid\(\)](#) on by calling [valid\(1\)](#). You should only do this after fixing the transformation inside a [draw\(\)](#) or after [make_current\(\)](#). This is done automatically after [draw\(\)](#) returns.

The documentation for this class was generated from the following files:

- [Fl_Gl_Window.H](#)
- [Fl_Gl_Overlay.cxx](#)
- [Fl_Gl_Window.cxx](#)

30.36 FL_Glut_Bitmap_Font Struct Reference

fltk glut font/size attributes used in the glutXXX functions

```
#include <glut.H>
```

Public Attributes

- [FL_Font](#) font
- [FL_Fontsize](#) size

30.36.1 Detailed Description

fltk glut font/size attributes used in the glutXXX functions

The documentation for this struct was generated from the following file:

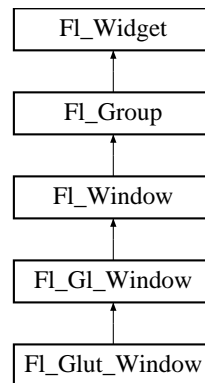
- glut.H

30.37 Fl_Glut_Window Class Reference

GLUT is emulated using this window class and these static variables (plus several more static variables hidden in glut_compatibility.cxx):.

```
#include <glut.H>
```

Inheritance diagram for Fl_Glut_Window::



Public Member Functions

- [Fl_Glut_Window](#) (int x, int y, int w, int h, const char *)
Creates a glut window, registers to the glut windows list.
- [Fl_Glut_Window](#) (int w, int h, const char *)
Creates a glut window, registers to the glut windows list.
- void [make_current](#) ()
The [make_current\(\)](#) method selects the OpenGL context for the widget.
- [~Fl_Glut_Window](#) ()
Destroys the glut window, first unregister it from the glut windows list.

Public Attributes

- void(* **display**)()
- void(* **entry**)(int)
- void(* **keyboard**)(uchar, int x, int y)
- int **menu** [3]
- void(* **motion**)(int x, int y)
- void(* **mouse**)(int b, int state, int x, int y)
- int **number**
- void(* **overlaydisplay**)()
- void(* **passivemotion**)(int x, int y)
- void(* **reshape**)(int w, int h)
- void(* **special**)(int, int x, int y)
- void(* **visibility**)(int)

Protected Member Functions

- void [draw](#) ()
Draws the [FL_Gl_Window](#).
- void [draw_overlay](#) ()
You must implement this virtual function if you want to draw into the overlay.
- int [handle](#) (int)
Handles the specified event.

30.37.1 Detailed Description

GLUT is emulated using this window class and these static variables (plus several more static variables hidden in `glut_compatibility.cxx`):.

30.37.2 Constructor & Destructor Documentation

30.37.2.1 [FL_Glut_Window::FL_Glut_Window](#) (int *W*, int *H*, const char * *t*)

Creates a glut window, registers to the glut windows list.

30.37.2.2 [FL_Glut_Window::FL_Glut_Window](#) (int *X*, int *Y*, int *W*, int *H*, const char * *t*)

Creates a glut window, registers to the glut windows list.

30.37.3 Member Function Documentation

30.37.3.1 void [FL_Glut_Window::draw](#) (void) [protected, virtual]

Draws the [FL_Gl_Window](#).

You **must** subclass [FL_Gl_Window](#) and provide an implementation for [draw\(\)](#).

You **must** override the [draw\(\)](#) method.

You may also provide an implementation of [draw_overlay\(\)](#) if you want to draw into the overlay planes. You can avoid reinitializing the viewport and lights and other things by checking [valid\(\)](#) at the start of [draw\(\)](#) and only doing the initialization if it is false.

The [draw\(\)](#) method can *only* use OpenGL calls. Do not attempt to call X, any of the functions in `<FL/fl_-draw.H>`, or glX directly. Do not call `gl_start()` or `gl_finish()`.

If double-buffering is enabled in the window, the back and front buffers are swapped after this function is completed.

Reimplemented from [FL_Gl_Window](#).

30.37.3.2 void [FL_Glut_Window::draw_overlay](#) () [protected, virtual]

You must implement this virtual function if you want to draw into the overlay.

The overlay is cleared before this is called. You should draw anything that is not clear using OpenGL. You must use `gl_color(i)` to choose colors (it allocates them from the colormap using system-specific calls), and remember that you are in an indexed OpenGL mode and drawing anything other than flat-shaded will probably not work.

Both this function and `FL_Glut_Window::draw()` should check `FL_Glut_Window::valid()` and set the same transformation. If you don't your code may not work on other systems. Depending on the OS, and on whether overlays are real or simulated, the OpenGL context may be the same or different between the overlay and main window.

Reimplemented from `FL_Glut_Window`.

30.37.3.3 `int FL_Glut_Window::handle (int event)` [protected, virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited `handle()` method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from `FL_Window`.

30.37.3.4 `void FL_Glut_Window::make_current ()`

The `make_current()` method selects the OpenGL context for the widget.

It is called automatically prior to the `draw()` method being called and can also be used to implement feedback and/or selection within the `handle()` method.

Reimplemented from `FL_Glut_Window`.

The documentation for this class was generated from the following files:

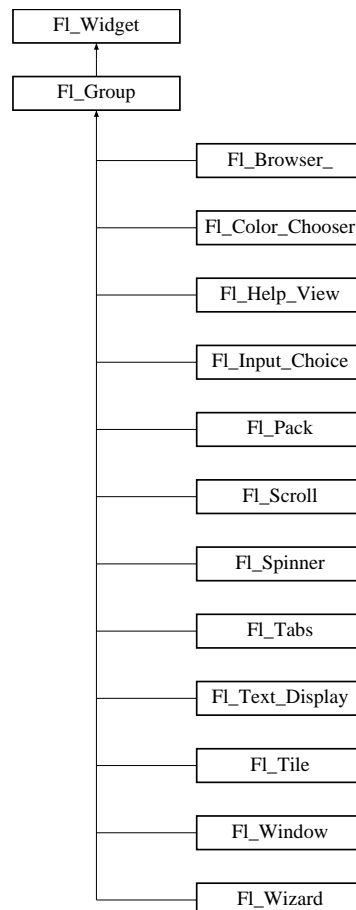
- `glut.H`
- `glut_compatibility.cxx`

30.38 FL_Group Class Reference

The [FL_Group](#) class is the FLTK container widget.

```
#include <Fl_Group.H>
```

Inheritance diagram for `Fl_Group::`



Public Member Functions

- [Fl_Widget *& _ddfdesign_kludge \(\)](#)
This is for forms compatibility only.
- void [add \(Fl_Widget *o\)](#)
See void [Fl_Group::add\(Fl_Widget &w\)](#).
- void [add \(Fl_Widget &\)](#)
The widget is removed from its current group (if any) and then added to the end of this group.
- void [add_resizable \(Fl_Widget &o\)](#)
Adds a widget to the group and makes it the resizable widget.

- `FL_Widget *const * array () const`
Returns a pointer to the array of children.
- `void begin ()`
Sets the current group so you can build the widget tree by just constructing the widgets.
- `FL_Widget * child (int n) const`
Returns `array()[n]`.
- `int children () const`
Returns how many child widgets the group has.
- `void clear ()`
Deletes all child widgets from memory recursively.
- `int clip_children ()`
Returns the current clipping mode.
- `void clip_children (int c)`
Controls whether the group widget clips the drawing of child widgets to its bounding box.
- `void end ()`
Exactly the same as `current(this->parent())`.
- `int find (const FL_Widget &o) const`
*See `int FL_Group::find(const FL_Widget *w) const`.*
- `int find (const FL_Widget *) const`
Searches the child array for the widget and returns the index.
- `FL_Group (int, int, int, int, const char * = 0)`
Creates a new `FL_Group` widget using the given position, size, and label string.
- `void focus (FL_Widget *W)`
- `void forms_end ()`
This is for forms compatibility only.
- `int handle (int)`
Handles the specified event.
- `void init_sizes ()`
Resets the internal array of widget sizes and positions.
- `void insert (FL_Widget &o, FL_Widget *before)`
This does `insert(w, find(before))`.
- `void insert (FL_Widget &, int i)`
The widget is removed from its current group (if any) and then inserted into this group.
- `void remove (FL_Widget *o)`

Removes the widget `o` from the group.

- void `remove` (`Fl_Widget &`)
Removes a widget from the group but does not delete it.
- `Fl_Widget * resizable` () const
*See void `Fl_Group::resizable(Fl_Widget *box)`.*
- void `resizable` (`Fl_Widget *o`)
The resizable widget defines the resizing box for the group.
- void `resizable` (`Fl_Widget &o`)
*See void `Fl_Group::resizable(Fl_Widget *box)`.*
- void `resize` (int, int, int, int)
Resizes the `Fl_Group` widget and all of its children.
- virtual `~Fl_Group` ()
The destructor also deletes all the children.

Static Public Member Functions

- static void `current` (`Fl_Group *g`)
*See static `Fl_Group *Fl_Groupcurrent()`.*
- static `Fl_Group * current` ()
Returns the currently active group.

Protected Types

- enum { `CLIP_CHILDREN` = 2048 }

Protected Member Functions

- void `draw` ()
Draws the widget.
- void `draw_child` (`Fl_Widget &widget`) const
Forces a child to redraw.
- void `draw_children` ()
Draws all children of the group.
- void `draw_outside_label` (const `Fl_Widget &widget`) const
Parents normally call this to draw outside labels of child widgets.
- int * `sizes` ()

Returns the internal array of widget sizes and positions.

- void `update_child` (FL_Widget &widget) const

Draws a child only if it needs it.

30.38.1 Detailed Description

The `FL_Group` class is the FLTK container widget.

It maintains an array of child widgets. These children can themselves be any widget including `FL_Group`. The most important subclass of `FL_Group` is `FL_Window`, however groups can also be used to control radio buttons or to enforce resize behavior.

30.38.2 Constructor & Destructor Documentation

30.38.2.1 `FL_Group::FL_Group (int X, int Y, int W, int H, const char *l = 0)`

Creates a new `FL_Group` widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

30.38.2.2 `FL_Group::~~FL_Group ()` [virtual]

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code.

It is allowed that the `FL_Group` and all of its children are automatic (local) variables, but you must declare the `FL_Group` *first*, so that it is destroyed last.

If you add static or automatic (local) variables to an `FL_Group`, then it is your responsibility to remove (or delete) all such static or automatic child widgets **before** *destroying* the group - otherwise the child widgets' destructors would be called twice!

30.38.3 Member Function Documentation

30.38.3.1 `FL_Widget *const * FL_Group::array () const`

Returns a pointer to the array of children.

This pointer is only valid until the next time a child is added or removed.

30.38.3.2 `void FL_Group::begin ()`

Sets the current group so you can build the widget tree by just constructing the widgets.

`begin()` is automatically called by the constructor for `FL_Group` (and thus for `FL_Window` as well). `begin()` is *exactly the same as* `current(this)`. *Don't forget to `end()` the group or window!*

30.38.3.3 `Fl_Widget* Fl_Group::child (int n) const` [inline]

Returns `array()[n]`.

No range checking is done!

30.38.3.4 `void Fl_Group::clear ()`

Deletes all child widgets from memory recursively.

This method differs from the `remove()` method in that it affects all child widgets and deletes them from memory.

Reimplemented in `Fl_Browser`, `Fl_Check_Browser`, `Fl_Input_Choice`, and `Fl_Scroll`.

30.38.3.5 `int Fl_Group::clip_children ()` [inline]

Returns the current clipping mode.

Returns:

true, if clipping is enabled, false otherwise.

See also:

void `Fl_Group::clip_children(int c)`

30.38.3.6 `void Fl_Group::clip_children (int c)` [inline]

Controls whether the group widget clips the drawing of child widgets to its bounding box.

Set *c* to 1 if you want to clip the child widgets to the bounding box.

The default is to not clip (0) the drawing of child widgets.

30.38.3.7 `Fl_Group * Fl_Group::current ()` [static]

Returns the currently active group.

The `Fl_Widget` constructor automatically does `current()->add(widget)` if this is not null. To prevent new widgets from being added to a group, call `Fl_Group::current(0)`.

Reimplemented in `Fl_Window`.

30.38.3.8 `void Fl_Group::draw ()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call `redraw()` instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual `draw()` method.

```
Fl_Widget *s = &scroll;          // scroll is an embedded Fl_Scrollbar
s->draw();                       // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

Reimplemented in [Fl_Browser_](#), [Fl_Cairo_Window](#), [Fl_Gl_Window](#), [Fl_Pack](#), [Fl_Scroll](#), [Fl_Tabs](#), [Fl_Text_Display](#), [Fl_Window](#), and [Fl_Glut_Window](#).

30.38.3.9 void FL_Group::draw_child (FL_Widget & widget) const [protected]

Forces a child to redraw.

This draws a child widget, if it is not clipped. The damage bits are cleared after drawing.

30.38.3.10 void FL_Group::draw_children () [protected]

Draws all children of the group.

This is useful, if you derived a widget from [FL_Group](#) and want to draw a special border or background. You can call [draw_children\(\)](#) from the derived [draw\(\)](#) method after drawing the box, border, or background.

30.38.3.11 void FL_Group::draw_outside_label (const FL_Widget & widget) const [protected]

Parents normally call this to draw outside labels of child widgets.

30.38.3.12 void FL_Group::end ()

Exactly the same as `current(this->parent())`.

Any new widgets added to the widget tree will be added to the parent of the group.

30.38.3.13 int FL_Group::find (const FL_Widget * o) const

Searches the child array for the widget and returns the index.

Returns [children\(\)](#) if the widget is NULL or not found.

30.38.3.14 void FL_Group::focus (FL_Widget * W) [inline]

Deprecated

This is for backwards compatibility only. You should use `W->take_focus()` instead.

See also:

[Fl_Widget::take_focus\(\)](#);

30.38.3.15 int FL_Group::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

Reimplemented in [Fl_Browser_](#), [Fl_Check_Browser](#), [Fl_Scroll](#), [Fl_Spinner](#), [Fl_Tabs](#), [Fl_Text_Display](#), [Fl_Text_Editor](#), [Fl_Tile](#), [Fl_Window](#), and [Fl_Glut_Window](#).

30.38.3.16 void Fl_Group::init_sizes ()

Resets the internal array of widget sizes and positions.

The [Fl_Group](#) widget keeps track of the original widget sizes and positions when resizing occurs so that if you resize a window back to its original size the widgets will be in the correct places. If you rearrange the widgets in your group, call this method to register the new arrangement with the [Fl_Group](#) that contains them.

If you add or remove widgets, this will be done automatically.

Note:

The internal array of widget sizes and positions will be allocated and filled when the next [resize\(\)](#) occurs.

See also:

[sizes\(\)](#)

30.38.3.17 void Fl_Group::insert (Fl_Widget & o, Fl_Widget * before) [inline]

This does insert(w, find(before)).

This will append the widget if *before* is not in the group.

30.38.3.18 void Fl_Group::insert (Fl_Widget & o, int index)

The widget is removed from its current group (if any) and then inserted into this group.

It is put at index *n* - or at the end, if *n* >= [children\(\)](#). This can also be used to rearrange the widgets inside a group.

30.38.3.19 void Fl_Group::remove (Fl_Widget * o) [inline]

Removes the widget *o* from the group.

See also:

void [remove\(Fl_Widget&\)](#)

30.38.3.20 void Fl_Group::remove (Fl_Widget & o)

Removes a widget from the group but does not delete it.

This method does nothing if the widget is not a child of the group.

This method differs from the [clear\(\)](#) method in that it only affects a single widget and does not delete it from memory.

30.38.3.21 void Fl_Group::resizable (Fl_Widget * o) [inline]

The resizable widget defines the resizing box for the group.

When the group is resized it calculates a new size and position for all of its children. Widgets that are horizontally or vertically inside the dimensions of the box are scaled to the new size. Widgets outside the box are moved.

In these examples the gray area is the resizable:

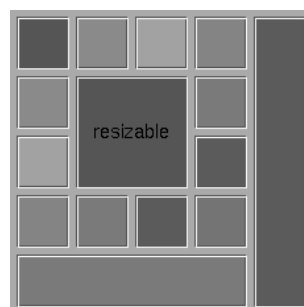


Figure 30.14: before resize

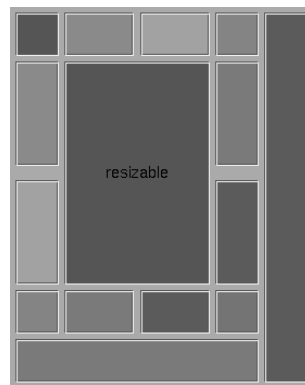


Figure 30.15: after resize

The `resizable` may be set to the group itself, in which case all the contents are resized. This is the default value for `FL_Group`, although `NULL` is the default for `FL_Window` and `FL_Pack`.

If the `resizable` is `NULL` then all widgets remain a fixed size and distance from the top-left corner.

It is possible to achieve any type of resize behavior by using an invisible `FL_Box` as the `resizable` and/or by using a hierarchy of child `FL_Group`'s.

30.38.3.22 void FL_Group::resize (int X, int Y, int W, int H) [virtual]

Resizes the `FL_Group` widget and all of its children.

The `FL_Group` widget first resizes itself, and then it moves and resizes all its children according to the rules documented for `FL_Group::resizable(FL_Widget*)`

See also:

`FL_Group::resizable(FL_Widget*)`
`FL_Group::resizable()`
`FL_Widget::resize(int,int,int,int)`

Reimplemented from `FL_Widget`.

Reimplemented in `FL_Browser_`, `FL_Double_Window`, `FL_Gl_Window`, `FL_Help_View`, `FL_Input_Choice`, `FL_Overlay_Window`, `FL_Scroll`, `FL_Spinner`, `FL_Text_Display`, `FL_Tile`, and `FL_Window`.

30.38.3.23 int * FL_Group::sizes () [protected]

Returns the internal array of widget sizes and positions.

If the `sizes()` array does not exist, it will be allocated and filled with the current widget sizes and positions.

Note:

You should never need to use this method directly, unless you have special needs to rearrange the children of a `FL_Group`. `FL_Tile` uses this to rearrange its widget positions.

See also:

`init_sizes()`

Todo

Should the internal representation of the [sizes\(\)](#) array be documented?

30.38.3.24 void Fl_Group::update_child (Fl_Widget & *widget*) const [protected]

Draws a child only if it needs it.

This draws a child widget, if it is not clipped *and* if any [damage\(\)](#) bits are set. The damage bits are cleared after drawing.

See also:

[Fl_Group::draw_child\(Fl_Widget& widget\) const](#)

The documentation for this class was generated from the following files:

- Fl_Group.H
- Fl_Group.cxx
- forms_compatibility.cxx

30.39 FL_Help_Dialog Class Reference

The [FL_Help_Dialog](#) widget displays a standard help dialog window using the [FL_Help_View](#) widget.

Public Member Functions

- [FL_Help_Dialog](#) ()
The constructor creates the dialog pictured above.
- int [h](#) ()
Returns the position and size of the help dialog.
- void [hide](#) ()
Hides the [FL_Help_Dialog](#) window.
- void [load](#) (const char *f)
Loads the specified HTML file into the [FL_Help_View](#) widget.
- void [position](#) (int xx, int yy)
Set the screen position of the dialog.
- void [resize](#) (int xx, int yy, int ww, int hh)
Change the position and size of the dialog.
- void [show](#) (int argc, char **argv)
*Shows the main Help Dialog Window Delegates call to encapsulated window_ void [FL_Window::show\(int argc, char **argv\)](#) instance method.*
- void [show](#) ()
Shows the [FL_Help_Dialog](#) window.
- [FL_Fontsize](#) [textsize](#) ()
Sets or gets the default text size for the help view.
- void [textsize](#) ([FL_Fontsize](#) s)
Sets or gets the default text size for the help view.
- void [topline](#) (int n)
Sets the top line in the [FL_Help_View](#) widget to the named or numbered line.
- void [topline](#) (const char *n)
Sets the top line in the [FL_Help_View](#) widget to the named or numbered line.
- const char * [value](#) () const
The first form sets the current buffer to the string provided and reformats the text.
- void [value](#) (const char *f)
The first form sets the current buffer to the string provided and reformats the text.

- `int visible ()`
Returns 1 if the *FL_Help_Dialog* window is visible.
- `int w ()`
Returns the position and size of the help dialog.
- `int x ()`
Returns the position and size of the help dialog.
- `int y ()`
Returns the position and size of the help dialog.
- `~FL_Help_Dialog ()`
The destructor destroys the widget and frees all memory that has been allocated for the current file.

30.39.1 Detailed Description

The *FL_Help_Dialog* widget displays a standard help dialog window using the *FL_Help_View* widget.

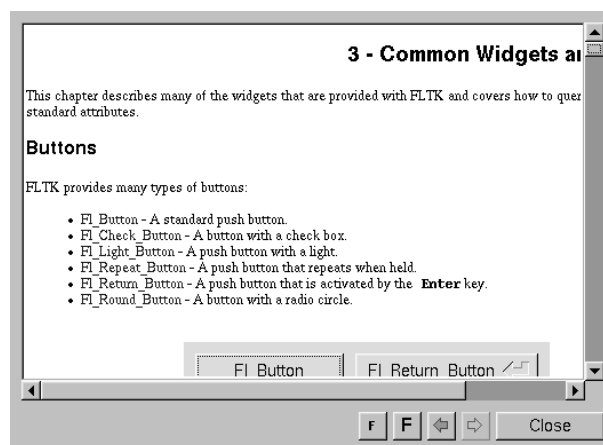


Figure 30.16: FL_Help_Dialog

30.39.2 Constructor & Destructor Documentation

30.39.2.1 FL_Help_Dialog::FL_Help_Dialog ()

The constructor creates the dialog pictured above.

30.39.3 Member Function Documentation

30.39.3.1 int FL_Help_Dialog::h ()

Returns the position and size of the help dialog.

30.39.3.2 void Fl_Help_Dialog::hide ()

Hides the [Fl_Help_Dialog](#) window.

30.39.3.3 void Fl_Help_Dialog::load (const char *f)

Loads the specified HTML file into the [Fl_Help_View](#) widget.

The filename can also contain a target name ("filename.html#target").

30.39.3.4 void Fl_Help_Dialog::position (int x, int y)

Set the screen position of the dialog.

30.39.3.5 void Fl_Help_Dialog::resize (int xx, int yy, int ww, int hh)

Change the position and size of the dialog.

30.39.3.6 void Fl_Help_Dialog::show ()

Shows the [Fl_Help_Dialog](#) window.

Shows the main Help Dialog Window Delegates call to encapsulated window_ void [Fl_Window::show\(\)](#) method.

30.39.3.7 uchar Fl_Help_Dialog::textsize ()

Sets or gets the default text size for the help view.

30.39.3.8 void Fl_Help_Dialog::textsize (Fl_Fontsize s)

Sets or gets the default text size for the help view.

Sets the internal [Fl_Help_View](#) instance text size.

Delegates call to encapsulated view_ void [Fl_Help_View::textsize\(Fl_Fontsize s\)](#) instance method

30.39.3.9 const char * Fl_Help_Dialog::value () const

The first form sets the current buffer to the string provided and reformats the text.

It also clears the history of the "back" and "forward" buttons. The second form returns the current buffer contents.

30.39.3.10 void Fl_Help_Dialog::value (const char *v)

The first form sets the current buffer to the string provided and reformats the text.

It also clears the history of the "back" and "forward" buttons. The second form returns the current buffer contents.

30.39.3.11 int Fl_Help_Dialog::visible ()

Returns 1 if the [Fl_Help_Dialog](#) window is visible.

30.39.3.12 int Fl_Help_Dialog::w ()

Returns the position and size of the help dialog.

30.39.3.13 int Fl_Help_Dialog::x ()

Returns the position and size of the help dialog.

30.39.3.14 int Fl_Help_Dialog::y ()

Returns the position and size of the help dialog.

The documentation for this class was generated from the following files:

- Fl_Help_Dialog.H
- Fl_Help_Dialog.cxx
- Fl_Help_Dialog_Dox.cxx

30.40 Fl_Help_Font_Style Struct Reference

[Fl_Help_View](#) font stack element definition.

```
#include <Fl_Help_View.H>
```

Public Member Functions

- **Fl_Help_Font_Style** ([Fl_Font](#) afont, [Fl_Fontsize](#) asize, [Fl_Color](#) acolor)
- void [get](#) ([Fl_Font](#) &afont, [Fl_Fontsize](#) &asize, [Fl_Color](#) &acolor)
Gets current font attributes.
- void [set](#) ([Fl_Font](#) afont, [Fl_Fontsize](#) asize, [Fl_Color](#) acolor)
Sets current font attributes.

Public Attributes

- [Fl_Color](#) c
Font Color.
- [Fl_Font](#) f
Font.
- [Fl_Fontsize](#) s
Font Size.

30.40.1 Detailed Description

[Fl_Help_View](#) font stack element definition.

The documentation for this struct was generated from the following file:

- [Fl_Help_View.H](#)

30.41 Fl_Help_Link Struct Reference

Definition of a link for the html viewer.

```
#include <Fl_Help_View.H>
```

Public Attributes

- char [filename](#) [192]
Reference filename.
- int [h](#)
Height of link text.
- char [name](#) [32]
Link target (blank if none).
- int [w](#)
Width of link text.
- int [x](#)
X offset of link text.
- int [y](#)
Y offset of link text.

30.41.1 Detailed Description

Definition of a link for the html viewer.

The documentation for this struct was generated from the following file:

- Fl_Help_View.H

30.42 Fl_Help_Target Struct Reference

[Fl_Help_Target](#) structure.

```
#include <Fl_Help_View.H>
```

Public Attributes

- char [name](#) [32]
Target name.
- int [y](#)
Y offset of target.

30.42.1 Detailed Description

[Fl_Help_Target](#) structure.

The documentation for this struct was generated from the following file:

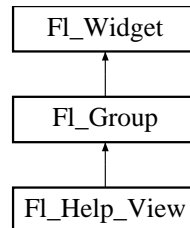
- Fl_Help_View.H

30.43 Fl_Help_View Class Reference

The [Fl_Help_View](#) widget displays HTML text.

```
#include <Fl_Help_View.H>
```

Inheritance diagram for Fl_Help_View::



Public Member Functions

- void [clear_selection](#) ()
Removes the current text selection.
- const char * [directory](#) () const
This method returns the current directory for the text in the buffer.
- const char * [filename](#) () const
This method returns the current filename for the text in the buffer.
- int [find](#) (const char *s, int p=0)
Find the specified string s at starting position p, return the matching pos or -1 if not found.
- [Fl_Help_View](#) (int xx, int yy, int ww, int hh, const char *l=0)
The constructor creates the [Fl_Help_View](#) widget at the specified position and size.
- int [leftline](#) () const
Gets the left position.
- void [leftline](#) (int)
Sets the left position.
- void [link](#) (Fl_Help_Func *fn)
This method assigns a callback function to use when a link is followed or a file is loaded (via [Fl_Help_View::load\(\)](#)) that requires a different file or path.
- int [load](#) (const char *f)
Load the specified file.
- void [resize](#) (int, int, int, int)
Resize the help widget.
- void [select_all](#) ()

Selects All the text in the view.

- void [size](#) (int W, int H)

Changes the size of the widget.

- int [size](#) () const

Gets the size of the Help view.

- [Fl_Color](#) [textcolor](#) () const

Returns the current default text color.

- void [textcolor](#) ([Fl_Color](#) c)

Sets the default text color.

- [Fl_Font](#) [textfont](#) () const

Returns the current default text font.

- void [textfont](#) ([Fl_Font](#) f)

Sets the default text font.

- [Fl_Fontsize](#) [textsize](#) () const

Gets the default text size.

- void [textsize](#) ([Fl_Fontsize](#) s)

Sets the default text size.

- const char * [title](#) ()

Returns the current document title, or NULL if there is no title.

- int [topline](#) () const

Returns the current top line in pixels.

- void [topline](#) (int)

Scroll the text to the indicated position, given a pixel line.

- void [topline](#) (const char *n)

Scroll the text to the indicated position, given a named destination.

- const char * [value](#) () const

Returns the current buffer contents.

- void [value](#) (const char *v)

Sets the current help text buffer to the string provided and reformats the text.

- [~Fl_Help_View](#) ()

Destroy a [Fl_Help_View](#) widget.

30.43.1 Detailed Description

The [Fl_Help_View](#) widget displays HTML text.

Most HTML 2.0 elements are supported, as well as a primitive implementation of tables. GIF, JPEG, and PNG images are displayed inline.

30.43.2 Constructor & Destructor Documentation

30.43.2.1 [Fl_Help_View::~~Fl_Help_View \(\)](#)

Destroy a [Fl_Help_View](#) widget.

The destructor destroys the widget and frees all memory that has been allocated for the current file.

30.43.3 Member Function Documentation

30.43.3.1 [void Fl_Help_View::clear_selection \(\)](#)

Removes the current text selection.

30.43.3.2 [const char* Fl_Help_View::directory \(\) const](#) [inline]

This method returns the current directory for the text in the buffer.

30.43.3.3 [const char* Fl_Help_View::filename \(\) const](#) [inline]

This method returns the current filename for the text in the buffer.

30.43.3.4 [int Fl_Help_View::leftline \(\) const](#) [inline]

Gets the left position.

30.43.3.5 [void Fl_Help_View::leftline \(int l\)](#)

Sets the left position.

30.43.3.6 [void Fl_Help_View::link \(Fl_Help_Func *fn\)](#) [inline]

This method assigns a callback function to use when a link is followed or a file is loaded (via [Fl_Help_View::load\(\)](#)) that requires a different file or path.

The callback function receives a pointer to the [Fl_Help_View](#) widget and the URI or full pathname for the file in question. It must return a pathname that can be opened as a local file or NULL:

```
const char *fn(Fl_Widget *w, const char *uri);
```

The link function can be used to retrieve remote or virtual documents, returning a temporary file that contains the actual data. If the link function returns NULL, the value of the [Fl_Help_View](#) widget will remain unchanged.

If the link callback cannot handle the URI scheme, it should return the uri value unchanged or set the [value\(\)](#) of the widget before returning NULL.

30.43.3.7 `int FL_Help_View::load (const char *f)`

Load the specified file.

This method loads the specified file or URL.

30.43.3.8 `void FL_Help_View::resize (int xx, int yy, int ww, int hh)` [virtual]

Resize the help widget.

Reimplemented from [FL_Group](#).

30.43.3.9 `void FL_Help_View::select_all ()`

Selects All the text in the view.

30.43.3.10 `void FL_Help_View::size (int W, int H)` [inline]

Changes the size of the widget.

size(W, H) is a shortcut for [resize\(x\(\), y\(\), W, H\)](#).

Parameters:

← *W,H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented from [FL_Widget](#).

30.43.3.11 `FL_Color FL_Help_View::textcolor () const` [inline]

Returns the current default text color.

30.43.3.12 `void FL_Help_View::textcolor (FL_Color c)` [inline]

Sets the default text color.

30.43.3.13 `FL_Font FL_Help_View::textfont () const` [inline]

Returns the current default text font.

30.43.3.14 `void FL_Help_View::textfont (FL_Font f)` [inline]

Sets the default text font.

30.43.3.15 `Fl_Fontsize Fl_Help_View::textsize () const` `[inline]`

Gets the default text size.

30.43.3.16 `void Fl_Help_View::textsize (Fl_Fontsize s)` `[inline]`

Sets the default text size.

30.43.3.17 `const char* Fl_Help_View::title ()` `[inline]`

Returns the current document title, or NULL if there is no title.

30.43.3.18 `int Fl_Help_View::topline () const` `[inline]`

Returns the current top line in pixels.

30.43.3.19 `void Fl_Help_View::topline (int t)`

Scroll the text to the indicated position, given a pixel line.

30.43.3.20 `const char* Fl_Help_View::value () const` `[inline]`

Returns the current buffer contents.

30.43.3.21 `void Fl_Help_View::value (const char * v)`

Sets the current help text buffer to the string provided and reformats the text.

The documentation for this class was generated from the following files:

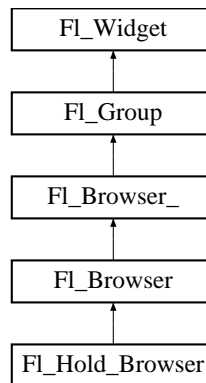
- Fl_Help_View.H
- Fl_Help_View.cxx

30.44 FL_Hold_Browser Class Reference

The [FL_Hold_Browser](#) is a subclass of [FL_Browser](#) which lets the user select a single item, or no items by clicking on the empty space.

```
#include <Fl_Hold_Browser.H>
```

Inheritance diagram for [FL_Hold_Browser](#)::



Public Member Functions

- [FL_Hold_Browser](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [FL_Hold_Browser](#) widget using the given position, size, and label string.

30.44.1 Detailed Description

The [FL_Hold_Browser](#) is a subclass of [FL_Browser](#) which lets the user select a single item, or no items by clicking on the empty space.

As long as the mouse button is held down the item pointed to by it is highlighted, and this highlighting remains on when the mouse button is released. Normally the callback is done when the user releases the mouse, but you can change this with [when\(\)](#).

See [FL_Browser](#) for methods to add and remove lines from the browser.

30.44.2 Constructor & Destructor Documentation

30.44.2.1 [FL_Hold_Browser::FL_Hold_Browser](#) (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [FL_Hold_Browser](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`. The constructor specializes [FL_Browser\(\)](#) by setting the type to `FL_HOLD_BROWSER`. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following file:

- [Fl_Hold_Browser.H](#)

30.45 Fl_Image Class Reference

[Fl_Image](#) is the base class used for caching and drawing all kinds of images in FLTK.

```
#include <Fl_Image.H>
```

Inheritance diagram for Fl_Image::



Public Member Functions

- virtual void [color_average](#) ([Fl_Color](#) c, float i)
The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.
- [Fl_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [Fl_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- int [count](#) () const
The [count\(\)](#) method returns the number of data values associated with the image.
- int [d](#) () const
The first form of the [d\(\)](#) method returns the current image depth.
- const char *const * [data](#) () const
The first form of the [data\(\)](#) method returns a pointer to the current image data array.
- virtual void [desaturate](#) ()
The [desaturate\(\)](#) method converts an image to grayscale.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx=0, int cy=0)
The [draw\(\)](#) methods draw the image.
- [Fl_Image](#) (int W, int H, int D)
The constructor creates an empty image with the specified width, height, and depth.
- int [h](#) () const
See void [Fl_Image::h\(int\)](#).
- void [inactive](#) ()

The `inactive()` method calls `color_average(FL_BACKGROUND_COLOR, 0.33f)` to produce an image that appears grayed out.

- virtual void `label` (`Fl_Menu_Item` *m)

The `label()` methods are an obsolete way to set the image attribute of a widget or menu item.

- virtual void `label` (`Fl_Widget` *w)

The `label()` methods are an obsolete way to set the image attribute of a widget or menu item.

- int `ld` () const

The first form of the `ld()` method returns the current line data size in bytes.

- virtual void `uncache` ()

If the image has been cached for display, delete the cache data.

- int `w` () const

See void `Fl_Image::w(int)`.

- virtual `~Fl_Image` ()

The destructor is a virtual method that frees all memory used by the image.

Protected Member Functions

- void `d` (int D)

The first form of the `d()` method returns the current image depth.

- void `data` (const char *const *p, int c)

See const char * const *data().

- void `draw_empty` (int X, int Y)

The protected method `draw_empty()` draws a box with an X in it.

- void `h` (int H)

The first form of the `h()` method returns the current image height in pixels.

- void `ld` (int LD)

See int `ld()`.

- void `w` (int W)

The first form of the `w()` method returns the current image width in pixels.

Static Protected Member Functions

- static void `labeltype` (const `Fl_Label` *lo, int lx, int ly, int lw, int lh, `Fl_Align` la)
- static void `measure` (const `Fl_Label` *lo, int &lw, int &lh)

30.45.1 Detailed Description

[FL_Image](#) is the base class used for caching and drawing all kinds of images in FLTK.

This class keeps track of common image data such as the pixels, colormap, width, height, and depth. Virtual methods are used to provide type-specific image handling.

Since the [FL_Image](#) class does not support image drawing by itself, calling the [draw\(\)](#) method results in a box with an X in it being drawn instead.

30.45.2 Constructor & Destructor Documentation

30.45.2.1 [FL_Image::FL_Image \(int *W*, int *H*, int *D*\)](#) [inline]

The constructor creates an empty image with the specified width, height, and depth.

The width and height are in pixels. The depth is 0 for bitmaps, 1 for pixmap (colormap) images, and 1 to 4 for color images.

30.45.3 Member Function Documentation

30.45.3.1 [void FL_Image::color_average \(FL_Color *c*, float *i*\)](#) [virtual]

The [color_average\(\)](#) method averages the colors in the image with the FLTK color value *c*.

The *i* argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color. *The original image data is not altered by this method.*

Reimplemented in [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.2 [FL_Image* FL_Image::copy \(\)](#) [inline]

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [FL_Shared_Image](#), released) when you are done with it.

Reimplemented in [FL_Bitmap](#), [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.3 [FL_Image * FL_Image::copy \(int *W*, int *H*\)](#) [virtual]

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [FL_Shared_Image](#), released) when you are done with it.

Reimplemented in [FL_Bitmap](#), [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.4 [int FL_Image::count \(\) const](#) [inline]

The [count\(\)](#) method returns the number of data values associated with the image.

The value will be 0 for images with no associated data, 1 for bitmap and color images, and greater than 2 for pixmap images.

30.45.3.5 `int FL_Image::d () const` [inline]

The first form of the `d()` method returns the current image depth.

The return value will be 0 for bitmaps, 1 for pixmaps, and 1 to 4 for color images.

The second form is a protected method that sets the current image depth.

30.45.3.6 `void FL_Image::d (int D)` [inline, protected]

The first form of the `d()` method returns the current image depth.

The return value will be 0 for bitmaps, 1 for pixmaps, and 1 to 4 for color images.

The second form is a protected method that sets the current image depth.

30.45.3.7 `const char* const* FL_Image::data () const` [inline]

The first form of the `data()` method returns a pointer to the current image data array.

Use the `count()` method to find the size of the data array.

The second form is a protected method that sets the current array pointer and count of pointers in the array.

30.45.3.8 `void FL_Image::desaturate ()` [virtual]

The `desaturate()` method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved. *This method does not alter the original image data.*

Reimplemented in [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.9 `void FL_Image::draw (int X, int Y)` [inline]

The `draw()` methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented in [FL_Bitmap](#), [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.10 `void FL_Image::draw (int X, int Y, int W, int H, int cx = 0, int cy = 0)` [virtual]

The `draw()` methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the `cx` and `cy` arguments.

Reimplemented in [FL_Bitmap](#), [FL_RGB_Image](#), [FL_Pixmap](#), [FL_Shared_Image](#), and [FL_Tiled_Image](#).

30.45.3.11 `void FL_Image::draw_empty (int X, int Y)` [protected]

The protected method `draw_empty()` draws a box with an X in it.

It can be used to draw any image that lacks image data.

30.45.3.12 void Fl_Image::h (int *H*) [inline, protected]

The first form of the [h\(\)](#) method returns the current image height in pixels.

The second form is a protected method that sets the current image height.

30.45.3.13 void Fl_Image::inactive () [inline]

The [inactive\(\)](#) method calls `color_average(FL_BACKGROUND_COLOR, 0.33f)` to produce an image that appears grayed out.

This method does not alter the original image data.

30.45.3.14 void Fl_Image::label (Fl_Menu_Item * *m*) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.

Use the [image\(\)](#) or [deimage\(\)](#) methods of the [Fl_Widget](#) and [Fl_Menu_Item](#) classes instead.

Reimplemented in [Fl_Bitmap](#), [Fl_RGB_Image](#), and [Fl_Pixmap](#).

30.45.3.15 void Fl_Image::label (Fl_Widget * *widget*) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.

Use the [image\(\)](#) or [deimage\(\)](#) methods of the [Fl_Widget](#) and [Fl_Menu_Item](#) classes instead.

Reimplemented in [Fl_Bitmap](#), [Fl_RGB_Image](#), and [Fl_Pixmap](#).

30.45.3.16 int Fl_Image::ld () const [inline]

The first form of the [ld\(\)](#) method returns the current line data size in bytes.

Line data is extra data that is included after each line of color image data and is normally not present.

The second form is a protected method that sets the current line data size in bytes.

30.45.3.17 void Fl_Image::uncache () [virtual]

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented in [Fl_Bitmap](#), [Fl_RGB_Image](#), [Fl_Pixmap](#), and [Fl_Shared_Image](#).

30.45.3.18 void Fl_Image::w (int *W*) [inline, protected]

The first form of the [w\(\)](#) method returns the current image width in pixels.

The second form is a protected method that sets the current image width.

The documentation for this class was generated from the following files:

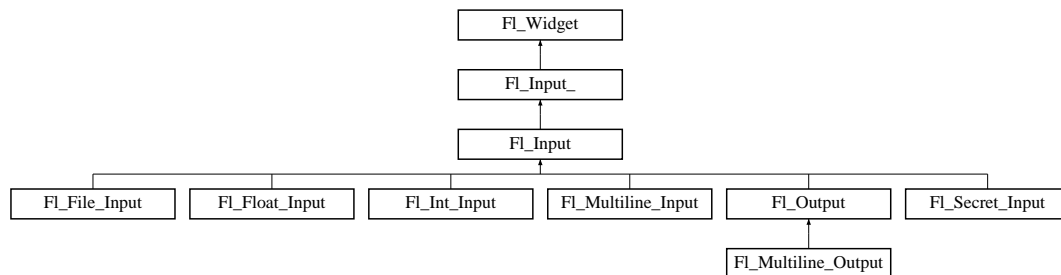
- [Fl_Image.H](#)
- [Fl_Image.cxx](#)

30.46 Fl_Input Class Reference

This is the FLTK text input widget.

```
#include <Fl_Input.H>
```

Inheritance diagram for Fl_Input::



Public Member Functions

- [Fl_Input](#) (int, int, int, int, const char * = 0)

Creates a new [Fl_Input](#) widget using the given position, size, and label string.

- int [handle](#) (int)

Handles the specified event.

Protected Member Functions

- void [draw](#) ()

Draws the widget.

30.46.1 Detailed Description

This is the FLTK text input widget.

It displays a single line of text and lets the user edit it. Normally it is drawn with an inset box and a white background. The text may contain any characters (even 0), and will correctly display anything, using ^X notation for unprintable control characters and \nnn notation for unprintable characters with the high bit set. It assumes the font can draw any characters in the ISO-8859-1 character set.

Mouse button 1	Moves the cursor to this point. Drag selects characters. Double click selects words. Triple click selects all text. Shift+click extends the selection. When you select text it is automatically copied to the clipboard.
Mouse button 2	Insert the clipboard at the point clicked. You can also select a region and replace it with the clipboard by selecting the region with mouse button 2.
Mouse button 3	Currently acts like button 1.
Backspace	Deletes one character to the left, or deletes the selected region.
Enter	May cause the callback, see when() .
^A or Home	Go to start of line.
^B or Left	Move left
^C	Copy the selection to the clipboard
^D or Delete	Deletes one character to the right or deletes the selected region.
^E or End	Go to the end of line.
^F or Right	Move right
^K	Delete to the end of line (next <code>\n</code> character) or deletes a single <code>\n</code> character. These deletions are all concatenated into the clipboard.
^N or Down	Move down (for Fl_Multiline_Input only, otherwise it moves to the next input field).
^P or Up	Move up (for Fl_Multiline_Input only, otherwise it moves to the previous input field).
^U	Delete everything.
^V or ^Y	Paste the clipboard
^X or ^W	Copy the region to the clipboard and delete it.
^Z or ^_	Undo. This is a single-level undo mechanism, but all adjacent deletions and insertions are concatenated into a single "undo". Often this will undo a lot more than you expected.
Shift+move	Move the cursor but also extend the selection.
RightCtrl or Compose	<p>Start a compose-character sequence. The next one or two keys typed define the character to insert (see table that follows.)</p> <p>For instance, to type <code>''</code> type <code>[compose][a][']</code> or <code>[compose][']</code><code>[a]</code>.</p> <p>The character <code>"nbsp"</code> (non-breaking space) is typed by using <code>[compose][space]</code>.</p> <p>The single-character sequences may be followed by a space if necessary to remove ambiguity. For instance, if you really want to type <code>"~"</code> rather than <code>""</code> you must type <code>[compose][a][space][~]</code>.</p> <p>The same key may be used to "quote" control characters into the text. If you need a <code>^Q</code> character you can get one by typing <code>[compose][Control+Q]</code>.</p> <p>X may have a key on the keyboard defined as <code>XK_Multi_key</code>. If so this key may be used as well as the right-hand control key. You can set this up with the program <code>xmodmap</code>.</p> <p>If your keyboard is set to support a foreign language you should also be able to type "dead key" prefix characters. On X you will actually be able to see what dead key you typed, and if you</p>
Generated on Mon Mar 23 19:19:46 2009 for FLTK by Doxygen	

Keys	Char	Keys	Char	Keys	Char	Keys	Char	Keys	Char	Keys	Char
sp	nbsp	*	~ N	' A	~ n	D -	' a	-	!	A ^	+ -
^ a	' A	#	^ O	' a	^ o	%	' O	2	o	\$	' O
,	o	: A		3		~ A		~ a			
y =		u		* A	~ O	* a	~ o				
		p		A E	: O	a e	o				
&		.		, C	x	, c	:				
:		,		E '	O /	' e	/				
c		l		' E	' U	' e	u				
a		o		^ E	' U	^ e	u				
< <		> >		: E	^ U	: e	^ u				
~		1 4		' I	: U	' i	u				
-		1 2		' I	' Y	' i	y				
r		3 4		^ I	T H	^ i	h				
_		?		: I	s s	: i	y				

Table 30.1: Character Composition Table

30.46.2 Constructor & Destructor Documentation

30.46.2.1 `Fl_Input::Fl_Input (int X, int Y, int W, int H, const char *l = 0)`

Creates a new [Fl_Input](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

30.46.3 Member Function Documentation

30.46.3.1 `void Fl_Input::draw ()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

Reimplemented in [Fl_File_Input](#).

30.46.3.2 `int Fl_Input::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

Reimplemented in [FL_File_Input](#).

The documentation for this class was generated from the following files:

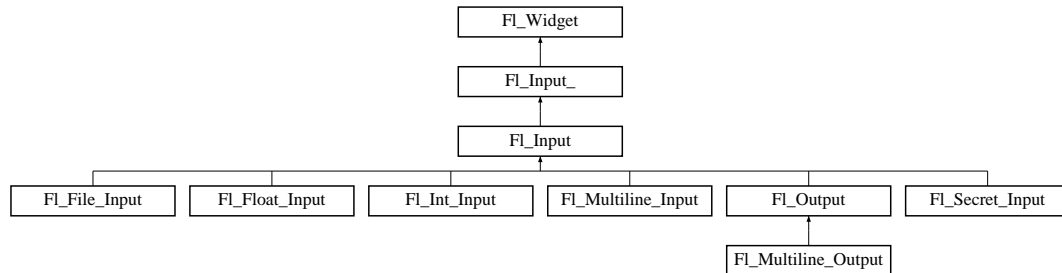
- [FL_Input.H](#)
- [FL_Input.cxx](#)

30.47 Fl_Input_ Class Reference

This is a virtual base class below [Fl_Input](#).

```
#include <Fl_Input_.H>
```

Inheritance diagram for `Fl_Input_`:



Public Member Functions

- `int copy` (int clipboard)
Put the current selection between [mark\(\)](#) and [position\(\)](#) into the specified clipboard.
- `int copy_cuts` ()
Copy all the previous contiguous cuts from the undo information to the clipboard.
- `void cursor_color` (unsigned n)
Sets the color of the cursor.
- `Fl_Color cursor_color` () const
Gets the color of the cursor.
- `int cut` (int a, int b)
See `int Fl_Input_::cut()`.
- `int cut` (int n)
See `int Fl_Input_::cut()`.
- `int cut` ()
Deletes the current selection.
- `Fl_Input_` (int, int, int, int, const char * = 0)
Creates a new [Fl_Input_](#) widget using the given position, size, and label string.
- `char index` (int i) const
Same as [value\(\)](#)[n], but may be faster in plausible implementations.
- `void input_type` (int t)
Sets the input field type.
- `int input_type` () const

Gets the input field type.

- `int insert (const char *t, int l=0)`
Insert the string t at the current position, and leave the mark and position after it.
- `int mark (int m)`
Sets the current selection mark.
- `int mark () const`
Gets the current selection mark.
- `void maximum_size (int m)`
Sets the maximum length of the input field.
- `int maximum_size () const`
Gets the maximum length of the input field.
- `int position (int p)`
See int Fl_Input_::position() const.
- `int position (int p, int m)`
See int Fl_Input_::position() const.
- `int position () const`
The input widget maintains two pointers into the string.
- `void readonly (int b)`
Sets the read-only state of the input field.
- `int readonly () const`
Gets the read-only state of the input field.
- `int replace (int, int, const char *, int=0)`
This call does all editing of the text.
- `void resize (int, int, int, int)`
Changes the size or position of the widget.
- `void shortcut (int s)`
See int shortcut() const.
- `int shortcut () const`
The first form returns the current shortcut key for the Input.
- `void size (int W, int H)`
Changes the size of the widget.
- `int size () const`
Returns the number of characters in value().

- `int static_value (const char *, int)`
See `int Fl_Input::static_value(const char)`.*
- `int static_value (const char *)`
Change the text and set the mark and the point to the end of it.
- `void textcolor (unsigned n)`
Sets the color of the text in the input field.
- `Fl_Color textcolor () const`
Gets the color of the text in the input field.
- `void textfont (Fl_Font s)`
Sets the font of the text in the input field.
- `Fl_Font textfont () const`
Gets the font of the text in the input field.
- `void textsize (Fl_Fontsize s)`
Sets the size of the text in the input field.
- `Fl_Fontsize textsize () const`
Gets the size of the text in the input field.
- `int undo ()`
Does undo of several previous calls to `replace()`.
- `const char * value () const`
The first form returns the current value, which is a pointer to the internal buffer and is valid only until the next event is handled.
- `int value (const char *, int)`
*See `const char *Fl_Input_value()` const.*
- `int value (const char *)`
*See `const char *Fl_Input_value()` const.*
- `void wrap (int b)`
Sets the word wrapping state of the input field.
- `int wrap () const`
Gets the word wrapping state of the input field.

Protected Member Functions

- `void drawtext (int, int, int, int)`
Draw the text in the passed bounding box.
- `void handle_mouse (int, int, int, int, int keepmark=0)`

- int **handletext** (int e, int, int, int, int)
- int **line_end** (int i) const
- int **line_start** (int i) const
- void **maybe_do_callback** ()
- int **up_down_position** (int, int keepmark=0)
Do the correct thing for arrow keys.

- int **word_end** (int i) const
- int **word_start** (int i) const
- int **xscroll** () const
- int **yscroll** () const

30.47.1 Detailed Description

This is a virtual base class below [Fl_Input](#).

It has all the same interfaces, but lacks the [handle\(\)](#) and [draw\(\)](#) method. You may want to subclass it if you are one of those people who likes to change how the editing keys work.

This can act like any of the subclasses of [Fl_Input](#), by setting [type\(\)](#) to one of the following values:

```
#define FL_NORMAL_INPUT          0
#define FL_FLOAT_INPUT          1
#define FL_INT_INPUT            2
#define FL_MULTILINE_INPUT      4
#define FL_SECRET_INPUT         5
#define FL_INPUT_TYPE           7
#define FL_INPUT_READONLY       8
#define FL_NORMAL_OUTPUT        (FL_NORMAL_INPUT | FL_INPUT_READONLY)
#define FL_MULTILINE_OUTPUT     (FL_MULTILINE_INPUT | FL_INPUT_READONLY)
#define FL_INPUT_WRAP           16
#define FL_MULTILINE_INPUT_WRAP (FL_MULTILINE_INPUT | FL_INPUT_WRAP)
#define FL_MULTILINE_OUTPUT_WRAP (FL_MULTILINE_INPUT | FL_INPUT_READONLY | FL_INPUT_WRAP)
```

30.47.2 Constructor & Destructor Documentation

30.47.2.1 [Fl_Input_::Fl_Input_](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [Fl_Input](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

30.47.3 Member Function Documentation

30.47.3.1 [int Fl_Input_::copy](#) (int *clipboard*)

Put the current selection between [mark\(\)](#) and [position\(\)](#) into the specified clipboard.

Does not replace the old clipboard contents if [position\(\)](#) and [mark\(\)](#) are equal. Clipboard 0 maps to the current text selection and clipboard 1 maps to the cut/paste clipboard.

30.47.3.2 [int Fl_Input_::copy_cuts](#) ()

Copy all the previous contiguous cuts from the undo information to the clipboard.

This is used to make ^K work.

30.47.3.3 `void FL_Input_::cursor_color (unsigned n)` [inline]

Sets the color of the cursor.

This is black by default.

30.47.3.4 `FL_Color FL_Input_::cursor_color () const` [inline]

Gets the color of the cursor.

This is black by default.

30.47.3.5 `int FL_Input_::cut ()` [inline]

Deletes the current selection.

cut(*n*) deletes *n* characters after the [position\(\)](#). cut(-*n*) deletes *n* characters before the [position\(\)](#). cut(*a*,*b*) deletes the characters between offsets *a* and *b*. *A*, *b*, and *n* are all clamped to the size of the string. The mark and point are left where the deleted text was.

If you want the data to go into the clipboard, do [FL_Input_::copy\(\)](#) before calling [FL_Input_::cut\(\)](#), or do [FL_Input_::copy_cuts\(\)](#) afterwards.

30.47.3.6 `void FL_Input_::drawtext (int X, int Y, int W, int H)` [protected]

Draw the text in the passed bounding box.

If [damage\(\)](#) & FL_DAMAGE_ALL is true, this assumes the area has already been erased to [color\(\)](#). Otherwise it does minimal update and erases the area itself.

30.47.3.7 `char FL_Input_::index (int i) const` [inline]

Same as [value\(\)\[n\]](#), but may be faster in plausible implementations.

No bounds checking is done.

30.47.3.8 `void FL_Input_::input_type (int t)` [inline]

Sets the input field type.

30.47.3.9 `int FL_Input_::input_type () const` [inline]

Gets the input field type.

30.47.3.10 `int FL_Input_::insert (const char * t, int l = 0)` [inline]

Insert the string *t* at the current position, and leave the mark and position after it.

If *l* is not zero then it is assumed to be strlen(*t*).

30.47.3.11 int Fl_Input_::mark (int *m*) [inline]

Sets the current selection mark.

mark(*n*) is the same as position([position\(\)](#),*n*).

30.47.3.12 int Fl_Input_::mark () const [inline]

Gets the current selection mark.

mark(*n*) is the same as position([position\(\)](#),*n*).

30.47.3.13 void Fl_Input_::maximum_size (int *m*) [inline]

Sets the maximum length of the input field.

30.47.3.14 int Fl_Input_::maximum_size () const [inline]

Gets the maximum length of the input field.

30.47.3.15 int Fl_Input_::position () const [inline]

The input widget maintains two pointers into the string.

The "position" is where the cursor is. The "mark" is the other end of the selected text. If they are equal then there is no selection. Changing this does not affect the clipboard (use [copy\(\)](#) to do that).

Changing these values causes a [redraw\(\)](#). The new values are bounds checked. The return value is non-zero if the new position is different than the old one. position(*n*) is the same as position(*n*,*n*). mark(*n*) is the same as position([position\(\)](#),*n*).

30.47.3.16 void Fl_Input_::readonly (int *b*) [inline]

Sets the read-only state of the input field.

30.47.3.17 int Fl_Input_::readonly () const [inline]

Gets the read-only state of the input field.

30.47.3.18 int Fl_Input_::replace (int *b*, int *e*, const char * *text*, int *ilen* = 0)

This call does all editing of the text.

It deletes the region between *a* and *b* (either one may be less or equal to the other), and then inserts the string *insert* at that point and leaves the [mark\(\)](#) and [position\(\)](#) after the insertion. Does the callback if [when\(\)](#) & FL_WHEN_CHANGED and there is a change.

Set start and end equal to not delete anything. Set *insert* to NULL to not insert anything.

length must be zero or strlen(*insert*), this saves a tiny bit of time if you happen to already know the length of the insertion, or can be used to insert a portion of a string or a string containing nul's.

a and *b* are clamped to the 0..[size\(\)](#) range, so it is safe to pass any values.

[cut\(\)](#) and [insert\(\)](#) are just inline functions that call [replace\(\)](#).

30.47.3.19 void FL_Input_::resize (int x, int y, int w, int h) [virtual]

Changes the size or position of the widget.

This is a virtual function so that the widget may implement its own handling of resizing. The default version does *not* call the [redraw\(\)](#) method, but instead relies on the parent widget to do so because the parent may know a faster way to update the display, such as scrolling from the old position.

Some window managers under X11 call [resize\(\)](#) a lot more often than needed. Please verify that the position or size of a widget did actually change before doing any extensive calculations.

[position\(X, Y\)](#) is a shortcut for [resize\(X, Y, w\(\), h\(\)\)](#), and [size\(W, H\)](#) is a shortcut for [resize\(x\(\), y\(\), W, H\)](#).

Parameters:

- ← *x,y* new position relative to the parent window
- ← *w,h* new size

See also:

[position\(int,int\)](#), [size\(int,int\)](#)

Reimplemented from [FL_Widget](#).

30.47.3.20 int FL_Input_::shortcut () const [inline]

The first form returns the current shortcut key for the Input.

The second form sets the shortcut key to key. Setting this overrides the use of '&' in the [label\(\)](#). The value is a bitwise OR of a key and a set of shift flags, for example [FL_ALT](#) | 'a', [FL_ALT](#) | ([FL_F](#) + 10), or just 'a'. A value of 0 disables the shortcut.

The key can be any value returned by [FL::event_key\(\)](#), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by [FL::event_state\(\)](#). If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

30.47.3.21 void FL_Input_::size (int W, int H) [inline]

Changes the size of the widget.

[size\(W, H\)](#) is a shortcut for [resize\(x\(\), y\(\), W, H\)](#).

Parameters:

- ← *W,H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented from [FL_Widget](#).

30.47.3.22 `int Fl_Input_::size () const` [inline]

Returns the number of characters in `value()`.

This may be greater than `strlen(value())` if there are nul characters in it.

30.47.3.23 `int Fl_Input_::static_value (const char * str)`

Change the text and set the mark and the point to the end of it.

The string is *not* copied. If the user edits the string it is copied to the internal buffer then. This can save a great deal of time and memory if your program is rapidly changing the values of text fields, but this will only work if the passed string remains unchanged until either the `Fl_Input` is destroyed or `value()` is called again.

30.47.3.24 `void Fl_Input_::textcolor (unsigned n)` [inline]

Sets the color of the text in the input field.

30.47.3.25 `Fl_Color Fl_Input_::textcolor () const` [inline]

Gets the color of the text in the input field.

30.47.3.26 `void Fl_Input_::textfont (Fl_Font s)` [inline]

Sets the font of the text in the input field.

30.47.3.27 `Fl_Font Fl_Input_::textfont () const` [inline]

Gets the font of the text in the input field.

30.47.3.28 `void Fl_Input_::textsize (Fl_Fontsize s)` [inline]

Sets the size of the text in the input field.

30.47.3.29 `Fl_Fontsize Fl_Input_::textsize () const` [inline]

Gets the size of the text in the input field.

30.47.3.30 `int Fl_Input_::undo ()`

Does undo of several previous calls to `replace()`.

Returns non-zero if any change was made.

30.47.3.31 `int Fl_Input_::up_down_position (int i, int keepmark = 0)` [protected]

Do the correct thing for arrow keys.

Sets the position (and mark if *keepmark* is zero) to somewhere in the same line as *i*, such that pressing the arrows repeatedly will cause the point to move up and down.

30.47.3.32 `const char* Fl_Input_::value () const` [inline]

The first form returns the current value, which is a pointer to the internal buffer and is valid only until the next event is handled.

The second two forms change the text and set the mark and the point to the end of it. The string is copied to the internal buffer. Passing NULL is the same as "". This returns non-zero if the new value is different than the current one. You can use the second version to directly set the length if you know it already or want to put nul's in the text.

30.47.3.33 `void Fl_Input_::wrap (int b)` [inline]

Sets the word wrapping state of the input field.

Word wrap is only functional with multi-line input fields.

30.47.3.34 `int Fl_Input_::wrap () const` [inline]

Gets the word wrapping state of the input field.

Word wrap is only functional with multi-line input fields.

The documentation for this class was generated from the following files:

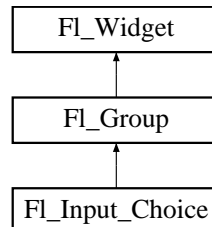
- `Fl_Input_.H`
- `Fl_Input_.cxx`

30.48 Fl_Input_Choice Class Reference

A combination of the input widget and a menu button.

```
#include <Fl_Input_Choice.H>
```

Inheritance diagram for Fl_Input_Choice::



Classes

- class **InputMenuButton**

Public Member Functions

- void **add** (const char *s)
Adds an item to the menu.
- int **changed** () const
Checks if the widget value changed since the last callback.
- void **clear** ()
Removes all items from the menu.
- void **clear_changed** ()
Marks the value of the widget as unchanged.
- void **down_box** (Fl_Boxtype b)
Sets the box type of the menu button.
- Fl_Boxtype **down_box** () const
Gets the box type of the menu button.
- Fl_Input_Choice (int x, int y, int w, int h, const char *l=0)
Creates a new Fl_Input_Choice widget using the given position, size, and label string.
- Fl_Input * **input** ()
Returns a reference to the internal Fl_Input widget.
- void **menu** (const Fl_Menu_Item *m)
Sets the Fl_Menu_Item array used for the menu.
- const Fl_Menu_Item * **menu** ()

Gets the [FL_Menu_Item](#) array used for the menu.

- [FL_Menu_Button](#) * [menubutton](#) ()
Returns a reference to the internal [FL_Menu_Button](#) widget.
- void [resize](#) (int X, int Y, int W, int H)
Resizes the [FL_Group](#) widget and all of its children.
- void [set_changed](#) ()
Marks the value of the widget as changed.
- void [textcolor](#) ([FL_Color](#) c)
Sets the encapsulated input text color attributes.
- [FL_Color](#) [textcolor](#) () const
Gets the encapsulated input text color attributes.
- void [textfont](#) ([FL_Font](#) f)
Sets the encapsulated input text font attributes.
- [FL_Font](#) [textfont](#) () const
Gets the encapsulated input text font attributes.
- void [textsize](#) ([FL_Fontsize](#) s)
Sets the encapsulated input size attributes.
- [FL_Fontsize](#) [textsize](#) () const
Gets the encapsulated input size attributes.
- void [value](#) (int val)
*See void [FL_Input_Choice::value\(const char *s\)](#).*
- void [value](#) (const char *val)
Sets or returns the input widget's current contents.
- const char * [value](#) () const
*See void [FL_Input_Choice::value\(const char *s\)](#).*

30.48.1 Detailed Description

A combination of the input widget and a menu button.

The user can either type into the input area, or use the menu button chooser on the right, which loads the input area with predefined text. Normally it is drawn with an inset box and a white background.

The application can directly access both the input and menu widgets directly, using the [menubutton\(\)](#) and [input\(\)](#) accessor methods.

30.48.2 Constructor & Destructor Documentation

30.48.2.1 `FL_Input_Choice::FL_Input_Choice (int x, int y, int w, int h, const char * l = 0)` `[inline]`

Creates a new [FL_Input_Choice](#) widget using the given position, size, and label string.

Inherited destructor Destroys the widget and any value associated with it.

30.48.3 Member Function Documentation

30.48.3.1 `void FL_Input_Choice::add (const char * s)` `[inline]`

Adds an item to the menu.

30.48.3.2 `int FL_Input_Choice::changed () const` `[inline]`

Checks if the widget value changed since the last callback.

"Changed" is a flag that is turned on when the user changes the value stored in the widget. This is only used by subclasses of [FL_Widget](#) that store values, but is in the base class so it is easier to scan all the widgets in a panel and [do_callback\(\)](#) on the changed ones in response to an "OK" button.

Most widgets turn this flag off when they do the callback, and when the program sets the stored value.

Return values:

0 if the value did not change

See also:

[set_changed\(\)](#), [clear_changed\(\)](#)

Reimplemented from [FL_Widget](#).

30.48.3.3 `void FL_Input_Choice::clear ()` `[inline]`

Removes all items from the menu.

Reimplemented from [FL_Group](#).

30.48.3.4 `void FL_Input_Choice::clear_changed ()` `[inline]`

Marks the value of the widget as unchanged.

See also:

[changed\(\)](#), [set_changed\(\)](#)

Reimplemented from [FL_Widget](#).

30.48.3.5 `FL_Input* FL_Input_Choice::input ()` `[inline]`

Returns a reference to the internal [FL_Input](#) widget.

30.48.3.6 void FL_Input_Choice::menu (const FL_Menu_Item * *m*) [inline]

Sets the [FL_Menu_Item](#) array used for the menu.

30.48.3.7 const FL_Menu_Item* FL_Input_Choice::menu () [inline]

Gets the [FL_Menu_Item](#) array used for the menu.

30.48.3.8 FL_Menu_Button* FL_Input_Choice::menubutton () [inline]

Returns a reference to the internal [FL_Menu_Button](#) widget.

30.48.3.9 void FL_Input_Choice::resize (int *X*, int *Y*, int *W*, int *H*) [inline, virtual]

Resizes the [FL_Group](#) widget and all of its children.

The [FL_Group](#) widget first resizes itself, and then it moves and resizes all its children according to the rules documented for [FL_Group::resizable\(FL_Widget*\)](#)

See also:

[FL_Group::resizable\(FL_Widget*\)](#)
[FL_Group::resizable\(\)](#)
[FL_Widget::resize\(int,int,int,int\)](#)

Reimplemented from [FL_Group](#).

30.48.3.10 void FL_Input_Choice::set_changed () [inline]

Marks the value of the widget as changed.

See also:

[changed\(\)](#), [clear_changed\(\)](#)

Reimplemented from [FL_Widget](#).

30.48.3.11 void FL_Input_Choice::value (const char * *val*) [inline]

Sets or returns the input widget's current contents.

The second form sets the contents using the index into the menu which you can set as an integer. Setting the value effectively 'chooses' this menu item, and sets it as the new input text, deleting the previous text.

The documentation for this class was generated from the following file:

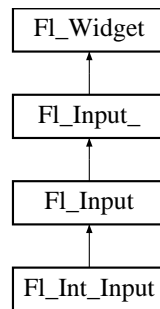
- [FL_Input_Choice.H](#)

30.49 Fl_Int_Input Class Reference

The [Fl_Int_Input](#) class is a subclass of [Fl_Input](#) that only allows the user to type decimal digits (or hex numbers of the form 0xae).

```
#include <Fl_Int_Input.H>
```

Inheritance diagram for Fl_Int_Input::



Public Member Functions

- [Fl_Int_Input](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [Fl_Int_Input](#) widget using the given position, size, and label string.

30.49.1 Detailed Description

The [Fl_Int_Input](#) class is a subclass of [Fl_Input](#) that only allows the user to type decimal digits (or hex numbers of the form 0xae).

30.49.2 Constructor & Destructor Documentation

30.49.2.1 Fl_Int_Input::Fl_Int_Input (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [Fl_Int_Input](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

Inherited destructor Destroys the widget and any value associated with it.

The documentation for this class was generated from the following file:

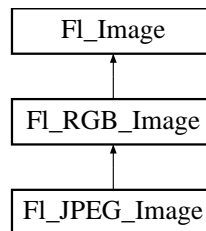
- `Fl_Int_Input.H`

30.50 Fl_JPEG_Image Class Reference

The [Fl_JPEG_Image](#) class supports loading, caching, and drawing of Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) images.

```
#include <Fl_JPEG_Image.H>
```

Inheritance diagram for Fl_JPEG_Image::



Public Member Functions

- [Fl_JPEG_Image](#) (const char *filename)

The constructor loads the JPEG image from the given jpeg filename.

30.50.1 Detailed Description

The [Fl_JPEG_Image](#) class supports loading, caching, and drawing of Joint Photographic Experts Group (JPEG) File Interchange Format (JFIF) images.

The class supports grayscale and color (RGB) JPEG image files.

30.50.2 Constructor & Destructor Documentation

30.50.2.1 Fl_JPEG_Image::Fl_JPEG_Image (const char *jpeg)

The constructor loads the JPEG image from the given jpeg filename.

The inherited destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

- Fl_JPEG_Image.H
- Fl_JPEG_Image.cxx

30.51 Fl_Label Struct Reference

This struct stores all information for a text or mixed graphics label.

```
#include <Fl_Widget.H>
```

Public Member Functions

- void [draw](#) (int, int, int, int, [Fl_Align](#)) const
Draws the label aligned to the given box.
- void [measure](#) (int &w, int &h) const
Measures the size of the label.

Public Attributes

- unsigned [color](#)
text color
- [Fl_Image](#) * [deimage](#)
optional image for a deactivated label
- [Fl_Font](#) [font](#)
label font used in text
- [Fl_Image](#) * [image](#)
optional image for an active label
- [Fl_Fontsize](#) [size](#)
size of label font
- [uchar](#) [type](#)
type of label.
- const char * [value](#)
label text

30.51.1 Detailed Description

This struct stores all information for a text or mixed graphics label.

Todo

For FLTK 1.3, the [Fl_Label](#) type will become a widget by itself. That way we will be avoiding a lot of code duplication by handling labels in a similar fashion to widgets containing text. We also provide an easy interface for very complex labels, containing html or vector graphics.

30.51.2 Member Function Documentation

30.51.2.1 void Fl_Label::draw (int *X*, int *Y*, int *W*, int *H*, Fl_Align *align*) const

Draws the label aligned to the given box.

Draws a label with arbitrary alignment in an arbitrary box.

30.51.2.2 void Fl_Label::measure (int & *W*, int & *H*) const

Measures the size of the label.

Parameters:

↔ *W,H* : this is the requested size for the label text plus image; on return, this will contain the size needed to fit the label

30.51.3 Member Data Documentation

30.51.3.1 uchar Fl_Label::type

type of label.

See also:

[Fl_Labeltype](#)

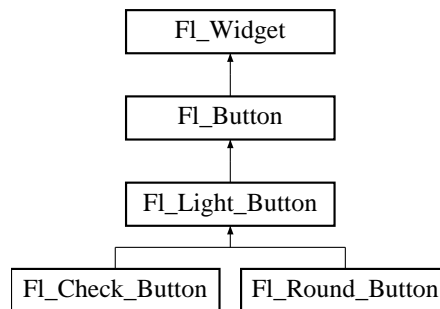
The documentation for this struct was generated from the following files:

- Fl_Widget.H
- fl_labeltype.cxx

30.52 Fl_Light_Button Class Reference

```
#include <Fl_Light_Button.H>
```

Inheritance diagram for Fl_Light_Button::



Public Member Functions

- [Fl_Light_Button](#) (int x, int y, int w, int h, const char *l=0)

Creates a new [Fl_Light_Button](#) widget using the given position, size, and label string.

- virtual int [handle](#) (int)

Handles the specified event.

Protected Member Functions

- virtual void [draw](#) ()

Draws the widget.

30.52.1 Detailed Description

This subclass displays the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_DOWN_BOX. The color of the light when on is controlled with [selection_color\(\)](#), which defaults to FL_YELLOW.

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for [type\(\)](#) and [when\(\)](#).

Figure 30.17: Fl_Light_Button

30.52.2 Constructor & Destructor Documentation

30.52.2.1 Fl_Light_Button::Fl_Light_Button (int *X*, int *Y*, int *W*, int *H*, const char **l* = 0)

Creates a new [Fl_Light_Button](#) widget using the given position, size, and label string.

The destructor deletes the check button.

30.52.3 Member Function Documentation

30.52.3.1 void Fl_Light_Button::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;          // scroll is an embedded Fl_Scrollbar
s->draw();                        // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Button](#).

30.52.3.2 int Fl_Light_Button::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0* if the event was not used or understood
- 1* if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Button](#).

The documentation for this class was generated from the following files:

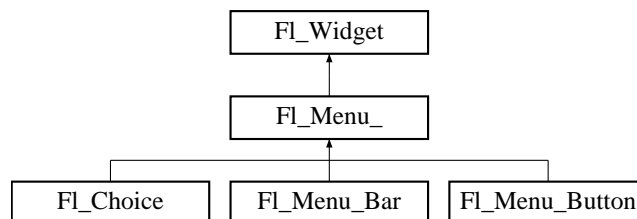
- Fl_Light_Button.H
- Fl_Light_Button.cxx

30.53 Fl_Menu_ Class Reference

Base class of all widgets that have a menu in FLTK.

```
#include <Fl_Menu_.H>
```

Inheritance diagram for Fl_Menu_:



Public Member Functions

- `int add (const char *)`
This is a Forms (and SGI GL library) compatible add function, it adds many menu items, with '|' separating the menu items, and tab separating the menu item names from an optional shortcut string.
- `int add (const char *a, const char *b, Fl_Callback *c, void *d=0, int e=0)`
See int [Fl_Menu_::add\(const char label, int shortcut, Fl_Callback*, void *user_data=0, int flags=0\)](#).*
- `int add (const char *, int shortcut, Fl_Callback *, void *=0, int=0)`
Adds a new menu item, with a title string, shortcut int (or string), callback, argument to the callback, and flags.
- `void clear ()`
Same as menu(NULL), set the array pointer to null, indicating a zero-length menu.
- `void copy (const Fl_Menu_Item *m, void *user_data=0)`
Sets the menu array pointer with a copy of m that will be automatically deleted.
- `void down_box (Fl_Boxtype b)`
See Fl_Boxtype [Fl_Menu_::down_box\(\)](#) const.
- `Fl_Boxtype down_box () const`
This box type is used to surround the currently-selected items in the menus.
- `void down_color (unsigned c)`
For back compatibility, same as [selection_color\(\)](#).
- `Fl_Color down_color () const`
For back compatibility, same as [selection_color\(\)](#).
- `const Fl_Menu_Item * find_item (const char *name)`
Find menu item index, given menu pathname eg.

- `Fl_Menu_ (int, int, int, int, const char *=0)`
Creates a new `Fl_Menu_` widget using the given position, size, and label string.
- `void global ()`
Make the shortcuts for this menu work no matter what window has the focus when you type it.
- `int item_pathname (char *name, int namelen, const Fl_Menu_Item *finditem=0) const`
Set 'pathname' of specified menuitem. If finditem==NULL, `mvalue()` is used (the most recently picked menuitem). Returns:
 - 0 : OK
 - -1 : item not found (name="")
 - -2 : 'name' not large enough (name="").
- `void menu (const Fl_Menu_Item *m)`
Sets the menu array pointer directly.
- `const Fl_Menu_Item * menu () const`
Returns a pointer to the array of `Fl_Menu_Items`.
- `int mode (int i) const`
Gets the flags of item i.
- `void mode (int i, int fl)`
Sets the flags of item i.
- `const Fl_Menu_Item * mvalue () const`
Returns a pointer to the last menu item that was picked.
- `const Fl_Menu_Item * picked (const Fl_Menu_Item *)`
When user picks a menu item, call this.
- `void remove (int)`
Deletes item i from the menu.
- `void replace (int, const char *)`
Changes the text of item i.
- `void shortcut (int i, int s)`
Changes the shortcut of item i to n.
- `void size (int W, int H)`
Changes the size of the widget.
- `int size () const`
This returns the number of `Fl_Menu_Item` structures that make up the menu, correctly counting submenus.
- `const Fl_Menu_Item * test_shortcut ()`
Internal use only.
- `const char * text (int i) const`

Returns the title of the last item chosen, or of item i.

- `const char * text () const`

Returns the title of the last item chosen, or of item i.

- `void textcolor (unsigned c)`

Sets the current color of menu item labels.

- `Fl_Color textcolor () const`

Get the current color of menu item labels.

- `void textfont (Fl_Font c)`

Sets the current font of menu item labels.

- `Fl_Font textfont () const`

Gets the current font of menu item labels.

- `void textsize (Fl_Fontsize c)`

Sets the font size of menu item labels.

- `Fl_Fontsize textsize () const`

Gets the font size of menu item labels.

- `int value (int i)`

The value is the index into [menu\(\)](#) of the last item chosen by the user.

- `int value (const Fl_Menu_Item *)`

The value is the index into [menu\(\)](#) of the last item chosen by the user.

- `int value () const`

Returns the index into [menu\(\)](#) of the last item chosen by the user.

Protected Attributes

- `uchar alloc`
- `uchar down_box_`
- `unsigned textcolor_`
- `Fl_Font textfont_`
- `Fl_Fontsize textsize_`

30.53.1 Detailed Description

Base class of all widgets that have a menu in FLTK.

Currently FLTK provides you with [Fl_Menu_Button](#), [Fl_Menu_Bar](#), and [Fl_Choice](#).

The class contains a pointer to an array of structures of type [Fl_Menu_Item](#). The array may either be supplied directly by the user program, or it may be "private": a dynamically allocated array managed by the [Fl_Menu_](#).

30.53.2 Constructor & Destructor Documentation

30.53.2.1 Fl_Menu_::Fl_Menu_ (int X, int Y, int W, int H, const char * l = 0)

Creates a new [Fl_Menu_](#) widget using the given position, size, and label string.

[menu\(\)](#) is initialized to null.

30.53.3 Member Function Documentation

30.53.3.1 int Fl_Menu_::add (const char * str)

This is a Forms (and SGI GL library) compatible add function, it adds many menu items, with '|' separating the menu items, and tab separating the menu item names from an optional shortcut string.

The passed string is split at any '|' characters and then add(s,0,0,0,0) is done with each section. This is often useful if you are just using the value, and is compatible with Forms and other GL programs. The section strings use the same special characters as described for the long version of [add\(\)](#).

No items must be added to a menu during a callback to the same menu.

30.53.3.2 int Fl_Menu_::add (const char * t, int s, Fl_Callback * c, void * v = 0, int f = 0)

Adds a new menu item, with a title string, shortcut int (or string), callback, argument to the callback, and flags.

If the menu array was directly set with menu(x), then [copy\(\)](#) is done to make a private array.

The characters "&", "/", "\", and "_" are treated as special characters in the label string. The "&" character specifies that the following character is an accelerator and will be underlined. The "\" character is used to escape the next character in the string. Labels starting with the "_" character cause a divider to be placed after that menu item.

A label of the form "foo/bar/baz" will create submenus called "foo" and "bar" with an entry called "baz". The "/" character is ignored if it appears as the first character of the label string, e.g. "/foo/bar/baz".

The label string is copied to new memory and can be freed. The other arguments (including the shortcut) are copied into the menu item unchanged.

If an item exists already with that name then it is replaced with this new one. Otherwise this new one is added to the end of the correct menu or submenu. The return value is the offset into the array that the new entry was placed at.

Shortcut can be 0L, or either a modifier/key combination (for example FL_CTRL+'A') or a string describing the shortcut in one of two ways:

```
[#+^]<ascii_value>    e.g. "97", "^97", "+97", "#97"
[#+^]<ascii_char>     e.g. "a", "^a", "+a", "#a"
```

..where <ascii_value> is a decimal value representing an ascii character (eg. 97 is the ascii for 'a'), and the optional prefixes enhance the value that follows. Multiple prefixes must appear in the above order.

```
# - Alt
+ - Shift
^ - Control
```

Text shortcuts are converted to integer shortcut by calling `int fl_old_shortcut(const char*)`.

The return value is the index into the array that the entry was put.

No items must be added to a menu during a callback to the same menu.

30.53.3.3 void `Fl_Menu_::clear()`

Same as `menu(NULL)`, set the array pointer to null, indicating a zero-length menu.

Menus must not be cleared during a callback to the same menu.

30.53.3.4 void `Fl_Menu_::copy(const Fl_Menu_Item * m, void * ud = 0)`

Sets the menu array pointer with a copy of `m` that will be automatically deleted.

If `ud` is not NULL, then all user data pointers are changed in the menus as well. See void [Fl_Menu_::menu\(const Fl_Menu_Item* m\)](#).

30.53.3.5 `Fl_Boxtype Fl_Menu_::down_box()` const [inline]

This box type is used to surround the currently-selected items in the menus.

If this is `FL_NO_BOX` then it acts like `FL_THIN_UP_BOX` and `selection_color()` acts like `FL_WHITE`, for back compatibility.

30.53.3.6 const `Fl_Menu_Item * Fl_Menu_::find_item(const char * name)`

Find menu item index, given menu pathname eg.

"Edit/Copy" Will also return submenus, eg. "Edit" Returns NULL if not found.

30.53.3.7 void `Fl_Menu_::global()`

Make the shortcuts for this menu work no matter what window has the focus when you type it.

This is done by using [Fl::add_handler\(\)](#). This `Fl_Menu_` widget does not have to be visible (ie the window it is in can be hidden, or it does not have to be put in a window at all).

Currently there can be only one [global\(\)](#) menu. Setting a new one will replace the old one. There is no way to remove the [global\(\)](#) setting (so don't destroy the widget!)

30.53.3.8 void `Fl_Menu_::menu(const Fl_Menu_Item * m)`

Sets the menu array pointer directly.

If the old menu is private it is deleted. NULL is allowed and acts the same as a zero-length menu. If you try to modify the array (with [add\(\)](#), [replace\(\)](#), or [delete\(\)](#)) a private copy is automatically done.

30.53.3.9 const `Fl_Menu_Item* Fl_Menu_::menu()` const [inline]

Returns a pointer to the array of `Fl_Menu_Items`.

This will either be the value passed to `menu(value)` or the private copy.

30.53.3.10 `int Fl_Menu_::mode (int i) const` `[inline]`

Gets the flags of item *i*.

For a list of the flags, see [Fl_Menu_Item](#).

30.53.3.11 `void Fl_Menu_::mode (int i, int fl)` `[inline]`

Sets the flags of item *i*.

For a list of the flags, see [Fl_Menu_Item](#).

30.53.3.12 `const Fl_Menu_Item* Fl_Menu_::mvalue () const` `[inline]`

Returns a pointer to the last menu item that was picked.

30.53.3.13 `const Fl_Menu_Item * Fl_Menu_::picked (const Fl_Menu_Item * v)`

When user picks a menu item, call this.

It will do the callback. Unfortunately this also casts away `const` for the checkboxes, but this was necessary so non-checkbox menus can really be declared `const`...

30.53.3.14 `void Fl_Menu_::remove (int i)`

Deletes item *i* from the menu.

If the menu array was directly set with `menu(x)` then `copy()` is done to make a private array.

No items must be removed from a menu during a callback to the same menu.

30.53.3.15 `void Fl_Menu_::replace (int i, const char * str)`

Changes the text of item *i*.

This is the only way to get slash into an `add()`'ed menu item. If the menu array was directly set with `menu(x)` then `copy()` is done to make a private array.

30.53.3.16 `void Fl_Menu_::shortcut (int i, int s)` `[inline]`

Changes the shortcut of item *i* to *n*.

30.53.3.17 `void Fl_Menu_::size (int W, int H)` `[inline]`

Changes the size of the widget.

`size(W, H)` is a shortcut for `resize(x(), y(), W, H)`.

Parameters:

← *W,H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented from [FL_Widget](#).

30.53.3.18 `int FL_Menu_::size () const`

This returns the number of [FL_Menu_Item](#) structures that make up the menu, correctly counting submenus. This includes the "terminator" item at the end. To copy a menu array you need to copy `size()*sizeof(FL_Menu_Item)` bytes. If the menu is NULL this returns zero (an empty menu will return 1).

30.53.3.19 `const FL_Menu_Item* FL_Menu_::test_shortcut ()` [inline]

Internal use only.

Reimplemented from [FL_Widget](#).

30.53.3.20 `const char* FL_Menu_::text (int i) const` [inline]

Returns the title of the last item chosen, or of item i.

30.53.3.21 `const char* FL_Menu_::text () const` [inline]

Returns the title of the last item chosen, or of item i.

30.53.3.22 `void FL_Menu_::textcolor (unsigned c)` [inline]

Sets the current color of menu item labels.

30.53.3.23 `FL_Color FL_Menu_::textcolor () const` [inline]

Get the current color of menu item labels.

30.53.3.24 `void FL_Menu_::textfont (FL_Font c)` [inline]

Sets the current font of menu item labels.

30.53.3.25 `FL_Font FL_Menu_::textfont () const` [inline]

Gets the current font of menu item labels.

30.53.3.26 `void FL_Menu_::textsize (FL_Fontsize c)` [inline]

Sets the font size of menu item labels.

30.53.3.27 `Fl_Fontsize Fl_Menu_::textsize () const` `[inline]`

Gets the font size of menu item labels.

30.53.3.28 `int Fl_Menu_::value (int i)` `[inline]`

The value is the index into [menu\(\)](#) of the last item chosen by the user.

It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one.

Reimplemented in [Fl_Choice](#).

30.53.3.29 `int Fl_Menu_::value (const Fl_Menu_Item * m)`

The value is the index into [menu\(\)](#) of the last item chosen by the user.

It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one.

Reimplemented in [Fl_Choice](#).

30.53.3.30 `int Fl_Menu_::value () const` `[inline]`

Returns the index into [menu\(\)](#) of the last item chosen by the user.

It is zero initially.

Reimplemented in [Fl_Choice](#).

The documentation for this class was generated from the following files:

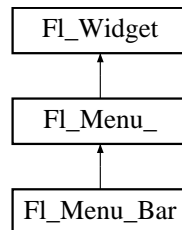
- `Fl_Menu_.H`
- `Fl_Menu_.cxx`
- `Fl_Menu_add.cxx`
- `Fl_Menu_global.cxx`

30.54 Fl_Menu_Bar Class Reference

This widget provides a standard menubar interface.

```
#include <Fl_Menu_Bar.H>
```

Inheritance diagram for Fl_Menu_Bar::



Public Member Functions

- [Fl_Menu_Bar](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Menu_Bar](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.54.1 Detailed Description

This widget provides a standard menubar interface.

Usually you will put this widget along the top edge of your window. The height of the widget should be 30 for the menu titles to draw correctly with the default font.

The items on the bar and the menus they bring up are defined by a single [Fl_Menu_Item](#) array. Because a [Fl_Menu_Item](#) array defines a hierarchy, the top level menu defines the items in the menubar, while the submenus define the pull-down menus. Sub-sub menus and lower pop up to the right of the submenus.

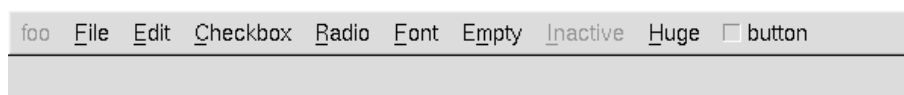


Figure 30.18: menubar

If there is an item in the top menu that is not a title of a submenu, then it acts like a "button" in the menubar. Clicking on it will pick it.

When the user picks an item off the menu, the item's callback is done with the menubar as the `Fl_Widget*` argument. If the item does not have a callback the menubar's callback is done instead.

Submenus will also pop up in response to shortcuts indicated by putting a '&' character in the name field of the menu item. If you put a '&' character in a top-level "button" then the shortcut picks it. The '&' character in submenus is ignored until the menu is popped up.

Typing the [shortcut\(\)](#) of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse.

30.54.2 Constructor & Destructor Documentation

30.54.2.1 `Fl_Menu_Bar::Fl_Menu_Bar (int X, int Y, int W, int H, const char *l = 0)` [inline]

Creates a new [Fl_Menu_Bar](#) widget using the given position, size, and label string.

The default boxtype is `FL_UP_BOX`.

The constructor sets [menu\(\)](#) to `NULL`. See [Fl_Menu_](#) for the methods to set or change the menu.

[labelsize\(\)](#), [labelfont\(\)](#), and [labelcolor\(\)](#) are used to control how the menubar items are drawn. They are initialized from the `Fl_Menu` static variables, but you can change them if desired.

[label\(\)](#) is ignored unless you change [align\(\)](#) to put it outside the menubar.

The destructor removes the [Fl_Menu_Bar](#) widget and all of its menu items.

30.54.3 Member Function Documentation

30.54.3.1 `void Fl_Menu_Bar::draw ()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.54.3.2 `int Fl_Menu_Bar::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0* if the event was not used or understood
- 1* if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

The documentation for this class was generated from the following files:

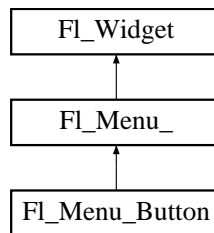
- Fl_Menu_Bar.H
- Fl_Menu_Bar.cxx

30.55 Fl_Menu_Button Class Reference

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of [Fl_Menu_Item](#) objects.

```
#include <Fl_Menu_Button.H>
```

Inheritance diagram for Fl_Menu_Button::



Public Types

- enum {
 POPUP1 = 1, **POPUP2**, **POPUP12**, **POPUP3**,
 POPUP13, **POPUP23**, **POPUP123** }

Public Member Functions

- [Fl_Menu_Button](#) (int, int, int, int, const char * = 0)
Creates a new [Fl_Menu_Button](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- const [Fl_Menu_Item](#) * [popup](#) ()
Act exactly as though the user clicked the button or typed the shortcut key.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.55.1 Detailed Description

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of [Fl_Menu_Item](#) objects.



Figure 30.19: menu_button

Normally any mouse button will pop up a menu and it is lined up below the button as shown in the picture. However an [FL_Menu_Button](#) may also control a pop-up menu. This is done by setting the [type\(\)](#), see below.

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the [label\(\)](#).

Typing the [shortcut\(\)](#) of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse. The '&' character in menu item names are only looked at when the menu is popped up, however.

When the user picks an item off the menu, the item's callback is done with the menu_button as the `FL_Widget*` argument. If the item does not have a callback the menu_button's callback is done instead.

30.55.2 Constructor & Destructor Documentation

30.55.2.1 `FL_Menu_Button::FL_Menu_Button (int X, int Y, int W, int H, const char *l = 0)`

Creates a new [FL_Menu_Button](#) widget using the given position, size, and label string.

The default boxtype is `FL_UP_BOX`.

The constructor sets [menu\(\)](#) to `NULL`. See [FL_Menu_](#) for the methods to set or change the menu.

30.55.3 Member Function Documentation

30.55.3.1 `void FL_Menu_Button::draw ()` [`protected`, `virtual`]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an

embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual `draw()` method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.55.3.2 `int Fl_Menu_Button::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited `handle()` method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0* if the event was not used or understood
- 1* if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

30.55.3.3 `const Fl_Menu_Item * Fl_Menu_Button::popup ()`

Act exactly as though the user clicked the button or typed the shortcut key.

The menu appears, it waits for the user to pick an item, and if they pick one it sets `value()` and does the callback or sets `changed()` as described above. The menu item is returned or NULL if the user dismisses the menu.

The documentation for this class was generated from the following files:

- `Fl_Menu_Button.H`
- `Fl_Menu_Button.cxx`

30.56 FL_Menu_Item Struct Reference

The [FL_Menu_Item](#) structure defines a single menu item that is used by the [FL_Menu_](#) class.

```
#include <FL_Menu_Item.H>
```

Public Member Functions

- void [activate](#) ()
Allows a menu item to be picked.
- int [active](#) () const
Gets whether or not the item can be picked.
- int [activevisible](#) () const
Returns non 0 if FL_INACTIVE and FL_INVISIBLE are cleared, 0 otherwise.
- int [add](#) (const char *a, const char *b, FL_Callback *c, void *d=0, int e=0)
See int [add\(const char, int shortcut, FL_Callback*, void*, int\)](#).*
- int [add](#) (const char *, int shortcut, FL_Callback *, void *=0, int=0)
Adds an item.
- void [argument](#) (long v)
For convenience you can also define the callback as taking a long argument.
- long [argument](#) () const
For convenience you can also define the callback as taking a long argument.
- void [callback](#) (FL_Callback1 *c, long p=0)
See FL_Callback_p FL_MenuItem::callback() const.
- void [callback](#) (FL_Callback0 *c)
See FL_Callback_p FL_MenuItem::callback() const.
- void [callback](#) (FL_Callback *c)
See FL_Callback_p FL_MenuItem::callback() const.
- void [callback](#) (FL_Callback *c, void *p)
See FL_Callback_p FL_MenuItem::callback() const.
- FL_Callback_p [callback](#) () const
Each item has space for a callback function and an argument for that function.
- void [check](#) ()
back compatibility only
- int [checkbox](#) () const
Returns true if a checkbox will be drawn next to this item.

- int `checked` () const
back compatibility only
- void `clear` ()
Turns the check or radio item "off" for the menu item.
- void `deactivate` ()
Prevents a menu item from being picked.
- void `do_callback` (Fl_Widget *o, long arg) const
Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).
- void `do_callback` (Fl_Widget *o, void *arg) const
Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).
- void `do_callback` (Fl_Widget *o) const
Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).
- void `draw` (int x, int y, int w, int h, const Fl_Menu_ *, int t=0) const
Draws the menu item in bounding box x,y,w,h, optionally selects the item.
- const Fl_Menu_Item * `find_shortcut` (int *ip=0) const
Search only the top level menu for a shortcut.
- Fl_Menu_Item * `first` ()
Returns the first menu item, same as `next(0)`.
- const Fl_Menu_Item * `first` () const
Returns the first menu item, same as `next(0)`.
- void `hide` ()
Hides an item in the menu.
- void `image` (Fl_Image &a)
compatibility api for FLUID, same as `a.label(this)`
- void `image` (Fl_Image *a)
compatibility api for FLUID, same as `a->label(this)`
- void `label` (Fl_Labeltype a, const char *b)
See const char `Fl_Menu_Item::label()` const.*
- void `label` (const char *a)
See const char `Fl_Menu_Item::label()` const.*
- const char * `label` () const
Returns the title of the item.

- void [labelcolor](#) (unsigned a)
See [Fl_Color](#) [Fl_Menu_Item::labelcolor\(\)](#) const.
- [Fl_Color](#) [labelcolor](#) () const
This color is passed to the [labeltype](#) routine, and is typically the color of the label text.
- void [labelfont](#) ([Fl_Font](#) a)
Fonts are identified by small 8-bit indexes into a table.
- [Fl_Font](#) [labelfont](#) () const
Fonts are identified by small 8-bit indexes into a table.
- void [labelsize](#) ([Fl_Fonsize](#) a)
Sets the label font pixel size/height.
- [Fl_Fonsize](#) [labelsize](#) () const
Gets the label font pixel size/height.
- void [labeltype](#) ([Fl_Labeltype](#) a)
A labeltype identifies a routine that draws the label of the widget.
- [Fl_Labeltype](#) [labeltype](#) () const
A labeltype identifies a routine that draws the label of the widget.
- int [measure](#) (int *h, const [Fl_Menu_](#) *) const
Measures width of label, including effect of & characters.
- [Fl_Menu_Item](#) * [next](#) (int i=1)
Advances a pointer by n items through a menu array, skipping the contents of submenus and invisible items.
- const [Fl_Menu_Item](#) * [next](#) (int=1) const
Advance a pointer by n items through a menu array, skipping the contents of submenus and invisible items.
- const [Fl_Menu_Item](#) * [popup](#) (int X, int Y, const char *title=0, const [Fl_Menu_Item](#) *picked=0, const [Fl_Menu_](#) *=0) const
This method is called by widgets that want to display menus.
- const [Fl_Menu_Item](#) * [pulldown](#) (int X, int Y, int W, int H, const [Fl_Menu_Item](#) *picked=0, const [Fl_Menu_](#) *=0, const [Fl_Menu_Item](#) *title=0, int menubar=0) const
Pulldown() is similar to [popup\(\)](#), but a rectangle is provided to position the menu.
- int [radio](#) () const
Returns true if this item is a radio item.
- void [set](#) ()
Turns the check or radio item "on" for the menu item.
- void [setonly](#) ()
Turns the radio item "on" for the menu item and turns off adjacent radio items set.

- void [shortcut](#) (int s)
Sets exactly what key combination will trigger the menu item.
- int [shortcut](#) () const
Gets what key combination shortcut will trigger the menu item.
- void [show](#) ()
Makes an item visible in the menu.
- int [size](#) () const
Size of the menu starting from this menu item.
- int [submenu](#) () const
Returns true if either FL_SUBMENU or FL_SUBMENU_POINTER is on in the flags.
- const [FL_Menu_Item](#) * [test_shortcut](#) () const
This is designed to be called by a widgets handle() method in response to a FL_SHORTCUT event.
- void [uncheck](#) ()
back compatibility only
- void [user_data](#) (void *v)
Get or set the user_data argument that is sent to the callback function.
- void * [user_data](#) () const
Get or set the user_data argument that is sent to the callback function.
- int [value](#) () const
Returns the current value of the check or radio item.
- int [visible](#) () const
Gets the visibility of an item.

Public Attributes

- [FL_Callback](#) * [callback_](#)
menu item callback
- int [flags](#)
menu item flags like FL_MENU_TOGGLE, FL_MENU_RADIO
- unsigned [labelcolor_](#)
menu item text color
- [FL_Font](#) [labelfont_](#)
which font for this menu item text

- [Fl_Fontsize labelsizes_](#)
size of menu item text
- [uchar labeltype_](#)
how the menu item text looks like
- [int shortcut_](#)
menu item shortcut
- [const char * text](#)
menu item text, returned by [label\(\)](#)
- [void * user_data_](#)
menu item user_data for 3rd party apps

30.56.1 Detailed Description

The [Fl_Menu_Item](#) structure defines a single menu item that is used by the [Fl_Menu_](#) class.

```
struct Fl_Menu_Item {
    const char*      text; // label()
    ulong            shortcut_;
    Fl_Callback*     callback_;
    void*            user_data_;
    int              flags;
    uchar            labeltype_;
    uchar            labelfont_;
    uchar            labelsizes_;
    uchar            labelcolor_;
};

enum { // values for flags:
    FL_MENU_INACTIVE      = 1,
    FL_MENU_TOGGLE        = 2,
    FL_MENU_VALUE         = 4,
    FL_MENU_RADIO         = 8,
    FL_MENU_INVISIBLE     = 0x10,
    FL_SUBMENU_POINTER    = 0x20,
    FL_SUBMENU            = 0x40,
    FL_MENU_DIVIDER       = 0x80,
    FL_MENU_HORIZONTAL    = 0x100
};
```

Typically menu items are statically defined; for example:

```
Fl_Menu_Item popup[] = {
    {"&alpha",    FL_ALT+'a', the_cb, (void*)1},
    {"&beta",     FL_ALT+'b', the_cb, (void*)2},
    {"&gamma",    FL_ALT+'c', the_cb, (void*)3, FL_MENU_DIVIDER},
    {"&strange",  0,    strange_cb},
    {"&charm",    0,    charm_cb},
    {"&truth",    0,    truth_cb},
    {"&beauty",   0,    beauty_cb},
    {"sub&menu",  0,    0, 0, FL_SUBMENU},
    {"one"},
    {"two"},
    {"three"},
};
```

```
{0},
{"inactive", FL_ALT+'i', 0, 0, FL_MENU_INACTIVE|FL_MENU_DIVIDER},
{"invisible", FL_ALT+'i', 0, 0, FL_MENU_INVISIBLE},
{"check",    FL_ALT+'i', 0, 0, FL_MENU_TOGGLE|FL_MENU_VALUE},
{"box",      FL_ALT+'i', 0, 0, FL_MENU_TOGGLE},
{0}};
```

produces:

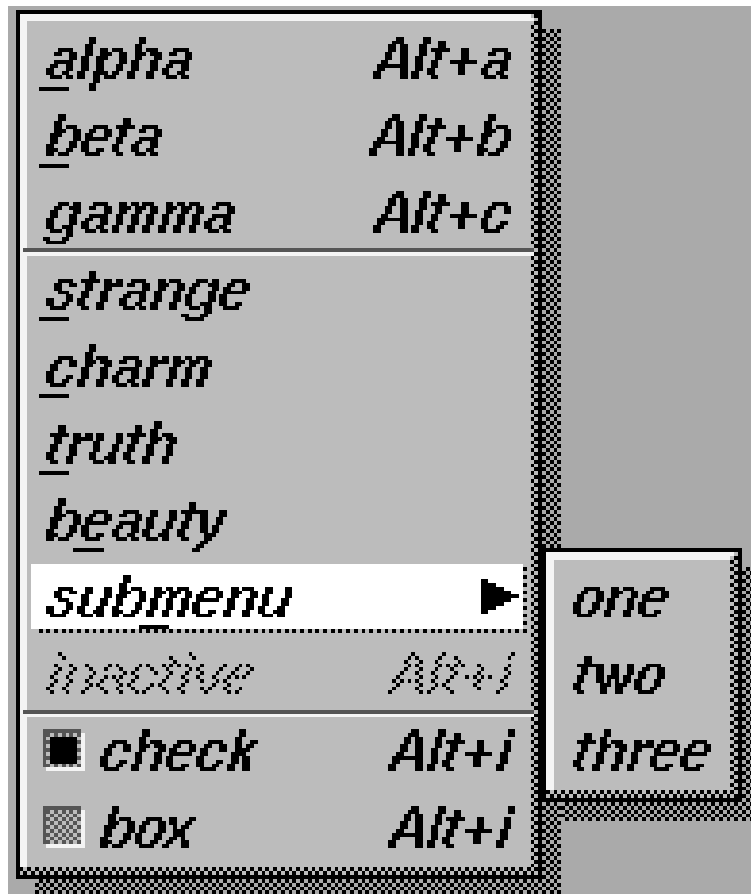


Figure 30.20: menu

A submenu title is identified by the bit `FL_SUBMENU` in the flags field, and ends with a `label()` that is `NULL`. You can nest menus to any depth. A pointer to the first item in the submenu can be treated as an `Fl_Menu` array itself. It is also possible to make separate submenu arrays with `FL_SUBMENU_POINTER` flags.

You should use the method functions to access structure members and not access them directly to avoid compatibility problems with future releases of FLTK.

30.56.2 Member Function Documentation

30.56.2.1 `void Fl_Menu_Item::activate ()` [inline]

Allows a menu item to be picked.

30.56.2.2 `int Fl_Menu_Item::active () const` `[inline]`

Gets whether or not the item can be picked.

30.56.2.3 `int Fl_Menu_Item::activevisible () const` `[inline]`

Returns non 0 if FL_INACTIVE and FL_INVISIBLE are cleared, 0 otherwise.

30.56.2.4 `int Fl_Menu_Item::add (const char * mytext, int sc, Fl_Callback * cb, void * data = 0, int myflags = 0)`

Adds an item.

The text is split at '/' characters to automatically produce submenus (actually a totally unnecessary feature as you can now add submenu titles directly by setting SUBMENU in the flags):

30.56.2.5 `void Fl_Menu_Item::argument (long v)` `[inline]`

For convenience you can also define the callback as taking a long argument.

This is implemented by casting this to a Fl_Callback and casting the long to a void* and may not be portable to some machines.

30.56.2.6 `long Fl_Menu_Item::argument () const` `[inline]`

For convenience you can also define the callback as taking a long argument.

This is implemented by casting this to a Fl_Callback and casting the long to a void* and may not be portable to some machines.

30.56.2.7 `Fl_Callback_p Fl_Menu_Item::callback () const` `[inline]`

Each item has space for a callback function and an argument for that function.

Due to back compatibility, the [Fl_Menu_Item](#) itself is not passed to the callback, instead you have to get it by calling ((Fl_Menu_*)w)->mvalue() where w is the widget argument.

30.56.2.8 `void Fl_Menu_Item::check ()` `[inline]`

back compatibility only

Deprecated**30.56.2.9** `int Fl_Menu_Item::checkbox () const` `[inline]`

Returns true if a checkbox will be drawn next to this item.

This is true if FL_MENU_TOGGLE or FL_MENU_RADIO is set in the flags.

30.56.2.10 `int Fl_Menu_Item::checked () const` `[inline]`

back compatibility only

Deprecated

30.56.2.11 `void Fl_Menu_Item::clear ()` `[inline]`

Turns the check or radio item "off" for the menu item.

30.56.2.12 `void Fl_Menu_Item::deactivate ()` `[inline]`

Prevents a menu item from being picked.

Note that this will also cause the menu item to appear grayed-out.

30.56.2.13 `void Fl_Menu_Item::do_callback (Fl_Widget * o, long arg) const` `[inline]`

Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).

You must first check that `callback()` is non-zero before calling this.

30.56.2.14 `void Fl_Menu_Item::do_callback (Fl_Widget * o, void * arg) const` `[inline]`

Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).

You must first check that `callback()` is non-zero before calling this.

30.56.2.15 `void Fl_Menu_Item::do_callback (Fl_Widget * o) const` `[inline]`

Calls the `Fl_Menu_Item` item's callback, and provides the `Fl_Widget` argument (and optionally overrides the `user_data()` argument).

You must first check that `callback()` is non-zero before calling this.

30.56.2.16 `void Fl_Menu_Item::draw (int x, int y, int w, int h, const Fl_Menu_ * m, int selected = 0) const`

Draws the menu item in bounding box x,y,w,h, optionally selects the item.

30.56.2.17 `const Fl_Menu_Item * Fl_Menu_Item::find_shortcut (int * ip = 0) const`

Search only the top level menu for a shortcut.

Either &x in the label or the shortcut fields are used.

30.56.2.18 `Fl_Menu_Item* Fl_Menu_Item::first ()` [inline]

Returns the first menu item, same as next(0).

30.56.2.19 `const Fl_Menu_Item* Fl_Menu_Item::first () const` [inline]

Returns the first menu item, same as next(0).

30.56.2.20 `void Fl_Menu_Item::hide ()` [inline]

Hides an item in the menu.

30.56.2.21 `const char* Fl_Menu_Item::label () const` [inline]

Returns the title of the item.

A NULL here indicates the end of the menu (or of a submenu). A '&' in the item will print an underscore under the next letter, and if the menu is popped up that letter will be a "shortcut" to pick that item. To get a real '&' put two in a row.

30.56.2.22 `Fl_Color Fl_Menu_Item::labelcolor () const` [inline]

This color is passed to the labeltype routine, and is typically the color of the label text.

This defaults to FL_BLACK. If this color is not black fltk will *not* use overlay bitplanes to draw the menu - this is so that images put in the menu draw correctly.

30.56.2.23 `void Fl_Menu_Item::labelfont (Fl_Font a)` [inline]

Fonts are identified by small 8-bit indexes into a table.

See the enumeration list for predefined fonts. The default value is a Helvetica font. The function [Fl::set_font\(\)](#) can define new fonts.

30.56.2.24 `Fl_Font Fl_Menu_Item::labelfont () const` [inline]

Fonts are identified by small 8-bit indexes into a table.

See the enumeration list for predefined fonts. The default value is a Helvetica font. The function [Fl::set_font\(\)](#) can define new fonts.

30.56.2.25 `void Fl_Menu_Item::labelsize (Fl_Fontsize a)` [inline]

Sets the label font pixel size/height.

30.56.2.26 `Fl_Fontsize Fl_Menu_Item::labelsize () const` [inline]

Gets the label font pixel size/height.

30.56.2.27 void Fl_Menu_Item::labeltype (Fl_Labeltype *a*) [inline]

A labeltype identifies a routine that draws the label of the widget.

This can be used for special effects such as emboss, or to use the [label\(\)](#) pointer as another form of data such as a bitmap. The value FL_NORMAL_LABEL prints the label as text.

30.56.2.28 Fl_Labeltype Fl_Menu_Item::labeltype () const [inline]

A labeltype identifies a routine that draws the label of the widget.

This can be used for special effects such as emboss, or to use the [label\(\)](#) pointer as another form of data such as a bitmap. The value FL_NORMAL_LABEL prints the label as text.

30.56.2.29 int Fl_Menu_Item::measure (int * *hp*, const Fl_Menu_ * *m*) const

Measures width of label, including effect of & characters.

Optionally, can get height if *hp* is not NULL.

30.56.2.30 Fl_Menu_Item* Fl_Menu_Item::next (int *i* = 1) [inline]

Advances a pointer by *n* items through a menu array, skipping the contents of submenus and invisible items.

There are two calls so that you can advance through const and non-const data.

30.56.2.31 const Fl_Menu_Item * Fl_Menu_Item::next (int *n* = 1) const

Advance a pointer by *n* items through a menu array, skipping the contents of submenus and invisible items.

There are two calls so that you can advance through const and non-const data.

30.56.2.32 const Fl_Menu_Item * Fl_Menu_Item::popup (int *X*, int *Y*, const char * *title* = 0, const Fl_Menu_Item * *picked* = 0, const Fl_Menu_ * *but* = 0) const

This method is called by widgets that want to display menus.

The menu stays up until the user picks an item or dismisses it. The selected item (or NULL if none) is returned. *This does not do the callbacks or change the state of check or radio items.*

X,Y is the position of the mouse cursor, relative to the window that got the most recent event (usually you can pass [Fl::event_x\(\)](#) and [Fl::event_y\(\)](#) unchanged here).

title is a character string title for the menu. If non-zero a small box appears above the menu with the title in it.

The menu is positioned so the cursor is centered over the item picked. This will work even if picked is in a submenu. If picked is zero or not in the menu item table the menu is positioned with the cursor in the top-left corner.

button is a pointer to an [Fl_Menu_](#) from which the color and boxtypes for the menu are pulled. If NULL then defaults are used.

30.56.2.33 `const Fl_Menu_Item * Fl_Menu_Item::pulldown (int X, int Y, int W, int H, const Fl_Menu_Item * initial_item = 0, const Fl_Menu_ * pbutton = 0, const Fl_Menu_Item * t = 0, int menubar = 0) const`

Pulldown() is similar to [popup\(\)](#), but a rectangle is provided to position the menu.

The menu is made at least W wide, and the picked item is centered over the rectangle (like [Fl_Choice](#) uses). If picked is zero or not found, the menu is aligned just below the rectangle (like a pulldown menu).

The title and menubar arguments are used internally by the [Fl_Menu_Bar](#) widget.

30.56.2.34 `int Fl_Menu_Item::radio () const` `[inline]`

Returns true if this item is a radio item.

When a radio button is selected all "adjacent" radio buttons are turned off. A set of radio items is delimited by an item that has [radio\(\)](#) false, or by an item with FL_MENU_DIVIDER turned on.

30.56.2.35 `void Fl_Menu_Item::set ()` `[inline]`

Turns the check or radio item "on" for the menu item.

Note that this does not turn off any adjacent radio items like `set_only()` does.

30.56.2.36 `void Fl_Menu_Item::setonly ()`

Turns the radio item "on" for the menu item and turns off adjacent radio items set.

30.56.2.37 `void Fl_Menu_Item::shortcut (int s)` `[inline]`

Sets exactly what key combination will trigger the menu item.

The value is a logical 'or' of a key and a set of shift flags, for instance FL_ALT+'a' or FL_ALT+FL_F+10 or just 'a'. A value of zero disables the shortcut.

The key can be any value returned by [Fl::event_key\(\)](#), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by [Fl::event_state\(\)](#). If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

30.56.2.38 `int Fl_Menu_Item::shortcut () const` `[inline]`

Gets what key combination shortcut will trigger the menu item.

30.56.2.39 `void Fl_Menu_Item::show ()` `[inline]`

Makes an item visible in the menu.

30.56.2.40 `int Fl_Menu_Item::submenu () const` `[inline]`

Returns true if either FL_SUBMENU or FL_SUBMENU_POINTER is on in the flags.

FL_SUBMENU indicates an embedded submenu that goes from the next item through the next one with a NULL `label()`. FL_SUBMENU_POINTER indicates that `user_data()` is a pointer to another menu array.

30.56.2.41 `const Fl_Menu_Item * Fl_Menu_Item::test_shortcut () const`

This is designed to be called by a widgets `handle()` method in response to a FL_SHORTCUT event.

If the current event matches one of the items shortcut, that item is returned. If the keystroke does not match any shortcuts then NULL is returned. This only matches the `shortcut()` fields, not the letters in the title preceded by ' .

30.56.2.42 `void Fl_Menu_Item::unchecked ()` `[inline]`

back compatibility only

Deprecated

30.56.2.43 `int Fl_Menu_Item::value () const` `[inline]`

Returns the current value of the check or radio item.

30.56.2.44 `int Fl_Menu_Item::visible () const` `[inline]`

Gets the visibility of an item.

The documentation for this struct was generated from the following files:

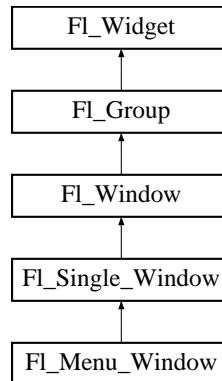
- Fl_Menu_Item.H
- Fl_Menu.cxx
- Fl_Menu_.cxx
- Fl_Menu_add.cxx

30.57 FL_Menu_Window Class Reference

The [FL_Menu_Window](#) widget is a window type used for menus.

```
#include <FL_Menu_Window.H>
```

Inheritance diagram for [FL_Menu_Window](#)::



Public Member Functions

- void [clear_overlay](#) ()
Tells FLTK to use normal drawing planes instead of overlay planes.
- void [erase](#) ()
Erases the window, does nothing if HAVE_OVERLAY is not defined config.h.
- [FL_Menu_Window](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [FL_Menu_Window](#) widget using the given position, size, and label string.
- [FL_Menu_Window](#) (int W, int H, const char *l=0)
Creates a new [FL_Menu_Window](#) widget using the given size, and label string.
- void [flush](#) ()
Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).
- void [hide](#) ()
Remove the window from the screen.
- int [overlay](#) ()
Tells if hardware overlay mode is set.
- void [set_overlay](#) ()
Tells FLTK to use hardware overlay planes if they are available.
- void [show](#) ()
Put the window on the screen.
- [~FL_Menu_Window](#) ()

Destroys the window and all of its children.

30.57.1 Detailed Description

The [FL_Menu_Window](#) widget is a window type used for menus.

By default the window is drawn in the hardware overlay planes if they are available so that the menu don't force the rest of the window to redraw.

30.57.2 Constructor & Destructor Documentation

30.57.2.1 [FL_Menu_Window::~~FL_Menu_Window \(\)](#)

Destroys the window and all of its children.

30.57.2.2 [FL_Menu_Window::FL_Menu_Window \(int *W*, int *H*, const char * *l* = 0\)](#) [inline]

Creates a new [FL_Menu_Window](#) widget using the given size, and label string.

30.57.2.3 [FL_Menu_Window::FL_Menu_Window \(int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0\)](#) [inline]

Creates a new [FL_Menu_Window](#) widget using the given position, size, and label string.

30.57.3 Member Function Documentation

30.57.3.1 [void FL_Menu_Window::clear_overlay \(\)](#) [inline]

Tells FLTK to use normal drawing planes instead of overlay planes.

This is usually necessary if your menu contains multi-color pixmaps.

30.57.3.2 [void FL_Menu_Window::flush \(\)](#) [virtual]

Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).

Reimplemented from [FL_Single_Window](#).

30.57.3.3 [void FL_Menu_Window::hide \(\)](#) [virtual]

Remove the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from [FL_Window](#).

30.57.3.4 [void FL_Menu_Window::set_overlay \(\)](#) [inline]

Tells FLTK to use hardware overlay planes if they are available.

30.57.3.5 void Fl_Menu_Window::show () [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [Fl_Single_Window](#).

The documentation for this class was generated from the following files:

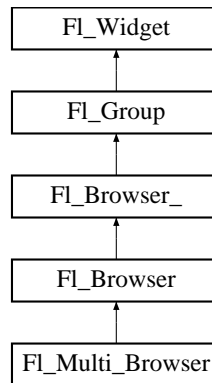
- Fl_Menu_Window.H
- Fl_Menu_Window.cxx

30.58 FL_Multi_Browser Class Reference

The [FL_Multi_Browser](#) class is a subclass of [FL_Browser](#) which lets the user select any set of the lines.

```
#include <Fl_Multi_Browser.H>
```

Inheritance diagram for FL_Multi_Browser::



Public Member Functions

- [FL_Multi_Browser](#) (int X, int Y, int W, int H, const char *L=0)

Creates a new [FL_Multi_Browser](#) widget using the given position, size, and label string.

30.58.1 Detailed Description

The [FL_Multi_Browser](#) class is a subclass of [FL_Browser](#) which lets the user select any set of the lines.

The user interface is Macintosh style: clicking an item turns off all the others and selects that one, dragging selects all the items the mouse moves over, and shift + click toggles the items. This is different then how forms did it. Normally the callback is done when the user releases the mouse, but you can change this with [when\(\)](#).

See [FL_Browser](#) for methods to add and remove lines from the browser.

30.58.2 Constructor & Destructor Documentation

30.58.2.1 [FL_Multi_Browser::FL_Multi_Browser](#) (int X, int Y, int W, int H, const char * L = 0)
[inline]

Creates a new [FL_Multi_Browser](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX. The constructor specializes [FL_Browser\(\)](#) by setting the type to FL_MULTI_BROWSER. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following file:

- FL_Multi_Browser.H

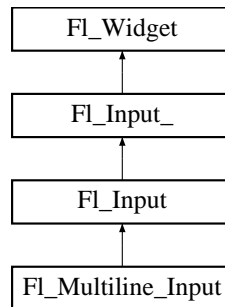
30.59 FL_Multiline_Input Class Reference

This input field displays '

' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys.

```
#include <Fl_Multiline_Input.H>
```

Inheritance diagram for FL_Multiline_Input::



Public Member Functions

- [FL_Multiline_Input](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [FL_Multiline_Input](#) widget using the given position, size, and label string.

30.59.1 Detailed Description

This input field displays '

' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys.

This is for editing multiline text.

This is far from the nirvana of text editors, and is probably only good for small bits of text, 10 lines at most. I think FLTK can be used to write a powerful text editor, but it is not going to be a built-in feature. Powerful text editors in a toolkit are a big source of bloat.

30.59.2 Constructor & Destructor Documentation

30.59.2.1 FL_Multiline_Input::FL_Multiline_Input (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [FL_Multiline_Input](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following file:

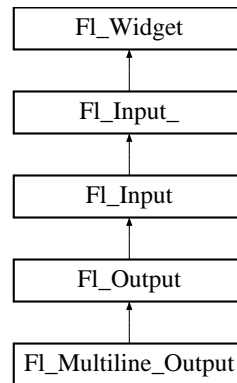
- FL_Multiline_Input.H

30.60 Fl_Multiline_Output Class Reference

This widget is a subclass of [Fl_Output](#) that displays multiple lines of text.

```
#include <Fl_Multiline_Output.H>
```

Inheritance diagram for Fl_Multiline_Output::



Public Member Functions

- [Fl_Multiline_Output](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Multiline_Output](#) widget using the given position, size, and label string.

30.60.1 Detailed Description

This widget is a subclass of [Fl_Output](#) that displays multiple lines of text.

It also displays tab characters as whitespace to the next column.

30.60.2 Constructor & Destructor Documentation

30.60.2.1 [Fl_Multiline_Output::Fl_Multiline_Output](#) (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [Fl_Multiline_Output](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following file:

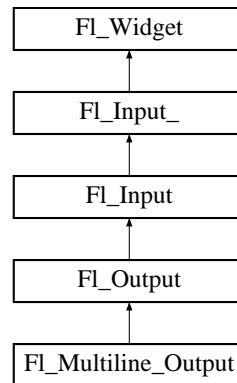
- Fl_Multiline_Output.H

30.61 FL_Output Class Reference

This widget displays a piece of text.

```
#include <FL_Output.H>
```

Inheritance diagram for FL_Output::



Public Member Functions

- [FL_Output](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [FL_Output](#) widget using the given position, size, and label string.

30.61.1 Detailed Description

This widget displays a piece of text.

When you set the [value\(\)](#) , [FL_Output](#) does a strcpy() to it's own storage, which is useful for program-generated values. The user may select portions of the text using the mouse and paste the contents into other fields or programs.

Figure 30.21: Fl_Output

There is a single subclass, [Fl_Multiline_Output](#), which allows you to display multiple lines of text.

The text may contain any characters except `\0`, and will correctly display anything, using `^X` notation for unprintable control characters and `\nnn` notation for unprintable characters with the high bit set. It assumes the font can draw any characters in the ISO-Latin1 character set.

30.61.2 Constructor & Destructor Documentation

30.61.2.1 `Fl_Output::Fl_Output (int X, int Y, int W, int H, const char *l = 0)` `[inline]`

Creates a new [Fl_Output](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following file:

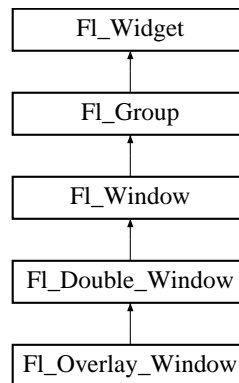
- `Fl_Output.H`

30.62 FL_Overlay_Window Class Reference

This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image.

```
#include <Fl_Overlay_Window.H>
```

Inheritance diagram for FL_Overlay_Window::



Public Member Functions

- `int can_do_overlay ()`
- `FL_Overlay_Window (int X, int Y, int W, int H, const char *l=0)`
*See `FL_Overlay_Window::FL_Overlay_Window(int W, int H, const char *l=0)`.*
- `FL_Overlay_Window (int W, int H, const char *l=0)`
Creates a new `FL_Overlay_Window` widget using the given position, size, and label (title) string.
- `void flush ()`
Forces the window to be redrawn.
- `void hide ()`
Remove the window from the screen.
- `void redraw_overlay ()`
Call this to indicate that the overlay data has changed and needs to be redrawn.
- `void resize (int, int, int, int)`
Changes the size and position of the window.
- `void show (int a, char **b)`
See virtual void `FL_Window::show()`.
- `void show ()`
Put the window on the screen.
- `~FL_Overlay_Window ()`
Destroys the window and all child widgets.

Friends

- class [_FL_Overlay](#)

30.62.1 Detailed Description

This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image.

The overlay is designed to be a rapidly-changing but simple graphic such as a mouse selection box. [FL_Overlay_Window](#) uses the overlay planes provided by your graphics hardware if they are available.

If no hardware support is found the overlay is simulated by drawing directly into the on-screen copy of the double-buffered window, and "erased" by copying the backbuffer over it again. This means the overlay will blink if you change the image in the window.

30.62.2 Constructor & Destructor Documentation

30.62.2.1 [FL_Overlay_Window::FL_Overlay_Window](#) (int *W*, int *H*, const char * *l* = 0)
[inline]

Creates a new [FL_Overlay_Window](#) widget using the given position, size, and label (title) string.

If the positions (x,y) are not given, then the window manager will choose them.

30.62.3 Member Function Documentation

30.62.3.1 void [FL_Overlay_Window::hide](#) () [virtual]

Remove the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from [FL_Double_Window](#).

30.62.3.2 void [FL_Overlay_Window::redraw_overlay](#) ()

Call this to indicate that the overlay data has changed and needs to be redrawn.

The overlay will be clear until the first time this is called, so if you want an initial display you must call this after calling [show\(\)](#).

30.62.3.3 void [FL_Overlay_Window::resize](#) (int, int, int, int) [virtual]

Changes the size and position of the window.

If [shown\(\)](#) is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If [shown\(\)](#) is false, the size and position are used when [show\(\)](#) is called. See [FL_Group](#) for the effect of resizing on the child widgets.

You can also call the [FL_Widget](#) methods [size\(x,y\)](#) and [position\(w,h\)](#), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from [Fl_Double_Window](#).

30.62.3.4 void Fl_Overlay_Window::show () [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [Fl_Double_Window](#).

The documentation for this class was generated from the following files:

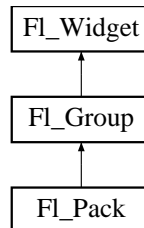
- [Fl_Overlay_Window.H](#)
- [Fl_Overlay_Window.cxx](#)

30.63 Fl_Pack Class Reference

This widget was designed to add the functionality of compressing and aligning widgets.

```
#include <Fl_Pack.H>
```

Inheritance diagram for Fl_Pack::



Public Types

- enum { **VERTICAL** = 0, **HORIZONTAL** = 1 }

Public Member Functions

- [Fl_Pack](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [Fl_Pack](#) widget using the given position, size, and label string.
- [uchar horizontal](#) () const
Same as [Fl_Group::type\(\)](#).
- void [spacing](#) (int i)
Sets the number of extra pixels of blank space that are added between the children.
- int [spacing](#) () const
Gets the number of extra pixels of blank space that are added between the children.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.63.1 Detailed Description

This widget was designed to add the functionality of compressing and aligning widgets.

If [type\(\)](#) is `Fl_Pack::HORIZONTAL` all the children are resized to the height of the [Fl_Pack](#), and are moved next to each other horizontally. If [type\(\)](#) is not `Fl_Pack::HORIZONTAL` then the children are resized to the width and are stacked below each other. Then the [Fl_Pack](#) resizes itself to surround the child widgets.

This widget is needed for the [Fl_Tabs](#). In addition you may want to put the [Fl_Pack](#) inside an [Fl_Scroll](#).

The resizable for [Fl_Pack](#) is set to NULL by default.

See also: [Fl_Group::resizable\(\)](#)

30.63.2 Constructor & Destructor Documentation

30.63.2.1 [Fl_Pack::Fl_Pack](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [Fl_Pack](#) widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the [Fl_Pack](#) and all of its children can be automatic (local) variables, but you must declare the [Fl_Pack](#) *first*, so that it is destroyed last.

30.63.3 Member Function Documentation

30.63.3.1 void [Fl_Pack::draw](#) () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Group](#).

The documentation for this class was generated from the following files:

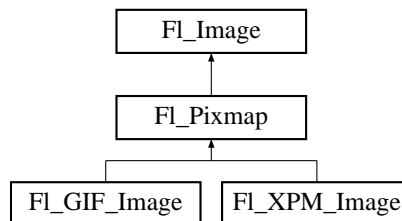
- [Fl_Pack.H](#)
- [Fl_Pack.cxx](#)

30.64 Fl_Pixmap Class Reference

The [Fl_Pixmap](#) class supports caching and drawing of colormap (pixmap) images, including transparency.

```
#include <Fl_Pixmap.H>
```

Inheritance diagram for Fl_Pixmap::



Public Member Functions

- virtual void [color_average](#) ([Fl_Color](#) c, float i)
The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.
- [Fl_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [Fl_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- virtual void [desaturate](#) ()
The [desaturate\(\)](#) method converts an image to grayscale.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx=0, int cy=0)
The [draw\(\)](#) methods draw the image.
- [Fl_Pixmap](#) (const [uchar](#) *const *D)
The constructors create a new pixmap from the specified XPM data.
- [Fl_Pixmap](#) (const char *const *D)
The constructors create a new pixmap from the specified XPM data.
- [Fl_Pixmap](#) ([uchar](#) *const *D)
The constructors create a new pixmap from the specified XPM data.
- [Fl_Pixmap](#) (char *const *D)
The constructors create a new pixmap from the specified XPM data.
- virtual void [label](#) ([Fl_Menu_Item](#) *m)
The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.

- virtual void `label (FL_Widget *w)`

The `label()` methods are an obsolete way to set the image attribute of a widget or menu item.

- virtual void `uncache ()`

If the image has been cached for display, delete the cache data.

- virtual `~FL_Pixmap ()`

The destructor free all memory and server resources that are used by the pixmap.

Public Attributes

- int `alloc_data`
- unsigned `id`
- unsigned `mask`

Protected Member Functions

- void `measure ()`

30.64.1 Detailed Description

The `FL_Pixmap` class supports caching and drawing of colormap (pixmap) images, including transparency.

30.64.2 Constructor & Destructor Documentation

30.64.2.1 `FL_Pixmap::FL_Pixmap (char *const *D)` [inline, explicit]

The constructors create a new pixmap from the specified XPM data.

30.64.2.2 `FL_Pixmap::FL_Pixmap (uchar *const *D)` [inline, explicit]

The constructors create a new pixmap from the specified XPM data.

30.64.2.3 `FL_Pixmap::FL_Pixmap (const char *const *D)` [inline, explicit]

The constructors create a new pixmap from the specified XPM data.

30.64.2.4 `FL_Pixmap::FL_Pixmap (const uchar *const *D)` [inline, explicit]

The constructors create a new pixmap from the specified XPM data.

30.64.3 Member Function Documentation

30.64.3.1 void FL_Pixmap::color_average (FL_Color c, float i) [virtual]

The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.

The i argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color. *The original image data is not altered by this method.*

Reimplemented from [FL_Image](#).

30.64.3.2 FL_Image* FL_Pixmap::copy () [inline]

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [FL_Shared_Image](#), released) when you are done with it.

Reimplemented from [FL_Image](#).

30.64.3.3 FL_Image * FL_Pixmap::copy (int W, int H) [virtual]

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [FL_Shared_Image](#), released) when you are done with it.

Reimplemented from [FL_Image](#).

30.64.3.4 void FL_Pixmap::desaturate () [virtual]

The [desaturate\(\)](#) method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved. *This method does not alter the original image data.*

Reimplemented from [FL_Image](#).

30.64.3.5 void FL_Pixmap::draw (int X, int Y) [inline]

The [draw\(\)](#) methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented from [FL_Image](#).

30.64.3.6 void FL_Pixmap::draw (int X, int Y, int W, int H, int cx = 0, int cy = 0) [virtual]

The [draw\(\)](#) methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the cx and cy arguments.

Reimplemented from [FL_Image](#).

30.64.3.7 void FL_Pixmap::label (FL_Menu_Item * *m*) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item. Use the [image\(\)](#) or [deimage\(\)](#) methods of the [FL_Widget](#) and [FL_Menu_Item](#) classes instead. Reimplemented from [FL_Image](#).

30.64.3.8 void FL_Pixmap::label (FL_Widget * *widget*) [virtual]

The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item. Use the [image\(\)](#) or [deimage\(\)](#) methods of the [FL_Widget](#) and [FL_Menu_Item](#) classes instead. Reimplemented from [FL_Image](#).

30.64.3.9 void FL_Pixmap::uncache () [virtual]

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from [FL_Image](#).

The documentation for this class was generated from the following files:

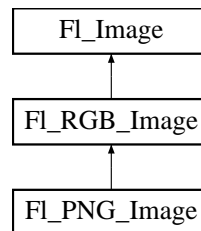
- [FL_Pixmap.H](#)
- [FL_Pixmap.cxx](#)

30.65 Fl_PNG_Image Class Reference

The [Fl_PNG_Image](#) class supports loading, caching, and drawing of Portable Network Graphics (PNG) image files.

```
#include <Fl_PNG_Image.H>
```

Inheritance diagram for Fl_PNG_Image::



Public Member Functions

- [Fl_PNG_Image](#) (const char *filename)

The constructor loads the named PNG image from the given png filename.

30.65.1 Detailed Description

The [Fl_PNG_Image](#) class supports loading, caching, and drawing of Portable Network Graphics (PNG) image files.

The class loads colormapped and full-color images and handles color- and alpha-based transparency.

30.65.2 Constructor & Destructor Documentation

30.65.2.1 Fl_PNG_Image::Fl_PNG_Image (const char * *png*)

The constructor loads the named PNG image from the given png filename.

The destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

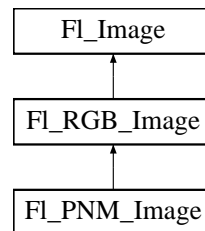
- Fl_PNG_Image.H
- Fl_PNG_Image.cxx

30.66 Fl_PNM_Image Class Reference

The [Fl_PNM_Image](#) class supports loading, caching, and drawing of Portable Anymap (PNM, PBM, PGM, PPM) image files.

```
#include <Fl_PNM_Image.H>
```

Inheritance diagram for Fl_PNM_Image::



Public Member Functions

- [Fl_PNM_Image](#) (const char *filename)

The constructor loads the named PNM image.

30.66.1 Detailed Description

The [Fl_PNM_Image](#) class supports loading, caching, and drawing of Portable Anymap (PNM, PBM, PGM, PPM) image files.

The class loads bitmap, grayscale, and full-color images in both ASCII and binary formats.

30.66.2 Constructor & Destructor Documentation

30.66.2.1 Fl_PNM_Image::Fl_PNM_Image (const char * name)

The constructor loads the named PNM image.

The inherited destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

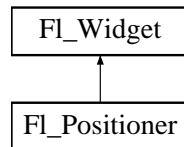
- Fl_PNM_Image.H
- Fl_PNM_Image.cxx

30.67 Fl_Positioner Class Reference

This class is provided for Forms compatibility.

```
#include <Fl_Positioner.H>
```

Inheritance diagram for Fl_Positioner::



Public Member Functions

- [Fl_Positioner](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [Fl_Positioner](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- int [value](#) (double, double)
Returns the current position in x and y.
- void [xbounds](#) (double, double)
Sets the X axis bounds.
- void [xmaximum](#) (double a)
Same as [xbounds\(xminimum\(\), a\)](#).
- double [xmaximum](#) () const
Gets the X axis maximum.
- void [xminimum](#) (double a)
Same as [xbounds\(a, xmaximum\(\)\)](#).
- double [xminimum](#) () const
Gets the X axis minimum.
- void [xstep](#) (double a)
Sets the stepping value for the X axis.
- int [xvalue](#) (double)
Sets the X axis coordinate.
- double [xvalue](#) () const
Gets the X axis coordinate.
- void [ybounds](#) (double, double)

Sets the Y axis bounds.

- void **ymaximum** (double a)
Same as ybounds(yminimum(), a).
- double **ymaximum** () const
Gets the Y axis maximum.
- void **yminimum** (double a)
Same as ybounds(a, ymaximum()).
- double **yminimum** () const
Gets the Y axis minimum.
- void **ystep** (double a)
Sets the stepping value for the Y axis.
- int **yvalue** (double)
Sets the Y axis coordinate.
- double **yvalue** () const
Gets the Y axis coordinate.

Protected Member Functions

- void **draw** ()
Draws the widget.
- void **draw** (int, int, int, int)
- int **handle** (int, int, int, int, int)

30.67.1 Detailed Description

This class is provided for Forms compatibility.

It provides 2D input. It would be useful if this could be put atop another widget so that the crosshairs are on top, but this is not implemented. The color of the crosshairs is [selection_color\(\)](#).

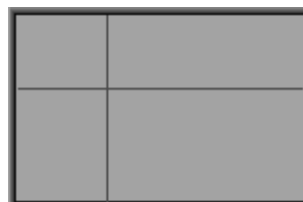


Figure 30.22: Fl_Positioner

30.67.2 Constructor & Destructor Documentation

30.67.2.1 FL_Positioner::FL_Positioner (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [FL_Positioner](#) widget using the given position, size, and label string.

Creates a new [FL_Positioner](#) widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

30.67.3 Member Function Documentation

30.67.3.1 void FL_Positioner::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to FL_Widget* and call the virtual [draw\(\)](#) method.

```
FL_Widget *s = &scroll;           // scroll is an embedded FL_Scrollbar
s->draw();                         // calls FL_Scrollbar::draw()
```

Implements [FL_Widget](#).

30.67.3.2 int FL_Positioner::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited handle() method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

30.67.3.3 int Fl_Positioner::value (double *X*, double *Y*)

Returns the current position in *x* and *y*.

30.67.3.4 void Fl_Positioner::xbounds (double *a*, double *b*)

Sets the *X* axis bounds.

30.67.3.5 void Fl_Positioner::xstep (double *a*) [inline]

Sets the stepping value for the *X* axis.

30.67.3.6 int Fl_Positioner::xvalue (double *X*)

Sets the *X* axis coordinate.

30.67.3.7 double Fl_Positioner::xvalue () const [inline]

Gets the *X* axis coordinate.

30.67.3.8 void Fl_Positioner::ybounds (double *a*, double *b*)

Sets the *Y* axis bounds.

30.67.3.9 void Fl_Positioner::ystep (double *a*) [inline]

Sets the stepping value for the *Y* axis.

30.67.3.10 int Fl_Positioner::yvalue (double *Y*)

Sets the *Y* axis coordinate.

30.67.3.11 double Fl_Positioner::yvalue () const [inline]

Gets the *Y* axis coordinate.

The documentation for this class was generated from the following files:

- Fl_Positioner.H
- Fl_Positioner.cxx

30.68 Fl_Preferences Class Reference

[Fl_Preferences](#) provides methods to store user settings between application starts.

```
#include <Fl_Preferences.H>
```

Classes

- struct [Entry](#)
An entry associates a preference name to its corresponding value.
- class [Name](#)
'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.
- class **Node**
- class **RootNode**

Public Types

- enum [Root](#) { [SYSTEM](#) = 0, [USER](#) }
Define the scope of the preferences.

Public Member Functions

- char [deleteEntry](#) (const char *entry)
Deletes a single name/value pair.
- char [deleteGroup](#) (const char *group)
Deletes a group.
- int [entries](#) ()
Returns the number of entries (name/value pairs) in a group.
- const char * [entry](#) (int index)
Returns the name of an entry.
- char [entryExists](#) (const char *key)
Returns non-zero if an entry with this name exists.
- [Fl_Preferences](#) ([Fl_Preferences](#) *, const char *group)
- [Fl_Preferences](#) ([Fl_Preferences](#) &parent, const char *group)
Generate or read a new group of entries within another group.
- [Fl_Preferences](#) (const char *path, const char *vendor, const char *application)
Use this constructor to create or read a preferences file at an arbitrary position in the file system.
- [Fl_Preferences](#) ([Root](#) root, const char *vendor, const char *application)

The constructor creates a group that manages name/value pairs and child groups.

- void [flush](#) ()
Writes all preferences to disk.
- char [get](#) (const char *entry, void *value, const void *defaultValue, int defaultSize, int maxSize)
Reads an entry from the group.
- char [get](#) (const char *entry, void *&value, const void *defaultValue, int defaultSize)
Reads an entry from the group.
- char [get](#) (const char *entry, char *value, const char *defaultValue, int maxSize)
Reads an entry from the group.
- char [get](#) (const char *entry, char *&value, const char *defaultValue)
Reads an entry from the group.
- char [get](#) (const char *entry, double &value, double defaultValue)
Reads an entry from the group.
- char [get](#) (const char *entry, float &value, float defaultValue)
Reads an entry from the group.
- char [get](#) (const char *entry, int &value, int defaultValue)
Reads an entry from the group.
- char [getUserdataPath](#) (char *path, int pathlen)
Creates a path that is related to the preferences file and that is usable for additional application data.
- const char * [group](#) (int num_group)
Returns the name of the Nth (num_group) group.
- char [groupExists](#) (const char *key)
Returns non-zero if a group with this name exists.
- int [groups](#) ()
Returns the number of groups that are contained within a group.
- char [set](#) (const char *entry, const void *value, int size)
Sets an entry (name/value pair).
- char [set](#) (const char *entry, const char *value)
Sets an entry (name/value pair).
- char [set](#) (const char *entry, double value, int precision)
Sets an entry (name/value pair).
- char [set](#) (const char *entry, double value)
Sets an entry (name/value pair).

- char [set](#) (const char *entry, float value, int precision)
Sets an entry (name/value pair).
- char [set](#) (const char *entry, float value)
Sets an entry (name/value pair).
- char [set](#) (const char *entry, int value)
Sets an entry (name/value pair).
- int [size](#) (const char *entry)
Returns the size of the value part of an entry.
- [~Fl_Preferences](#) ()
The destructor removes allocated resources.

Friends

- class **Node**
- class **RootNode**

30.68.1 Detailed Description

[Fl_Preferences](#) provides methods to store user settings between application starts.

It is similar to the Registry on WIN32 and Preferences on MacOS, and provides a simple configuration mechanism for UNIX.

[Fl_Preferences](#) uses a hierarchy to store data. It bundles similar data into groups and manages entries into those groups as name/value pairs.

Preferences are stored in text files that can be edited manually. The file format is easy to read and relatively forgiving. Preferences files are the same on all platforms. User comments in preference files are preserved. Filenames are unique for each application by using a vendor/application naming scheme. The user must provide default values for all entries to ensure proper operation should preferences be corrupted or not yet exist.

Entries can be of any length. However, the size of each preferences file should be kept under 100k for performance reasons. One application can have multiple preferences files. Extensive binary data however should be stored in separate files: see [getUserdataPath\(\)](#).

Note:

Starting with FLTK 1.3, preference databases are expected to be in utf8 encoding. Previous databases were stored in the current character set or code page which renders them incompatible for text entries using international characters.

30.68.2 Member Enumeration Documentation

30.68.2.1 enum [Fl_Preferences::Root](#)

Define the scope of the preferences.

Enumerator:

SYSTEM Preferences are used system-wide.

USER Preferences apply only to the current user.

30.68.3 Constructor & Destructor Documentation**30.68.3.1 FL_Preferences::FL_Preferences (Root *root*, const char * *vendor*, const char * *application*)**

The constructor creates a group that manages name/value pairs and child groups.

Groups are ready for reading and writing at any time. The root argument is either [FL_Preferences::USER](#) or [FL_Preferences::SYSTEM](#).

This constructor creates the *base* instance for all following entries and reads existing databases into memory. The vendor argument is a unique text string identifying the development team or vendor of an application. A domain name or an EMail address are great unique names, e.g. "researchATmatthiasm.com" or "ftk.org". The application argument can be the working title or final name of your application. Both vendor and application must be valid relative UNIX pathnames and may contain '/'s to create deeper file structures.

Parameters:

- ← **root** can be USER or SYSTEM for user specific or system wide preferences
- ← **vendor** unique text describing the company or author of this file
- ← **application** unique text describing the application

30.68.3.2 FL_Preferences::FL_Preferences (const char * *path*, const char * *vendor*, const char * *application*)

Use this constructor to create or read a preferences file at an arbitrary position in the file system.

The file name is generated in the form *path/application.prefs*. If application is NULL, path must contain the full file name.

Parameters:

- ← **path** path to the directory that contains the preferences file
- ← **vendor** unique text describing the company or author of this file
- ← **application** unique text describing the application

30.68.3.3 FL_Preferences::FL_Preferences (FL_Preferences & *parent*, const char * *group*)

Generate or read a new group of entries within another group.

Use the *group* argument to name the group that you would like to access. Group can also contain a path to a group further down the hierarchy by separating group names with a forward slash '/'.

Parameters:

- ← **parent** reference object for the new group
- ← **group** name of the group to access (may contain '/'s)

30.68.3.4 FL_Preferences::FL_Preferences (FL_Preferences * *parent*, const char * *group*)

See also:

[FL_Preferences\(FL_Preferences&, const char *group \)](#)

30.68.3.5 FL_Preferences::~~FL_Preferences ()

The destructor removes allocated resources.

When used on the *base* preferences group, the destructor flushes all changes to the preferences file and deletes all internal databases.

The destructor does not remove any data from the database. It merely deletes your reference to the database.

30.68.4 Member Function Documentation

30.68.4.1 char FL_Preferences::deleteEntry (const char * *key*)

Deletes a single name/value pair.

This function removes the entry *key* from the database.

Parameters:

← *key* name of entry to delete

Returns:

0 if deleting the entry failed

30.68.4.2 char FL_Preferences::deleteGroup (const char * *group*)

Deletes a group.

Removes a group and all keys and groups within that group from the database.

Parameters:

← *group* name of the group to delete

Returns:

0 if call failed

30.68.4.3 int FL_Preferences::entries ()

Returns the number of entries (name/value pairs) in a group.

Returns:

number of entries

30.68.4.4 `const char * Fl_Preferences::entry (int index)`

Returns the name of an entry.

There is no guaranteed order of entry names. The index must be within the range given by [entries\(\)](#).

Parameters:

← *index* number indexing the requested entry

Returns:

pointer to value cstring

30.68.4.5 `char Fl_Preferences::entryExists (const char * key)`

Returns non-zero if an entry with this name exists.

Parameters:

← *key* name of entry that is searched for

Returns:

0 if entry was not found

30.68.4.6 `void Fl_Preferences::flush ()`

Writes all preferences to disk.

This function works only with the base preferences group. This function is rarely used as deleting the base preferences flushes automatically.

30.68.4.7 `char Fl_Preferences::get (const char * key, void * data, const void * defaultValue, int defaultSize, int maxSize)`

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). 'maxSize' is the maximum length of text that will be read.

Parameters:

← *key* name of entry

→ *data* value returned from preferences or default value if none was set

← *defaultValue* default value to be used if no preference was set

← *defaultSize* size of default value array

← *maxSize* maximum length of value

Returns:

0 if the default value was used

Todo

maxSize should receive the number of bytes that were read.

30.68.4.8 char Fl_Preferences::get (const char * *key*, void *& *data*, const void * *defaultValue*, int *defaultSize*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). `get()` allocates memory of sufficient size to hold the value. The buffer must be free'd by the developer using `'free(value)'`.

Parameters:

- ← *key* name of entry
- *data* returned from preferences or default value if none was set
- ← *defaultValue* default value to be used if no preference was set
- ← *defaultSize* size of default value array

Returns:

- 0 if the default value was used

30.68.4.9 char Fl_Preferences::get (const char * *key*, char * *text*, const char * *defaultValue*, int *maxSize*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). '*maxSize*' is the maximum length of text that will be read. The text buffer must allow for one additional byte for a trailing zero.

Parameters:

- ← *key* name of entry
- *text* returned from preferences or default value if none was set
- ← *defaultValue* default value to be used if no preference was set
- ← *maxSize* maximum length of value plus one byte for a trailing zero

Returns:

- 0 if the default value was used

30.68.4.10 char Fl_Preferences::get (const char * *key*, char *& *text*, const char * *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0). `get()` allocates memory of sufficient size to hold the value. The buffer must be free'd by the developer using `'free(value)'`.

Parameters:

- ← *key* name of entry
- *text* returned from preferences or default value if none was set

← *defaultValue* default value to be used if no preference was set

Returns:

0 if the default value was used

30.68.4.11 char Fl_Preferences::get (const char * *key*, double & *value*, double *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters:

← *key* name of entry

→ *value* returned from preferences or default value if none was set

← *defaultValue* default value to be used if no preference was set

Returns:

0 if the default value was used

30.68.4.12 char Fl_Preferences::get (const char * *key*, float & *value*, float *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters:

← *key* name of entry

→ *value* returned from preferences or default value if none was set

← *defaultValue* default value to be used if no preference was set

Returns:

0 if the default value was used

30.68.4.13 char Fl_Preferences::get (const char * *key*, int & *value*, int *defaultValue*)

Reads an entry from the group.

A default value must be supplied. The return value indicates if the value was available (non-zero) or the default was used (0).

Parameters:

← *key* name of entry

→ *value* returned from preferences or default value if none was set

← *defaultValue* default value to be used if no preference was set

Returns:

0 if the default value was used

30.68.4.14 char FL_Preferences::getUserdataPath (char * *path*, int *pathlen*)

Creates a path that is related to the preferences file and that is usable for additional application data.

This function creates a directory that is named after the preferences database without the .prefs extension and located in the same directory. It then fills the given buffer with the complete path name.

Exmample:

```
FL_Preferences prefs( USER, "matthiasm.com", "test" );
char path[FL_PATH_MAX];
prefs.getUserdataPath( path );
```

creates the preferences database in (MS Windows):

```
c:/Documents and Settings/matt/Application Data/matthiasm.com/test.prefs
```

and returns the userdata path:

```
c:/Documents and Settings/matt/Application Data/matthiasm.com/test/
```

Parameters:

- *path* buffer for user data path
- ← *pathlen* size of path buffer (should be at least FL_PATH_MAX)

Returns:

0 if path was not created or pathname can't fit into buffer

30.68.4.15 const char * FL_Preferences::group (int *num_group*)

Returns the name of the Nth (*num_group*) group.

There is no guaranteed order of group names. The index must be within the range given by [groups\(\)](#).

Parameters:

- ← *num_group* number indexing the requested group

Returns:

'C' string pointer to the group name

30.68.4.16 char FL_Preferences::groupExists (const char * *key*)

Returns non-zero if a group with this name exists.

Group names are relative to the Preferences node and can contain a path. "." describes the current node, "/" describes the topmost node. By preceding a groupname with a "/", its path becomes relative to the topmost node.

Parameters:

- ← *key* name of group that is searched for

Returns:

0 if no group by that name was found

30.68.4.17 int Fl_Preferences::groups ()

Returns the number of groups that are contained within a group.

Returns:

0 for no groups at all

30.68.4.18 char Fl_Preferences::set (const char * *key*, const void * *data*, int *dsize*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

- ← *key* name of entry
- ← *data* set this entry to value
- ← *dsize* size of data array

Returns:

0 if setting the value failed

30.68.4.19 char Fl_Preferences::set (const char * *key*, const char * *text*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

- ← *key* name of entry
- ← *text* set this entry to value

Returns:

0 if setting the value failed

30.68.4.20 char Fl_Preferences::set (const char * *key*, double *value*, int *precision*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

- ← *key* name of entry
- ← *value* set this entry to value

← *precision* number of decimal digits to represent value

Returns:

0 if setting the value failed

30.68.4.21 char Fl_Preferences::set (const char * *key*, double *value*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

← *key* name of entry

← *value* set this entry to value

Returns:

0 if setting the value failed

30.68.4.22 char Fl_Preferences::set (const char * *key*, float *value*, int *precision*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

← *key* name of entry

← *value* set this entry to value

← *precision* number of decimal digits to represent value

Returns:

0 if setting the value failed

30.68.4.23 char Fl_Preferences::set (const char * *key*, float *value*)

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

← *key* name of entry

← *value* set this entry to value

Returns:

0 if setting the value failed

30.68.4.24 `char Fl_Preferences::set (const char * key, int value)`

Sets an entry (name/value pair).

The return value indicates if there was a problem storing the data in memory. However it does not reflect if the value was actually stored in the preferences file.

Parameters:

- ← *key* name of entry
- ← *value* set this entry to *value*

Returns:

- 0 if setting the value failed

30.68.4.25 `int Fl_Preferences::size (const char * key)`

Returns the size of the value part of an entry.

Parameters:

- ← *key* name of entry

Returns:

- size of value

The documentation for this class was generated from the following files:

- Fl_Preferences.H
- Fl_Preferences.cxx

30.69 Fl_Preferences::Entry Struct Reference

An entry associates a preference name to its corresponding value.

```
#include <Fl_Preferences.H>
```

Public Attributes

- char * **name**
- char * **value**

30.69.1 Detailed Description

An entry associates a preference name to its corresponding value.

The documentation for this struct was generated from the following file:

- Fl_Preferences.H

30.70 Fl_Preferences::Name Class Reference

'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.

```
#include <Fl_Preferences.H>
```

Public Member Functions

- [Name](#) (const char *format,...)
Creates a group name or entry name on the fly.
- [Name](#) (unsigned int n)
Creates a group name or entry name on the fly.
- [operator const char *](#) ()
Return the [Name](#) as a "C" string.

30.70.1 Detailed Description

'Name' provides a simple method to create numerical or more complex procedural names for entries and groups on the fly.

Example: `prefs.set(Fl_Preferences::Name("File%d",i),file[i]);`.

See `test/preferences.cxx` as a sample for writing arrays into preferences.

'Name' is actually implemented as a class inside [Fl_Preferences](#). It casts into `const char*` and gets automatically destroyed after the enclosing call ends.

30.70.2 Constructor & Destructor Documentation

30.70.2.1 Fl_Preferences::Name::Name (unsigned int *n*)

Creates a group name or entry name on the fly.

This version creates a simple unsigned integer as an entry name.

```
int n, i;
Fl_Preferences prev( appPrefs, "PreviousFiles" );
prev.get( "n", 0 );
for ( i=0; i<n; i++ )
    prev.get( Fl_Preferences::Name(i), prevFile[i], " " );
```

30.70.2.2 Fl_Preferences::Name::Name (const char **format*, ...)

Creates a group name or entry name on the fly.

This version creates entry names as in 'printf'.


```
int n, i;
Fl_Preferences prefs( USER, "matthiasm.com", "test" );
prev.get( "nFiles", 0 );
for ( i=0; i<n; i++ )
    prev.get( Fl_Preferences::Name( "File%d", i ), prevFile[i], "" );
```

The documentation for this class was generated from the following files:

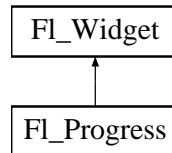
- Fl_Preferences.H
- Fl_Preferences.cxx

30.71 Fl_Progress Class Reference

Displays a progress bar for the user.

```
#include <Fl_Progress.H>
```

Inheritance diagram for Fl_Progress::



Public Member Functions

- [Fl_Progress](#) (int x, int y, int w, int h, const char *l=0)
The constructor creates the progress bar using the position, size, and label.
- float [maximum](#) () const
Gets the maximum value in the progress widget.
- void [maximum](#) (float v)
Sets the maximum value in the progress widget.
- float [minimum](#) () const
Gets the minimum value in the progress widget.
- void [minimum](#) (float v)
Sets the minimum value in the progress widget.
- float [value](#) () const
Gets the current value in the progress widget.
- void [value](#) (float v)
Sets the current value in the progress widget.

Protected Member Functions

- virtual void [draw](#) ()
Draws the progress bar.

30.71.1 Detailed Description

Displays a progress bar for the user.

30.71.2 Constructor & Destructor Documentation

30.71.2.1 FL_Progress::FL_Progress (int X, int Y, int W, int H, const char * l = 0)

The constructor creates the progress bar using the position, size, and label.

The inherited destructor removes the progress bar.

30.71.3 Member Function Documentation

30.71.3.1 void FL_Progress::draw (void) [protected, virtual]

Draws the progress bar.

Implements [FL_Widget](#).

30.71.3.2 float FL_Progress::maximum () const [inline]

Gets the maximum value in the progress widget.

30.71.3.3 void FL_Progress::maximum (float v) [inline]

Sets the maximum value in the progress widget.

30.71.3.4 float FL_Progress::minimum () const [inline]

Gets the minimum value in the progress widget.

30.71.3.5 void FL_Progress::minimum (float v) [inline]

Sets the minimum value in the progress widget.

30.71.3.6 float FL_Progress::value () const [inline]

Gets the current value in the progress widget.

30.71.3.7 void FL_Progress::value (float v) [inline]

Sets the current value in the progress widget.

The documentation for this class was generated from the following files:

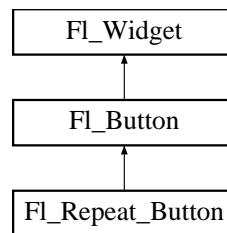
- FL_Progress.H
- FL_Progress.cxx

30.72 Fl_Repeat_Button Class Reference

The [Fl_Repeat_Button](#) is a subclass of [Fl_Button](#) that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down.

```
#include <Fl_Repeat_Button.H>
```

Inheritance diagram for `Fl_Repeat_Button`:



Public Member Functions

- void [deactivate](#) ()
Deactivates the widget.
- [Fl_Repeat_Button](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Repeat_Button](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.

30.72.1 Detailed Description

The [Fl_Repeat_Button](#) is a subclass of [Fl_Button](#) that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down.

The speed of the repeat is fixed and depends on the implementation.

30.72.2 Constructor & Destructor Documentation

30.72.2.1 `Fl_Repeat_Button::Fl_Repeat_Button (int X, int Y, int W, int H, const char *l = 0)`
[inline]

Creates a new [Fl_Repeat_Button](#) widget using the given position, size, and label string.

The default boxtype is `FL_UP_BOX`. Deletes the button.

30.72.3 Member Function Documentation

30.72.3.1 `void Fl_Repeat_Button::deactivate ()` [inline]

Deactivates the widget.

Inactive widgets will be drawn "grayed out", e.g. with less contrast than the active widget. Inactive widgets will not receive any keyboard or mouse button events. Other events (including FL_ENTER, FL_MOVE, FL_LEAVE, FL_SHORTCUT, and others) will still be sent. A widget is only active if [active\(\)](#) is true on it *and all of its parents*.

Changing this value will send FL_DEACTIVATE to the widget if [active_r\(\)](#) is true.

Currently you cannot deactivate [Fl_Window](#) widgets.

See also:

[activate\(\)](#), [active\(\)](#), [active_r\(\)](#)

Reimplemented from [Fl_Widget](#).

30.72.3.2 int Fl_Repeat_Button::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Button](#).

The documentation for this class was generated from the following files:

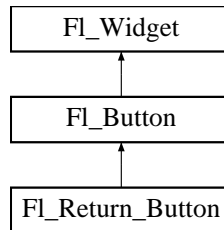
- Fl_Repeat_Button.H
- Fl_Repeat_Button.cxx

30.73 Fl_Return_Button Class Reference

The [Fl_Return_Button](#) is a subclass of [Fl_Button](#) that generates a callback when it is pressed or when the user presses the Enter key.

```
#include <Fl_Return_Button.H>
```

Inheritance diagram for [Fl_Return_Button](#)::



Public Member Functions

- [Fl_Return_Button](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [Fl_Return_Button](#) widget using the given position, size, and label string.

- int [handle](#) (int)

Handles the specified event.

Protected Member Functions

- void [draw](#) ()

Draws the widget.

30.73.1 Detailed Description

The [Fl_Return_Button](#) is a subclass of [Fl_Button](#) that generates a callback when it is pressed or when the user presses the Enter key.

A carriage-return symbol is drawn next to the button label.

Figure 30.23: Fl_Return_Button

30.73.2 Constructor & Destructor Documentation

30.73.2.1 Fl_Return_Button::Fl_Return_Button (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0) [inline]

Creates a new [Fl_Return_Button](#) widget using the given position, size, and label string.

The default boxtype is FL_UP_BOX.

The inherited destructor deletes the button.

30.73.3 Member Function Documentation

30.73.3.1 void Fl_Return_Button::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to Fl_Widget* and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Button](#).

30.73.3.2 int Fl_Return_Button::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0* if the event was not used or understood
- 1* if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Button](#).

The documentation for this class was generated from the following files:

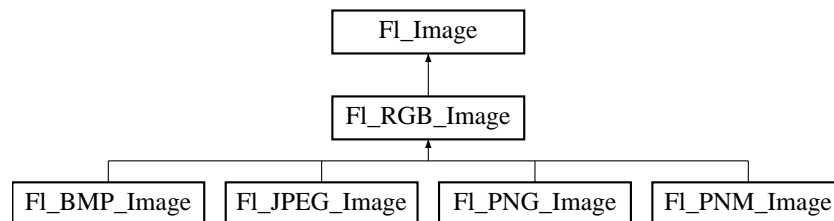
- Fl_Return_Button.H
- Fl_Return_Button.cxx

30.74 FL_RGB_Image Class Reference

The [FL_RGB_Image](#) class supports caching and drawing of full-color images with 1 to 4 channels of color information.

```
#include <Fl_Image.H>
```

Inheritance diagram for FL_RGB_Image::



Public Member Functions

- virtual void [color_average](#) ([FL_Color](#) c, float i)
The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.
- [FL_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [FL_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- virtual void [desaturate](#) ()
The [desaturate\(\)](#) method converts an image to grayscale.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx=0, int cy=0)
The [draw\(\)](#) methods draw the image.
- [FL_RGB_Image](#) (const [uchar](#) *bits, int W, int H, int D=3, int LD=0)
The constructor creates a new image from the specified data.
- virtual void [label](#) ([FL_Menu_Item](#) *m)
The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void [label](#) ([FL_Widget](#) *w)
The [label\(\)](#) methods are an obsolete way to set the image attribute of a widget or menu item.
- virtual void [uncache](#) ()
If the image has been cached for display, delete the cache data.
- virtual [~FL_RGB_Image](#) ()
The destructor free all memory and server resources that are used by the image.

Public Attributes

- int **alloc_array**
- const [uchar](#) * **array**
- unsigned **id**
- unsigned **mask**

30.74.1 Detailed Description

The [Fl_RGB_Image](#) class supports caching and drawing of full-color images with 1 to 4 channels of color information.

Images with an even number of channels are assumed to contain alpha information, which is used to blend the image with the contents of the screen.

[Fl_RGB_Image](#) is defined in [<FL/Fl_Image.H>](#), however for compatibility reasons [<FL/Fl_RGB_Image.H>](#) should be included.

30.74.2 Constructor & Destructor Documentation

30.74.2.1 [Fl_RGB_Image::Fl_RGB_Image](#) (const [uchar](#) * *bits*, int *W*, int *H*, int *D* = 3, int *LD* = 0) [\[inline\]](#)

The constructor creates a new image from the specified data.

30.74.2.2 [Fl_RGB_Image::~~Fl_RGB_Image](#) () [\[virtual\]](#)

The destructor free all memory and server resources that are used by the image.

30.74.3 Member Function Documentation

30.74.3.1 void [Fl_RGB_Image::color_average](#) ([Fl_Color](#) *c*, float *i*) [\[virtual\]](#)

The [color_average\(\)](#) method averages the colors in the image with the FLTK color value *c*.

The *i* argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color. *The original image data is not altered by this method.*

Reimplemented from [Fl_Image](#).

30.74.3.2 [Fl_Image* Fl_RGB_Image::copy](#) () [\[inline\]](#)

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [Fl_Shared_Image](#), released) when you are done with it.

Reimplemented from [Fl_Image](#).

30.74.3.3 `FL_Image * FL_RGB_Image::copy (int W, int H)` [virtual]

The `copy()` method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of `FL_Shared_Image`, released) when you are done with it.

Reimplemented from `FL_Image`.

30.74.3.4 `void FL_RGB_Image::desaturate ()` [virtual]

The `desaturate()` method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved. *This method does not alter the original image data.*

Reimplemented from `FL_Image`.

30.74.3.5 `void FL_RGB_Image::draw (int X, int Y)` [inline]

The `draw()` methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented from `FL_Image`.

30.74.3.6 `void FL_RGB_Image::draw (int X, int Y, int W, int H, int cx = 0, int cy = 0)`
[virtual]

The `draw()` methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the `cx` and `cy` arguments.

Reimplemented from `FL_Image`.

30.74.3.7 `void FL_RGB_Image::label (FL_Menu_Item * m)` [virtual]

The `label()` methods are an obsolete way to set the image attribute of a widget or menu item.

Use the `image()` or `deimage()` methods of the `FL_Widget` and `FL_Menu_Item` classes instead.

Reimplemented from `FL_Image`.

30.74.3.8 `void FL_RGB_Image::label (FL_Widget * widget)` [virtual]

The `label()` methods are an obsolete way to set the image attribute of a widget or menu item.

Use the `image()` or `deimage()` methods of the `FL_Widget` and `FL_Menu_Item` classes instead.

Reimplemented from `FL_Image`.

30.74.3.9 `void FL_RGB_Image::uncache ()` [virtual]

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from [Fl_Image](#).

The documentation for this class was generated from the following files:

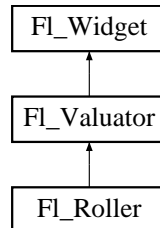
- `Fl_Image.H`
- `Fl_Image.cxx`

30.75 Fl_Roller Class Reference

The [Fl_Roller](#) widget is a "dolly" control commonly used to move 3D objects.

```
#include <Fl_Roller.H>
```

Inheritance diagram for Fl_Roller::



Public Member Functions

- [Fl_Roller](#) (int X, int Y, int W, int H, const char *L=0)
Creates a new [Fl_Roller](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.75.1 Detailed Description

The [Fl_Roller](#) widget is a "dolly" control commonly used to move 3D objects.

Figure 30.24: Fl_Roller

30.75.2 Constructor & Destructor Documentation

30.75.2.1 `Fl_Roller::Fl_Roller (int X, int Y, int W, int H, const char * L = 0)`

Creates a new [Fl_Roller](#) widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

Inherited destructor destroys the valuator.

30.75.3 Member Function Documentation

30.75.3.1 `void Fl_Roller::draw ()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.75.3.2 `int Fl_Roller::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

The documentation for this class was generated from the following files:

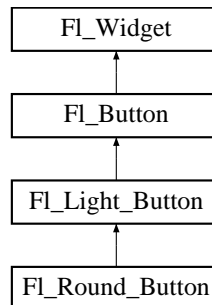
- [Fl_Roller.H](#)
- [Fl_Roller.cxx](#)

30.76 Fl_Round_Button Class Reference

Buttons generate callbacks when they are clicked by the user.

```
#include <Fl_Round_Button.H>
```

Inheritance diagram for Fl_Round_Button::



Public Member Functions

- [Fl_Round_Button](#) (int x, int y, int w, int h, const char *l=0)

Creates a new [Fl_Round_Button](#) widget using the given position, size, and label string.

30.76.1 Detailed Description

Buttons generate callbacks when they are clicked by the user.

You control exactly when and how by changing the values for [type\(\)](#) and [when\(\)](#).

Figure 30.25: Fl_Round_Button

The [Fl_Round_Button](#) subclass display the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_ROUND_DOWN_BOX. The color of the light when on is controlled with [selection_color\(\)](#), which defaults to FL_RED.

The documentation for this class was generated from the following files:

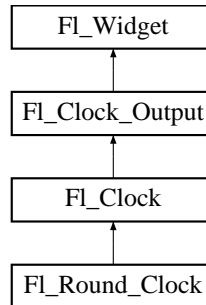
- `Fl_Round_Button.H`
- `Fl_Round_Button.cxx`

30.77 Fl_Round_Clock Class Reference

A clock widget of type FL_ROUND_CLOCK.

```
#include <Fl_Round_Clock.H>
```

Inheritance diagram for Fl_Round_Clock::



Public Member Functions

- [Fl_Round_Clock](#) (int x, int y, int w, int h, const char *l=0)

Creates the clock widget, setting his type and box.

30.77.1 Detailed Description

A clock widget of type FL_ROUND_CLOCK.

Has no box.

30.77.2 Constructor & Destructor Documentation

30.77.2.1 Fl_Round_Clock::Fl_Round_Clock (int x, int y, int w, int h, const char * l = 0) [inline]

Creates the clock widget, setting his type and box.

The documentation for this class was generated from the following file:

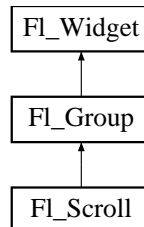
- Fl_Round_Clock.H

30.78 Fl_Scroll Class Reference

This container widget lets you maneuver around a set of widgets much larger than your window.

```
#include <Fl_Scroll.H>
```

Inheritance diagram for Fl_Scroll::



Public Types

- enum {
 HORIZONTAL = 1, **VERTICAL** = 2, **BOTH** = 3, **ALWAYS_ON** = 4,
 HORIZONTAL_ALWAYS = 5, **VERTICAL_ALWAYS** = 6, **BOTH_ALWAYS** = 7 }

Public Member Functions

- void **clear** ()
 Clear all but the scrollbars.
- Fl_Scroll** (int X, int Y, int W, int H, const char *l=0)
 *Creates a new **Fl_Scroll** widget using the given position, size, and label string.*
- int **handle** (int)
 Handles the specified event.
- void **resize** (int, int, int, int)
 *Resizes the **Fl_Group** widget and all of its children.*
- void **scroll_to** (int, int)
 Moves the contents of the scroll group to a new position.
- int **xposition** () const
 Gets the current horizontal scrolling position.
- int **yposition** () const
 Gets the current vertical scrolling position.

Public Attributes

- Fl_Scrollbar** **hscrollbar**
- Fl_Scrollbar** **scrollbar**

Protected Member Functions

- void `bbox` (int &, int &, int &, int &)

Returns the bounding box for the interior of the scrolling area, inside the scrollbars.

- void `draw` ()

Draws the widget.

30.78.1 Detailed Description

This container widget lets you maneuver around a set of widgets much larger than your window.

If the child widgets are larger than the size of this object then scrollbars will appear so that you can scroll over to them:

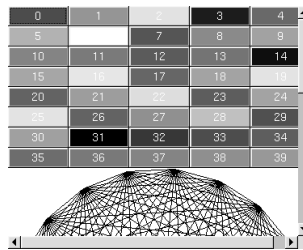


Figure 30.26: Fl_Scroll

If all of the child widgets are packed together into a solid rectangle then you want to set `box()` to `FL_NO_BOX` or one of the `_FRAME` types. This will result in the best output. However, if the child widgets are a sparse arrangement you must set `box()` to a real `_BOX` type. This can result in some blinking during redrawing, but that can be solved by using a `Fl_Double_Window`.

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the area of the scroll.

Use `Fl_Scroll::type()` to change this as follows :

- 0 - No scrollbars
- `Fl_Scroll::HORIZONTAL` - Only a horizontal scrollbar.
- `Fl_Scroll::VERTICAL` - Only a vertical scrollbar.
- `Fl_Scroll::BOTH` - The default is both scrollbars.
- `Fl_Scroll::HORIZONTAL_ALWAYS` - Horizontal scrollbar always on, vertical always off.
- `Fl_Scroll::VERTICAL_ALWAYS` - Vertical scrollbar always on, horizontal always off.
- `Fl_Scroll::BOTH_ALWAYS` - Both always on.

Use `scrollbar.align(int)` (see void `Fl_Widget::align(Fl_Align)`) : to change what side the scrollbars are drawn on.

If the FL_ALIGN_LEFT bit is on, the vertical scrollbar is on the left. If the FL_ALIGN_TOP bit is on, the horizontal scrollbar is on the top. Note that only the alignment flags in scrollbar are considered. The flags in hscrollbar however are ignored.

This widget can also be used to pan around a single child widget "canvas". This child widget should be of your own class, with a `draw()` method that draws the contents. The scrolling is done by changing the `x()` and `y()` of the widget, so this child must use the `x()` and `y()` to position it's drawing. To speed up drawing it should test `fl_push_clip()`.

Another very useful child is a single `FL_Pack`, which is itself a group that packs it's children together and changes size to surround them. Filling the `FL_Pack` with `FL_Tabs` groups (and then putting normal widgets inside those) gives you a very powerful scrolling list of individually-openable panels.

Fluid lets you create these, but you can only lay out objects that fit inside the `FL_Scroll` without scrolling. Be sure to leave space for the scrollbars, as Fluid won't show these either.

You cannot use `FL_Window` as a child of this since the clipping is not conveyed to it when drawn, and it will draw over the scrollbars and neighboring objects.

30.78.2 Constructor & Destructor Documentation

30.78.2.1 `FL_Scroll::FL_Scroll (int X, int Y, int W, int H, const char * L = 0)`

Creates a new `FL_Scroll` widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `FL_Scroll` and all of it's children can be automatic (local) variables, but you must declare the `FL_Scroll` first, so that it is destroyed last.

30.78.3 Member Function Documentation

30.78.3.1 `void FL_Scroll::bbox (int & X, int & Y, int & W, int & H)` [protected]

Returns the bounding box for the interior of the scrolling area, inside the scrollbars.

Currently this is only reliable after `draw()`, and before any resizing of the `FL_Scroll` or any child widgets occur.

Todo

The visibility of the scrollbars ought to be checked/calculated outside of the `draw()` method (STR #1895).

30.78.3.2 `void FL_Scroll::clear ()`

Clear all but the scrollbars.

..

Reimplemented from `FL_Group`.

30.78.3.3 void `Fl_Scroll::draw()` [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Group](#).

30.78.3.4 int `Fl_Scroll::handle(int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Group](#).

30.78.3.5 void `Fl_Scroll::resize(int X, int Y, int W, int H)` [virtual]

Resizes the [Fl_Group](#) widget and all of its children.

The [Fl_Group](#) widget first resizes itself, and then it moves and resizes all its children according to the rules documented for [Fl_Group::resizable\(Fl_Widget*\)](#)

See also:

[Fl_Group::resizable\(Fl_Widget*\)](#)
[Fl_Group::resizable\(\)](#)
[Fl_Widget::resize\(int,int,int,int\)](#)

Reimplemented from [Fl_Group](#).

30.78.3.6 void Fl_Scroll::scroll_to (int X, int Y)

Moves the contents of the scroll group to a new position.

30.78.3.7 int Fl_Scroll::xposition () const [inline]

Gets the current horizontal scrolling position.

30.78.3.8 int Fl_Scroll::yposition () const [inline]

Gets the current vertical scrolling position.

The documentation for this class was generated from the following files:

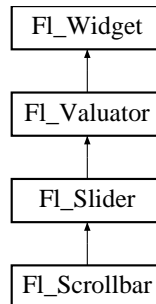
- Fl_Scroll.H
- Fl_Scroll.cxx

30.79 FL_Scrollbar Class Reference

The [FL_Scrollbar](#) widget displays a slider with arrow buttons at the ends of the scrollbar.

```
#include <FL_Scrollbar.H>
```

Inheritance diagram for FL_Scrollbar::



Public Member Functions

- [FL_Scrollbar](#) (int X, int Y, int W, int H, const char *L=0)
Creates a new [FL_Scrollbar](#) widget with given position, size, and label.
- int [handle](#) (int)
Handles the specified event.
- void [linesize](#) (int i)
This number controls how big the steps are that the arrow keys do.
- int [linesize](#) () const
Get the size of step, in lines, that the arrow keys move.
- int [value](#) (int pos, int size, int first, int total)
Sets the position, size and range of the slider in the scrollbar.
- int [value](#) (int p)
Sets the value (position) of the slider in the scrollbar.
- int [value](#) () const
Gets the integer value (position) of the slider in the scrollbar.
- [~FL_Scrollbar](#) ()
Destroys the Scrollbar.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.79.1 Detailed Description

The [FL_Scrollbar](#) widget displays a slider with arrow buttons at the ends of the scrollbar.

Clicking on the arrows move up/left and down/right by [linesize\(\)](#). Scrollbars also accept FL_SHORTCUT events: the arrows move by [linesize\(\)](#), and vertical scrollbars take Page Up/Down (they move by the page size minus [linesize\(\)](#)) and Home/End (they jump to the top or bottom).

Scrollbars have [step\(1\)](#) preset (they always return integers). If desired you can set the [step\(\)](#) to non-integer values. You will then have to use casts to get at the floating-point versions of [value\(\)](#) from [FL_Slider](#).

Figure 30.27: FL_Scrollbar

30.79.2 Constructor & Destructor Documentation

30.79.2.1 [FL_Scrollbar::FL_Scrollbar](#) (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates a new [FL_Scrollbar](#) widget with given position, size, and label.

You need to do `type(FL_HORIZONTAL)` if you want a horizontal scrollbar.

30.79.2.2 [FL_Scrollbar::~~FL_Scrollbar](#) ()

Destroys the Scrollbar.

30.79.3 Member Function Documentation

30.79.3.1 [void FL_Scrollbar::draw](#) () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `FL_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Slider](#).

30.79.3.2 `int Fl_Scrollbar::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Slider](#).

30.79.3.3 `void Fl_Scrollbar::linesize (int i)` [inline]

This number controls how big the steps are that the arrow keys do.

In addition page up/down move by the size last sent to [value\(\)](#) minus one [linesize\(\)](#). The default is 16.

30.79.3.4 `int Fl_Scrollbar::value (int pos, int size, int first, int total)` [inline]

Sets the position, size and range of the slider in the scrollbar.

Parameters:

- ← *pos* position, first line displayed
- ← *size* window size, number of lines displayed
- ← *first* number of first line
- ← *total* total number of lines

You should call this every time your window changes size, your data changes size, or your scroll position changes (even if in response to a callback from this scrollbar). All necessary calls to [redraw\(\)](#) are done.

Calls [Fl_Slider::scrollvalue\(int pos, int size, int first, int total\)](#).

30.79.3.5 int FL_Scrollbar::value (int *p*) [inline]

Sets the value (position) of the slider in the scrollbar.

See also:

[FL_Scrollbar::value\(\)](#)
[FL_Scrollbar::value\(int pos, int size, int first, int total\)](#)

30.79.3.6 int FL_Scrollbar::value () const [inline]

Gets the integer value (position) of the slider in the scrollbar.

You can get the floating point value with [FL_Slider::value\(\)](#).

See also:

[FL_Scrollbar::value\(int p\)](#)
[FL_Scrollbar::value\(int pos, int size, int first, int total\)](#)

Reimplemented from [FL_Valuator](#).

The documentation for this class was generated from the following files:

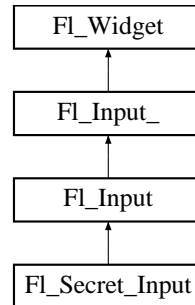
- [FL_Scrollbar.H](#)
- [FL_Scrollbar.cxx](#)

30.80 FL_Secret_Input Class Reference

The [FL_Secret_Input](#) class is a subclass of [FL_Input](#) that displays its input as a string of asterisks.

```
#include <FL_Secret_Input.H>
```

Inheritance diagram for FL_Secret_Input::



Public Member Functions

- [FL_Secret_Input](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [FL_Secret_Input](#) widget using the given position, size, and label string.

30.80.1 Detailed Description

The [FL_Secret_Input](#) class is a subclass of [FL_Input](#) that displays its input as a string of asterisks.

This subclass is usually used to receive passwords and other "secret" information.

30.80.2 Constructor & Destructor Documentation

30.80.2.1 FL_Secret_Input::FL_Secret_Input (int X, int Y, int W, int H, const char * l = 0) [inline]

Creates a new [FL_Secret_Input](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

Inherited destructor destroys the widget and any value associated with it.

The documentation for this class was generated from the following file:

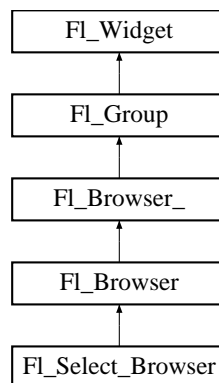
- [FL_Secret_Input.H](#)

30.81 Fl_Select_Browser Class Reference

The class is a subclass of [Fl_Browser](#) which lets the user select a single item, or no items by clicking on the empty space.

```
#include <Fl_Select_Browser.H>
```

Inheritance diagram for Fl_Select_Browser::



Public Member Functions

- [Fl_Select_Browser](#) (int X, int Y, int W, int H, const char *l=0)

Creates a new [Fl_Select_Browser](#) widget using the given position, size, and label string.

30.81.1 Detailed Description

The class is a subclass of [Fl_Browser](#) which lets the user select a single item, or no items by clicking on the empty space.

As long as the mouse button is held down on an unselected item it is highlighted. Normally the callback is done when the user presses the mouse, but you can change this with [when\(\)](#).

See [Fl_Browser](#) for methods to add and remove lines from the browser.

30.81.2 Constructor & Destructor Documentation

30.81.2.1 Fl_Select_Browser::Fl_Select_Browser (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [Fl_Select_Browser](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX. The constructor specializes [Fl_Browser\(\)](#) by setting the type to FL_SELECT_BROWSER. The destructor destroys the widget and frees all memory that has been allocated.

The documentation for this class was generated from the following file:

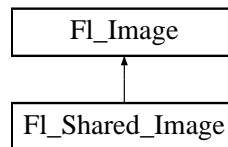
- Fl_Select_Browser.H

30.82 FL_Shared_Image Class Reference

This class supports caching, loading, and drawing of image files.

```
#include <Fl_Shared_Image.H>
```

Inheritance diagram for FL_Shared_Image::



Public Member Functions

- virtual void [color_average](#) ([FL_Color](#) c, float i)
The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.
- [FL_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [FL_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- virtual void [desaturate](#) ()
The [desaturate\(\)](#) method converts an image to grayscale.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx, int cy)
The [draw\(\)](#) methods draw the image.
- const char * [name](#) ()
Returns the filename of the shared image.
- int [refcount](#) ()
Returns the number of references of this shared image.
- void [release](#) ()
Releases and possibly destroys (if refcount <=0) a shared image.
- void [reload](#) ()
Reloads the shared image from disk.
- virtual void [uncache](#) ()
If the image has been cached for display, delete the cache data.

Static Public Member Functions

- static void [add_handler](#) (Fl_Shared_Handler f)
Adds a shared image handler, which is basically a test function for adding new formats.
- static [Fl_Shared_Image](#) * [find](#) (const char *n, int W=0, int H=0)
Finds a shared image from its named and size specifications.
- static [Fl_Shared_Image](#) * [get](#) (const char *n, int W=0, int H=0)
Gets a shared image, if it exists already ; it will return it.
- static [Fl_Shared_Image](#) ** [images](#) ()
Returns the Fl_Shared_Image array.*
- static int [num_images](#) ()
Returns the total number of shared images in the array.
- static void [remove_handler](#) (Fl_Shared_Handler f)
Removes a shared image handler.

Protected Member Functions

- void [add](#) ()
- [Fl_Shared_Image](#) (const char *n, [Fl_Image](#) *img=0)
Creates a shared image from its filename and its corresponding Fl_Image img.*
- [Fl_Shared_Image](#) ()
Creates an empty shared image.
- void [update](#) ()
- virtual [~Fl_Shared_Image](#) ()
The destructor free all memory and server resources that are used by the image.

Static Protected Member Functions

- static int [compare](#) ([Fl_Shared_Image](#) **i0, [Fl_Shared_Image](#) **i1)

Protected Attributes

- int [alloc_image_](#)
- [Fl_Image](#) * [image_](#)
- const char * [name_](#)
- int [original_](#)
- int [refcount_](#)

Static Protected Attributes

- static int `alloc_handlers_` = 0
- static int `alloc_images_` = 0
- static `Fl_Shared_Handler` * `handlers_` = 0
- static `Fl_Shared_Image` ** `images_` = 0
- static int `num_handlers_` = 0
- static int `num_images_` = 0

30.82.1 Detailed Description

This class supports caching, loading, and drawing of image files.

Most applications will also want to link against the `fltk_images` library and call the `fl_register_images()` function to support standard image formats such as BMP, GIF, JPEG, and PNG.

30.82.2 Constructor & Destructor Documentation

30.82.2.1 `Fl_Shared_Image::Fl_Shared_Image ()` [protected]

Creates an empty shared image.

The constructors create a new shared image record in the image cache.

The constructors are protected and cannot be used directly from a program. Use the [get\(\)](#) method instead.

30.82.2.2 `Fl_Shared_Image::Fl_Shared_Image (const char * n, Fl_Image * img = 0)` [protected]

Creates a shared image from its filename and its corresponding `Fl_Image*` `img`.

The constructors create a new shared image record in the image cache.

The constructors are protected and cannot be used directly from a program. Use the [get\(\)](#) method instead.

30.82.2.3 `Fl_Shared_Image::~~Fl_Shared_Image ()` [protected, virtual]

The destructor free all memory and server resources that are used by the image.

The destructor is protected and cannot be used directly from a program. Use the [Fl_Shared_Image::release\(\)](#) method instead.

30.82.3 Member Function Documentation

30.82.3.1 `void Fl_Shared_Image::color_average (Fl_Color c, float i)` [virtual]

The [color_average\(\)](#) method averages the colors in the image with the FLTK color value `c`.

The `i` argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color. *The original image data is not altered by this method.*

Reimplemented from [Fl_Image](#).

30.82.3.2 `FL_Image* FL_Shared_Image::copy ()` `[inline]`

The `copy()` method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of `FL_Shared_Image`, released) when you are done with it.

Reimplemented from `FL_Image`.

30.82.3.3 `FL_Image * FL_Shared_Image::copy (int W, int H)` `[virtual]`

The `copy()` method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of `FL_Shared_Image`, released) when you are done with it.

Reimplemented from `FL_Image`.

30.82.3.4 `void FL_Shared_Image::desaturate ()` `[virtual]`

The `desaturate()` method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved. *This method does not alter the original image data.*

Reimplemented from `FL_Image`.

30.82.3.5 `void FL_Shared_Image::draw (int X, int Y)` `[inline]`

The `draw()` methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented from `FL_Image`.

30.82.3.6 `void FL_Shared_Image::draw (int X, int Y, int W, int H, int cx, int cy)` `[virtual]`

The `draw()` methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the `cx` and `cy` arguments.

Reimplemented from `FL_Image`.

30.82.3.7 `FL_Shared_Image * FL_Shared_Image::get (const char * n, int W = 0, int H = 0)`
`[static]`

Gets a shared image, if it exists already ; it will return it.

If it does not exist or if it exist but with other size, then the existing image is deleted and replaced by a new image from the `n` filename of the proper dimension. If `n` is not a valid image filename, then `get()` will return `NULL`.

30.82.3.8 `int Fl_Shared_Image::num_images ()` `[static]`

Returns the total number of shared images in the array.

30.82.3.9 `int Fl_Shared_Image::refcount ()` `[inline]`

Returns the number of references of this shared image.

When reference is below 1, the image is deleted.

30.82.3.10 `void Fl_Shared_Image::release ()`

Releases and possibly destroys (if refcount ≤ 0) a shared image.

In the latter case, it will reorganize the shared image array so that no hole will occur.

30.82.3.11 `void Fl_Shared_Image::uncache ()` `[virtual]`

If the image has been cached for display, delete the cache data.

This allows you to change the data used for the image and then redraw it without recreating an image object.

Reimplemented from [Fl_Image](#).

The documentation for this class was generated from the following files:

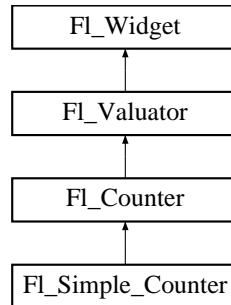
- `Fl_Shared_Image.H`
- `Fl_Shared_Image.cxx`

30.83 Fl_Simple_Counter Class Reference

This widget creates a counter with only 2 arrow buttons.

```
#include <Fl_Simple_Counter.H>
```

Inheritance diagram for Fl_Simple_Counter::



Public Member Functions

- **Fl_Simple_Counter** (int x, int y, int w, int h, const char *l=0)

30.83.1 Detailed Description

This widget creates a counter with only 2 arrow buttons.

Figure 30.28: Fl_Simple_Counter

The documentation for this class was generated from the following file:

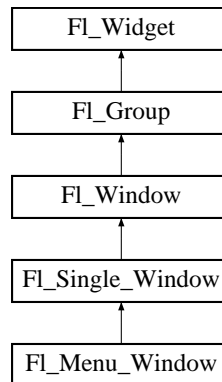
- Fl_Simple_Counter.H

30.84 FL_Single_Window Class Reference

This is the same as [FL_Window](#).

```
#include <Fl_Single_Window.H>
```

Inheritance diagram for `Fl_Single_Window`:



Public Member Functions

- [Fl_Single_Window](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Single_Window](#) widget using the given position, size, and label (title) string.
- [Fl_Single_Window](#) (int W, int H, const char *l=0)
Creates a new [Fl_Single_Window](#) widget using the given size, and label (title) string.
- void [flush](#) ()
Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).
- int [make_current](#) ()
Sets things up so that the drawing functions in [<FL/fl_draw.H>](#) will go into this window.
- void [show](#) (int a, char **b)
See virtual void [Fl_Window::show\(\)](#).
- void [show](#) ()
Put the window on the screen.

30.84.1 Detailed Description

This is the same as [FL_Window](#).

However, it is possible that some implementations will provide double-buffered windows by default. This subclass can be used to force single-buffering. This may be useful for modifying existing programs that use incremental update, or for some types of image data, such as a movie flipbook.

30.84.2 Member Function Documentation

30.84.2.1 void FL_Single_Window::flush () [virtual]

Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Menu_Window](#).

30.84.2.2 int FL_Single_Window::make_current ()

Sets things up so that the drawing functions in <FL/fl_draw.H> will go into this window.

This is useful for incremental update of windows, such as in an idle callback, which will make your program behave much better if it draws a slow graphic. **Danger: incremental update is very hard to debug and maintain!**

This method only works for the [FL_Window](#) and [FL_Gl_Window](#) derived classes.

Reimplemented from [FL_Window](#).

30.84.2.3 void FL_Single_Window::show () [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [FL_Window](#).

Reimplemented in [FL_Menu_Window](#).

The documentation for this class was generated from the following files:

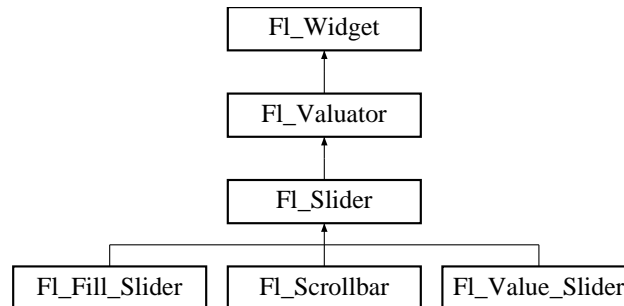
- [FL_Single_Window.H](#)
- [FL_Single_Window.cxx](#)

30.85 FL_Slider Class Reference

The [FL_Slider](#) widget contains a sliding knob inside a box.

```
#include <Fl_Slider.H>
```

Inheritance diagram for FL_Slider::



Public Member Functions

- void [bounds](#) (double a, double b)
Sets the minimum (a) and maximum (b) values for the valuator widget.
- [FL_Slider](#) ([uchar](#) t, int X, int Y, int W, int H, const char *L)
Creates a new [FL_Slider](#) widget using the given box type, position, size, and label string.
- [FL_Slider](#) (int X, int Y, int W, int H, const char *L=0)
Creates a new [FL_Slider](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- int [scrollvalue](#) (int pos, int size, int first, int total)
Sets the size and position of the sliding knob in the box.
- void [slider](#) ([FL_Boxtype](#) c)
Sets the slider box type.
- [FL_Boxtype slider](#) () const
Gets the slider box type.
- void [slider_size](#) (double v)
Set the dimensions of the moving piece of slider.
- float [slider_size](#) () const
Get the dimensions of the moving piece of slider.

Protected Member Functions

- void `draw` ()
Draws the widget.
- void `draw` (int, int, int, int)
- int `handle` (int, int, int, int, int)

30.85.1 Detailed Description

The `Fl_Slider` widget contains a sliding knob inside a box.

It is often used as a scrollbar. Moving the box all the way to the top/left sets it to the `minimum()`, and to the bottom/right to the `maximum()`. The `minimum()` may be greater than the `maximum()` to reverse the slider direction.

Use `void Fl_Widget::type(int)` to set how the slider is drawn, which can be one of the following:

- `FL_VERTICAL` - Draws a vertical slider (this is the default).
- `FL_HORIZONTAL` - Draws a horizontal slider.
- `FL_VERT_FILL_SLIDER` - Draws a filled vertical slider, useful as a progress or value meter.
- `FL_HOR_FILL_SLIDER` - Draws a filled horizontal slider, useful as a progress or value meter.
- `FL_VERT_NICE_SLIDER` - Draws a vertical slider with a nice looking control knob.
- `FL_HOR_NICE_SLIDER` - Draws a horizontal slider with a nice looking control knob.

Figure 30.29: `Fl_Slider`

30.85.2 Constructor & Destructor Documentation

30.85.2.1 `Fl_Slider::Fl_Slider` (int *X*, int *Y*, int *W*, int *H*, const char * *L* = 0)

Creates a new `Fl_Slider` widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

30.85.3 Member Function Documentation

30.85.3.1 void `Fl_Slider::bounds` (double *a*, double *b*)

Sets the minimum (*a*) and maximum (*b*) values for the valuator widget.

if at least one of the values is changed, a partial redraw is asked.

Reimplemented from [Fl_Valuator](#).

30.85.3.2 void `Fl_Slider::draw` () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

Reimplemented in [Fl_Scrollbar](#), and [Fl_Value_Slider](#).

30.85.3.3 int `Fl_Slider::handle` (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited `handle()` method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee `retval`.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

Reimplemented in [Fl_Scrollbar](#), and [Fl_Value_Slider](#).

30.85.3.4 int Fl_Slider::scrollvalue (int *pos*, int *size*, int *first*, int *total*)

Sets the size and position of the sliding knob in the box.

Parameters:

- ← *pos* position of first line displayed
- ← *size* size of window in lines
- ← *first* number of first line
- ← *total* total number of lines Returns Fl_Valuator::value(p)

30.85.3.5 void Fl_Slider::slider (Fl_Boxtype *c*) [inline]

Sets the slider box type.

30.85.3.6 Fl_Boxtype Fl_Slider::slider () const [inline]

Gets the slider box type.

30.85.3.7 void Fl_Slider::slider_size (double *v*)

Set the dimensions of the moving piece of slider.

This is the fraction of the size of the entire widget. If you set this to 1 then the slider cannot move. The default value is .08.

For the "fill" sliders this is the size of the area around the end that causes a drag effect rather than causing the slider to jump to the mouse.

The documentation for this class was generated from the following files:

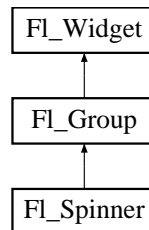
- Fl_Slider.H
- Fl_Slider.cxx

30.86 Fl_Spinner Class Reference

This widget is a combination of the input widget and repeat buttons.

```
#include <Fl_Spinner.H>
```

Inheritance diagram for Fl_Spinner::



Public Member Functions

- **Fl_Spinner** (int X, int Y, int W, int H, const char *L=0)
*Creates a new **Fl_Spinner** widget using the given position, size, and label string.*
- void **format** (const char *f)
Sets or returns the format string for the value.
- const char * **format** ()
Sets or returns the format string for the value.
- int **handle** (int event)
Handles the specified event.
- void **maximum** (double m)
Sets the maximum value of the widget.
- double **maximum** () const
Gets the maximum value of the widget.
- double **maximum** () const
Speling mistakes retained for source compatibility.
- void **minimum** (double m)
Sets the minimum value of the widget.
- double **minimum** () const
Gets the minimum value of the widget.
- double **mininum** () const
Speling mistakes retained for source compatibility.
- void **range** (double a, double b)
Sets the minimum and maximum values for the widget.

- void [resize](#) (int X, int Y, int W, int H)
Resizes the [FL_Group](#) widget and all of its children.
- void [step](#) (double s)
See double [FL_Spinner::step\(\)](#) const.
- double [step](#) () const
Sets or returns the amount to change the value when the user clicks a button.
- void [textcolor](#) (FL_Color c)
Sets the color of the text in the input field.
- FL_Color [textcolor](#) () const
Gets the color of the text in the input field.
- void [textfont](#) (FL_Font f)
Sets the font of the text in the input field.
- FL_Font [textfont](#) () const
Gets the font of the text in the input field.
- void [textsize](#) (FL_Fontsize s)
Sets the size of the text in the input field.
- FL_Fontsize [textsize](#) () const
Gets the size of the text in the input field.
- void [type](#) (uchar v)
See uchar [FL_Spinner::type\(\)](#) const.
- uchar [type](#) () const
Sets or Gets the numeric representation in the input field.
- void [value](#) (double v)
Sets the current value of the widget.
- double [value](#) () const
Gets the current value of the widget.

30.86.1 Detailed Description

This widget is a combination of the input widget and repeat buttons.

The user can either type into the input area or use the buttons to change the value.

30.86.2 Constructor & Destructor Documentation

30.86.2.1 `FL_Spinner::FL_Spinner (int X, int Y, int W, int H, const char * L = 0)` `[inline]`

Creates a new [FL_Spinner](#) widget using the given position, size, and label string.

Inherited destructor Destroys the widget and any value associated with it.

30.86.3 Member Function Documentation

30.86.3.1 `void FL_Spinner::format (const char * f)` `[inline]`

Sets or returns the format string for the value.

30.86.3.2 `const char* FL_Spinner::format ()` `[inline]`

Sets or returns the format string for the value.

30.86.3.3 `int FL_Spinner::handle (int event)` `[inline, virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Group](#).

30.86.3.4 `void FL_Spinner::maximum (double m)` `[inline]`

Sets the maximum value of the widget.

30.86.3.5 `double FL_Spinner::maximum () const` `[inline]`

Gets the maximum value of the widget.

30.86.3.6 double FL_Spinner::maximum () const [inline]

Speling mistakes retained for source compatibility.

[Deprecated](#)

30.86.3.7 void FL_Spinner::minimum (double *m*) [inline]

Sets the minimum value of the widget.

30.86.3.8 double FL_Spinner::minimum () const [inline]

Gets the minimum value of the widget.

30.86.3.9 double FL_Spinner::mininum () const [inline]

Speling mistakes retained for source compatibility.

[Deprecated](#)

30.86.3.10 void FL_Spinner::range (double *a*, double *b*) [inline]

Sets the minimum and maximum values for the widget.

30.86.3.11 void FL_Spinner::resize (int *X*, int *Y*, int *W*, int *H*) [inline, virtual]

Resizes the [FL_Group](#) widget and all of its children.

The [FL_Group](#) widget first resizes itself, and then it moves and resizes all its children according to the rules documented for [FL_Group::resizable\(FL_Widget*\)](#)

See also:

- [FL_Group::resizable\(FL_Widget*\)](#)
- [FL_Group::resizable\(\)](#)
- [FL_Widget::resize\(int,int,int,int\)](#)

Reimplemented from [FL_Group](#).

30.86.3.12 double FL_Spinner::step () const [inline]

Sets or returns the amount to change the value when the user clicks a button.

Before setting step to a non-integer value, the spinner [type\(\)](#) should be changed to floating point.

30.86.3.13 void FL_Spinner::textcolor (FL_Color c) [inline]

Sets the color of the text in the input field.

30.86.3.14 FL_Color FL_Spinner::textcolor () const [inline]

Gets the color of the text in the input field.

30.86.3.15 void FL_Spinner::textfont (FL_Font f) [inline]

Sets the font of the text in the input field.

30.86.3.16 FL_Font FL_Spinner::textfont () const [inline]

Gets the font of the text in the input field.

30.86.3.17 void FL_Spinner::textsize (FL_Fontsize s) [inline]

Sets the size of the text in the input field.

30.86.3.18 FL_Fontsize FL_Spinner::textsize () const [inline]

Gets the size of the text in the input field.

30.86.3.19 uchar FL_Spinner::type () const [inline]

Sets or Gets the numeric representation in the input field.

Valid values are FL_INT_INPUT and FL_FLOAT_INPUT. The first form also changes the [format\(\)](#) template. Setting a new spinner type via a superclass pointer will not work.

Note:

type is not a virtual function.

Reimplemented from [FL_Widget](#).

30.86.3.20 void FL_Spinner::value (double v) [inline]

Sets the current value of the widget.

Before setting value to a non-integer value, the spinner [type\(\)](#) should be changed to floating point.

30.86.3.21 double FL_Spinner::value () const [inline]

Gets the current value of the widget.

The documentation for this class was generated from the following file:

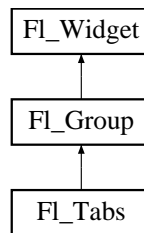
- FL_Spinner.H

30.87 Fl_Tabs Class Reference

The [Fl_Tabs](#) widget is the "file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits.

```
#include <Fl_Tabs.H>
```

Inheritance diagram for Fl_Tabs::



Public Member Functions

- [Fl_Tabs](#) (int, int, int, int, const char *=0)
Creates a new [Fl_Tabs](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- int [push](#) ([Fl_Widget](#) *)
- [Fl_Widget](#) * [push](#) () const
- int [value](#) ([Fl_Widget](#) *)
Sets the widget to become the current visible widget/tab.
- [Fl_Widget](#) * [value](#) ()
Gets the currently visible widget/tab.
- [Fl_Widget](#) * [which](#) (int event_x, int event_y)

Protected Member Functions

- void [draw](#) ()
Draws the widget.
- void [redraw_tabs](#) ()

30.87.1 Detailed Description

The [Fl_Tabs](#) widget is the "file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits.

Figure 30.30: FL_Tabs

Clicking the tab makes a child [visible\(\)](#) by calling [show\(\)](#) on it, and all other children are made invisible by calling [hide\(\)](#) on them. Usually the children are [FL_Group](#) widgets containing several widgets themselves.

Each child makes a card, and its [label\(\)](#) is printed on the card tab, including the label font and style. The selection color of that child is used to color the tab, while the color of the child determines the background color of the pane.

The size of the tabs is controlled by the bounding box of the children (there should be some space between the children and the edge of the [FL_Tabs](#)), and the tabs may be placed "inverted" on the bottom - this is determined by which gap is larger. It is easiest to lay this out in fluid, using the fluid browser to select each child group and resize them until the tabs look the way you want them to.

30.87.2 Constructor & Destructor Documentation

30.87.2.1 [FL_Tabs::FL_Tabs](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [FL_Tabs](#) widget using the given position, size, and label string.

The default boxtype is `FL_THIN_UP_BOX`.

Use `add(FL_Widget)` to add each child, which are usually [FL_Group](#) widgets. The children should be sized to stay away from the top or bottom edge of the [FL_Tabs](#) widget, which is where the tabs will be drawn.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the [FL_Tabs](#) and all of its children can be automatic (local) variables, but you must declare the [FL_Tabs](#) widget *first* so that it is

destroyed last.

30.87.3 Member Function Documentation

30.87.3.1 void Fl_Tabs::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Group](#).

30.87.3.2 int Fl_Tabs::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Group](#).

30.87.3.3 int Fl_Tabs::value (Fl_Widget * newvalue)

Sets the widget to become the current visible widget/tab.

Setting the value hides all other children, and makes this one visible, if it is really a child.

30.87.3.4 `Fl_Widget * Fl_Tabs::value ()`

Gets the currently visible widget/tab.

The `value()` is the first visible child (or the last child if none are visible) and this also hides any other children. This allows the tabs to be deleted, moved to other groups, and `show()/hide()` called without it screwing up.

The documentation for this class was generated from the following files:

- `Fl_Tabs.H`
- `Fl_Tabs.cxx`

30.88 Fl_Text_Buffer Class Reference

The [Fl_Text_Buffer](#) class is used by the [Fl_Text_Display](#) and [Fl_Text_Editor](#) to manage complex text data and is based upon the excellent NEdit text editor engine - see <http://www.nedit.org/>.

```
#include <Fl_Text_Buffer.H>
```

Public Member Functions

- void [add_modify_callback](#) (Fl_Text_Modify_Cb bufModifiedCB, void *cbArg)
Adds a callback function that is called whenever the text buffer is modified.
- void [add_predelete_callback](#) (Fl_Text_Predelete_Cb bufPredeICB, void *cbArg)
Adds a callback routine to be called before text is deleted from the buffer.
- void [append](#) (const char *t)
Appends the text string to the end of the buffer.
- int [appendfile](#) (const char *file, int buflen=128 *1024)
Appends the named file to the end of the buffer.
- void [call_modify_callbacks](#) ()
Calls all modify callbacks that have been registered using the [add_modify_callback\(\)](#) method.
- void [call_predelete_callbacks](#) ()
Calls the stored pre-delete callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- void [canUndo](#) (char flag=1)
Lets the undo system know if we can undo changes.
- char [character](#) (int pos)
Returns the character at the specified position pos in the buffer.
- void [clear_rectangular](#) (int start, int end, int rectStart, int rectEnd)
Clears text in the specified area.
- void [copy](#) (Fl_Text_Buffer *fromBuf, int fromStart, int fromEnd, int toPos)
Copies text from one buffer to this one; fromBuf may be the same as this.
- int [count_displayed_characters](#) (int lineStartPos, int targetPos)
Count the number of displayed characters between buffer position lineStartPos and targetPos.
- int [count_lines](#) (int startPos, int endPos)
Counts the number of newlines between startPos and endPos in buffer.
- int [expand_character](#) (int pos, int indent, char *outStr)
Expands the given character to a displayable format.
- int [findchar_backward](#) (int startPos, char searchChar, int *foundPos)

Search backwards in buffer `buf` for character `searchChar`, starting with the character *BEFORE* `startPos`, returning the result in `foundPos` returns 1 if found, 0 if not.

- `int findchar_forward (int startPos, char searchChar, int *foundPos)`
Finds the next occurrence of the specified character.
- `int findchars_backward (int startPos, const char *searchChars, int *foundPos)`
Finds the previous occurrence of the specified characters.
- `int findchars_forward (int startPos, const char *searchChars, int *foundPos)`
Finds the next occurrence of the specified characters.
- `Fl_Text_Buffer (int requestedSize=0, int preferredGapSize=1024)`
Create an empty text buffer of a pre-determined size.
- `int highlight ()`
Returns the highlighted text.
- `void highlight (int start, int end)`
Highlights the specified text within the buffer.
- `int highlight_position (int *start, int *end, int *isRect, int *rectStart, int *rectEnd)`
Highlights the specified rectangle of text within the buffer.
- `int highlight_position (int *start, int *end)`
Highlights the specified text between `start` and `end` within the buffer.
- `void highlight_rectangular (int start, int end, int rectStart, int rectEnd)`
Highlights a rectangular selection in the buffer.
- `Fl_Text_Selection * highlight_selection ()`
Returns the current highlight selection.
- `char * highlight_text ()`
Returns the highlighted text.
- `void insert (int pos, const char *text)`
Inserts null-terminated string `text` at position `pos`.
- `void insert_column (int column, int startPos, const char *text, int *charsInserted, int *charsDeleted)`
Insert `s` columnwise into buffer starting at displayed character position `column` on the line beginning at `startPos`.
- `int insertfile (const char *file, int pos, int buflen=128 * 1024)`
Inserts a file at the specified position.
- `int length ()`
Returns the number of characters in the buffer.

- `int line_end (int pos)`
Finds and returns the position of the end of the line containing position `pos` (which is either a pointer to the newline character ending the line, or a pointer to one character beyond the end of the buffer).
- `int line_start (int pos)`
Returns the position of the start of the line containing position `pos`.
- `char * line_text (int pos)`
Returns the text from the entire line containing the specified character position.
- `int loadfile (const char *file, int buflen=128 *1024)`
Loads a text file into the buffer.
- `char null_substitution_character ()`
Returns the current nul substitution character.
- `int outputfile (const char *file, int start, int end, int buflen=128 *1024)`
Writes the specified portions of the file to a file.
- `void overlay_rectangular (int startPos, int rectStart, int rectEnd, const char *text, int *charsInserted, int *charsDeleted)`
Overlay text between displayed character positions `rectStart` and `rectEnd` on the line beginning at `startPos`.
- `Fl_Text_Selection * primary_selection ()`
Returns the primary selection.
- `void remove (int start, int end)`
Deletes a range of characters in the buffer.
- `void remove_modify_callback (Fl_Text_Modify_Cb bufModifiedCB, void *cbArg)`
Removes a modify callback.
- `void remove_predelete_callback (Fl_Text_Predelete_Cb predelCB, void *cbArg)`
Removes a callback routine `bufPreDeleteCB` associated with argument `cbArg` to be called before text is deleted from the buffer.
- `void remove_rectangular (int start, int end, int rectStart, int rectEnd)`
Removes a rectangular swath of characters between character positions `start` and `end` and horizontal displayed-character offsets `rectStart` and `rectEnd`.
- `void remove_secondary_selection ()`
Removes the text from the buffer corresponding to the secondary text selection object.
- `void remove_selection ()`
Removes the text in the primary selection.
- `void replace (int start, int end, const char *text)`
Deletes the characters between `start` and `end`, and inserts the null-terminated string `text` in their place in the buffer.

- void [replace_rectangular](#) (int start, int end, int rectStart, int rectEnd, const char *text)
Replaces a rectangular area in the buffer, given by start, end, rectStart, and rectEnd, with text.
- void [replace_secondary_selection](#) (const char *text)
Replaces the text from the buffer corresponding to the secondary text selection object with the new string text.
- void [replace_selection](#) (const char *text)
Replaces the text in the primary selection.
- int [rewind_lines](#) (int startPos, int nLines)
Finds and returns the position of the first character of the line nLines backwards from startPos (not counting the character pointed to by startPos if that is a newline) in the buffer.
- int [savefile](#) (const char *file, int buflen=128 *1024)
Saves a text file from the current buffer.
- int [search_backward](#) (int startPos, const char *searchString, int *foundPos, int matchCase=0)
Search backwards in buffer for string searchString, starting with the character BEFORE startPos, returning the result in foundPos returns 1 if found, 0 if not.
- int [search_forward](#) (int startPos, const char *searchString, int *foundPos, int matchCase=0)
Search forwards in buffer for string searchString, starting with the character startPos, and returning the result in foundPos returns 1 if found, 0 if not.
- void [secondary_select](#) (int start, int end)
Selects a range of characters in the secondary selection.
- void [secondary_select_rectangular](#) (int start, int end, int rectStart, int rectEnd)
Achieves a rectangular selection on the secondary text selection object.
- int [secondary_selected](#) ()
Returns a non 0 value if text has been selected in the secondary text selection, 0 otherwise.
- [Fl_Text_Selection](#) * [secondary_selection](#) ()
Returns the secondary selection.
- int [secondary_selection_position](#) (int *start, int *end, int *isRect, int *rectStart, int *rectEnd)
Returns the current selection in the secondary text selection object.
- int [secondary_selection_position](#) (int *start, int *end)
Returns the current selection in the secondary text selection object.
- char * [secondary_selection_text](#) ()
Returns the text in the secondary selection.
- void [secondary_unselect](#) ()
Clears any selection in the secondary text selection object.
- void [select](#) (int start, int end)

Selects a range of characters in the buffer.

- void [select_rectangular](#) (int start, int end, int rectStart, int rectEnd)
Achieves a rectangular selection on the primary text selection object.
- int [selected](#) ()
Returns a non 0 value if text has been selected, 0 otherwise.
- int [selection_position](#) (int *start, int *end, int *isRect, int *rectStart, int *rectEnd)
Gets the selection position, and rectangular selection info.
- int [selection_position](#) (int *start, int *end)
Gets the selection position.
- char * [selection_text](#) ()
Returns the currently selected text.
- int [skip_displayed_characters](#) (int lineStartPos, int nChars)
Count forward from buffer position startPos in displayed characters (displayed characters are the characters shown on the screen to represent characters in the buffer, where tabs and control characters are expanded).
- int [skip_lines](#) (int startPos, int nLines)
Finds the first character of the line nLines forward from startPos in the buffer and returns its position.
- int [substitute_null_characters](#) (char *string, int length)
The primary routine for integrating new text into a text buffer with substitution of another character for ascii nuls.
- void [tab_distance](#) (int tabDist)
Set the hardware tab distance (width) used by all displays for this buffer, and used in computing offsets for rectangular selection operations.
- int [tab_distance](#) ()
Gets the tab width.
- void [text](#) (const char *text)
Replaces the entire contents of the text buffer.
- char * [text](#) ()
Get the entire contents of a text buffer.
- char * [text_in_rectangle](#) (int start, int end, int rectStart, int rectEnd)
Returns the text from the given rectangle.
- char * [text_range](#) (int start, int end)
Return a copy of the text between start and end character positions from text buffer buf.
- int [undo](#) (int *cp=0)
Undo text modification according to the undo variables or insert text from the undo buffer.

- void [unhighlight](#) ()
Unhighlights text in the buffer.
- void [unselect](#) ()
Cancels any previous selection on the primary text selection object.
- void [unsubstitute_null_characters](#) (char *string)
Converts strings obtained from buffers which contain null characters, which have been substituted for by a special substitution character, back to a null-containing string.
- int [word_end](#) (int pos)
Returns the position corresponding to the end of the word.
- int [word_start](#) (int pos)
Returns the position corresponding to the start of the word.
- [~Fl_Text_Buffer](#) ()
Frees a text buffer.

Static Public Member Functions

- static int [character_width](#) (char c, int indent, int tabDist, char nullSubsChar)
Return the length in displayed characters of character c expanded for display (as discussed above in [expand_character\(\)](#)).
- static int [expand_character](#) (char c, int indent, char *outStr, int tabDist, char nullSubsChar)
Expand a single character c from the text buffer into it's displayable screen representation (which may be several characters for a tab or a control code).

Protected Member Functions

- void [call_modify_callbacks](#) (int pos, int nDeleted, int nInserted, int nRestyled, const char *deletedText)
Calls the stored modify callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- void [call_predelete_callbacks](#) (int pos, int nDeleted)
Calls the stored pre-delete callback procedure(s) for this buffer to update the changed area(s) on the screen and any other listeners.
- int [insert_](#) (int pos, const char *text)
Internal (non-redisplaying) version of BufInsert.
- void [insert_column_](#) (int column, int startPos, const char *insText, int *nDeleted, int *nInserted, int *endPos)
Inserts a column of text without calling the modify callbacks.
- void [move_gap](#) (int pos)

- void [overlay_rectangular_](#) (int startPos, int rectStart, int rectEnd, const char *insText, int *nDeleted, int *nInserted, int *endPos)
Overlay a rectangular area of text without calling the modify callbacks.
- void [reallocate_with_gap](#) (int newGapStart, int newGapLen)
Reallocates the text storage in the buffer to have a gap starting at newGapStart and a gap size of newGapLen, preserving the buffer's current contents.
- void [rectangular_selection_boundaries](#) (int lineStartPos, int rectStart, int rectEnd, int *selStart, int *selEnd)
Finds the first and last character position in a line within a rectangular selection (for copying).
- void [redisplay_selection](#) ([FL_Text_Selection](#) *oldSelection, [FL_Text_Selection](#) *newSelection)
Calls the stored redisplay procedure(s) for this buffer to update the screen for a change in a selection.
- void [remove_](#) (int start, int end)
Internal (non-redisplaying) version of BufRemove.
- void [remove_rectangular_](#) (int start, int end, int rectStart, int rectEnd, int *replaceLen, int *endPos)
Deletes a rectangle of text without calling the modify callbacks.
- void [remove_selection_](#) ([FL_Text_Selection](#) *sel)
Removes the text from the buffer corresponding to sel.
- void [replace_selection_](#) ([FL_Text_Selection](#) *sel, const char *text)
Replaces the text in selection sel.
- char * [selection_text_](#) ([FL_Text_Selection](#) *sel)
- void [update_selections](#) (int pos, int nDeleted, int nInserted)
Updates all of the selections in the buffer for changes in the buffer's text.

Protected Attributes

- char * [mBuf](#)
allocated memory where the text is stored
- char [mCanUndo](#)
if this buffer is used for attributes, it must not do any undo calls
- void ** [mCbArgs](#)
caller arguments for modifyProcs above
- int [mCursorPosHint](#)
hint for reasonable cursor position after a buffer modification operation
- int [mGapEnd](#)
points to the first char after the gap
- int [mGapStart](#)

points to the first character of the gap

- [Fl_Text_Selection mHighlight](#)
highlighted areas
- [int mLength](#)
length of the text in the buffer (the length of the buffer itself must be calculated: gapEnd - gapStart + length)
- [int mNModifyProcs](#)
number of modify-redisplay procs attached
- [Fl_Text_Modify_Cb * mNodifyProcs](#)
< procedures to call when buffer is
- [int mNPredeleteProcs](#)
number of pre-delete procs attached
- [char mNullSubsChar](#)
NEdit is based on C null-terminated strings, so ascii-nul characters must be substituted with something else.
- [void ** mPredeleteCbArgs](#)
caller argument for pre-delete proc above
- [Fl_Text_Predelete_Cb * mPredeleteProcs](#)
< procedure to call before text is deleted
- [int mPreferredGapSize](#)
the default allocation for the text gap is 1024 bytes and should only be increased if frequent and large changes in buffer size are expected
- [Fl_Text_Selection mPrimary](#)
highlighted areas
- [Fl_Text_Selection mSecondary](#)
highlighted areas
- [int mTabDist](#)
equiv.
- [int mUseTabs](#)
True if buffer routines are allowed to use tabs for padding in rectangular operations.

30.88.1 Detailed Description

The [Fl_Text_Buffer](#) class is used by the [Fl_Text_Display](#) and [Fl_Text_Editor](#) to manage complex text data and is based upon the excellent NEdit text editor engine - see <http://www.nedit.org/>.

The [Fl_Text_Buffer](#) class is used by the [Fl_Text_Display](#) and [Fl_Text_Editor](#) to manage complex text data and is based upon the excellent NEdit text editor engine - see <http://www.nedit.org/>.

30.88.2 Constructor & Destructor Documentation

30.88.2.1 Fl_Text_Buffer::Fl_Text_Buffer (int *requestedSize* = 0, int *preferredGapSize* = 1024)

Create an empty text buffer of a pre-determined size.

Parameters:

requestedSize use this to avoid unnecessary re-allocation if you know exactly how much the buffer will need to hold

preferredGapSize Initial size for the buffer gap (empty space in the buffer where text might be inserted if the user is typing sequential chars) Creates a new text buffer of the specified initial size.

30.88.3 Member Function Documentation

30.88.3.1 void Fl_Text_Buffer::add_modify_callback (Fl_Text_Modify_Cb *bufModifiedCB*, void * *cbArg*)

Adds a callback function that is called whenever the text buffer is modified.

The callback function is declared as follows:

```
typedef void (*Fl_Text_Modify_Cb)(int pos, int nInserted, int nDeleted,
                                  int nRestyled, const char* deletedText,
                                  void* cbArg);
```

30.88.3.2 void Fl_Text_Buffer::add_predelete_callback (Fl_Text_Predelete_Cb *bufPreDeleteCB*, void * *cbArg*)

Adds a callback routine to be called before text is deleted from the buffer.

30.88.3.3 void Fl_Text_Buffer::append (const char * *t*) [inline]

Appends the text string to the end of the buffer.

30.88.3.4 int Fl_Text_Buffer::appendfile (const char * *file*, int *buflen* = 128*1024) [inline]

Appends the named file to the end of the buffer.

Returns 0 on success, non-zero on error (strerror() contains reason). 1 indicates open for read failed (no data loaded). 2 indicates error occurred while reading data (data was partially loaded).

30.88.3.5 char Fl_Text_Buffer::character (int *pos*)

Returns the character at the specified position *pos* in the buffer.

Positions start at 0

30.88.3.6 int Fl_Text_Buffer::character_width (char *c*, int *indent*, int *tabDist*, char *nullSubsChar*)
[static]

Return the length in displayed characters of character *c* expanded for display (as discussed above in [expand_character\(\)](#)).

If the buffer for which the character width is being measured is doing null substitution, *nullSubsChar* should be passed as that character (or nul to ignore).

30.88.3.7 void Fl_Text_Buffer::clear_rectangular (int *start*, int *end*, int *rectStart*, int *rectEnd*)

Clears text in the specified area.

It clears a rectangular "hole" out of the buffer between character positions *start* and *end* and horizontal displayed-character offsets *rectStart* and *rectEnd*.

30.88.3.8 int Fl_Text_Buffer::count_displayed_characters (int *lineStartPos*, int *targetPos*)

Count the number of displayed characters between buffer position *lineStartPos* and *targetPos*.

(displayed characters are the characters shown on the screen to represent characters in the buffer, where tabs and control characters are expanded)

30.88.3.9 int Fl_Text_Buffer::count_lines (int *startPos*, int *endPos*)

Counts the number of newlines between *startPos* and *endPos* in buffer.

The character at position *endPos* is not counted.

30.88.3.10 int Fl_Text_Buffer::expand_character (char *c*, int *indent*, char * *outStr*, int *tabDist*, char *nullSubsChar*)
[static]

Expand a single character *c* from the text buffer into it's displayable screen representation (which may be several characters for a tab or a control code).

Returns the number of characters added to *outStr*. *indent* is the number of characters from the start of the line for figuring tabs of length *tabDist*. Output string is guaranteed to be shorter or equal in length to FL_TEXT_MAX_EXP_CHAR_LEN. Tabs and other control characters are given special treatment. *nullSubsChar* represent the null character to be transformed in <nul>

30.88.3.11 int Fl_Text_Buffer::expand_character (int *pos*, int *indent*, char * *outStr*)

Expands the given character to a displayable format.

Tabs and other control characters are given special treatment. Get a character from the text buffer expanded into its screen representation (which may be several characters for a tab or a control code). Returns the number of characters written to *outStr*. *indent* is the number of characters from the start of the line for figuring tabs. Output string is guranteed to be shorter or equal in length to FL_TEXT_MAX_EXP_CHAR_LEN

30.88.3.12 `int Fl_Text_Buffer::findchar_backward (int startPos, char searchChar, int *foundPos)`

Search backwards in buffer `buf` for character `searchChar`, starting with the character BEFORE `startPos`, returning the result in `foundPos` returns 1 if found, 0 if not.

(The difference between this and `BufSearchBackward` is that it's optimized for single characters. The overall performance of the text widget is dependent on its ability to count lines quickly, hence searching for a single character: newline)

30.88.3.13 `int Fl_Text_Buffer::findchar_forward (int startPos, char searchChar, int *foundPos)`

Finds the next occurrence of the specified character.

Search forwards in buffer for character `searchChar`, starting with the character `startPos`, and returning the result in `foundPos` returns 1 if found, 0 if not. (The difference between this and `BufSearchForward` is that it's optimized for single characters. The overall performance of the text widget is dependent on its ability to count lines quickly, hence searching for a single character: newline)

30.88.3.14 `int Fl_Text_Buffer::findchars_backward (int startPos, const char *searchChars, int *foundPos)`

Finds the previous occurrence of the specified characters.

Search backwards in buffer for characters in `searchChars`, starting with the character BEFORE `startPos`, returning the result in `foundPos` returns 1 if found, 0 if not.

30.88.3.15 `int Fl_Text_Buffer::findchars_forward (int startPos, const char *searchChars, int *foundPos)`

Finds the next occurrence of the specified characters.

Search forwards in buffer for characters in `searchChars`, starting with the character `startPos`, and returning the result in `foundPos` returns 1 if found, 0 if not.

30.88.3.16 `int Fl_Text_Buffer::highlight ()` `[inline]`

Returns the highlighted text.

When you are done with the text, free it using the `free()` function.

30.88.3.17 `void Fl_Text_Buffer::highlight (int start, int end)`

Highlights the specified text within the buffer.

30.88.3.18 `int Fl_Text_Buffer::highlight_position (int *start, int *end, int *isRect, int *rectStart, int *rectEnd)`

Highlights the specified rectangle of text within the buffer.

30.88.3.19 `int Fl_Text_Buffer::highlight_position (int *start, int *end)`

Highlights the specified text between `start` and `end` within the buffer.

30.88.3.20 `Fl_Text_Selection* Fl_Text_Buffer::highlight_selection ()` `[inline]`

Returns the current highlight selection.

30.88.3.21 `char * Fl_Text_Buffer::highlight_text ()`

Returns the highlighted text.

When you are done with the text, free it using the `free()` function.

30.88.3.22 `void Fl_Text_Buffer::insert (int pos, const char * text)`

Inserts null-terminated string `text` at position `pos`.

30.88.3.23 `int Fl_Text_Buffer::insert_ (int pos, const char * text)` `[protected]`

Internal (non-redisplaying) version of `BufInsert`.

Returns the length of text inserted (this is just `strlen(text)`), however this calculation can be expensive and the length will be required by any caller who will continue on to call `redisplay`. `pos` must be contiguous with the existing text in the buffer (i.e. not past the end).

30.88.3.24 `void Fl_Text_Buffer::insert_column (int column, int startPos, const char * text, int * charsInserted, int * charsDeleted)`

Insert `s` columnwise into buffer starting at displayed character position `column` on the line beginning at `startPos`.

Opens a rectangular space the width and height of `s`, by moving all text to the right of `column` right. If `charsInserted` and `charsDeleted` are not NULL, the number of characters inserted and deleted in the operation (beginning at `startPos`) are returned in these arguments.

30.88.3.25 `void Fl_Text_Buffer::insert_column_ (int column, int startPos, const char * insText, int * nDeleted, int * nInserted, int * endPos)` `[protected]`

Inserts a column of text without calling the modify callbacks.

Note that in some pathological cases, inserting can actually decrease the size of the buffer because of spaces being coalesced into tabs. `nDeleted` and `nInserted` return the number of characters deleted and inserted beginning at the start of the line containing `startPos`. `endPos` returns buffer position of the lower left edge of the inserted column (as a hint for routines which need to set a cursor position).

30.88.3.26 `int Fl_Text_Buffer::insertfile (const char * file, int pos, int buflen = 128*1024)`

Inserts a file at the specified position.

Returns 0 on success, non-zero on error (`strerror()` contains reason). 1 indicates open for read failed (no data loaded). 2 indicates error occurred while reading data (data was partially loaded).

30.88.3.27 `int Fl_Text_Buffer::length ()` `[inline]`

Returns the number of characters in the buffer.

30.88.3.28 `int Fl_Text_Buffer::line_start (int pos)`

Returns the position of the start of the line containing position `pos`.

30.88.3.29 `char * Fl_Text_Buffer::line_text (int pos)`

Returns the text from the entire line containing the specified character position.

When you are done with the text, free it using the `free()` function.

30.88.3.30 `char Fl_Text_Buffer::null_substitution_character () [inline]`

Returns the current nul substitution character.

30.88.3.31 `int Fl_Text_Buffer::outputfile (const char * file, int start, int end, int buflen = 128*1024)`

Writes the specified portions of the file to a file.

Returns 0 on success, non-zero on error (`strerror()` contains reason). 1 indicates open for write failed (no data saved). 2 indicates error occurred while writing data (data was partially saved).

30.88.3.32 `void Fl_Text_Buffer::overlay_rectangular (int startPos, int rectStart, int rectEnd, const char * text, int * charsInserted, int * charsDeleted)`

Overlay text between displayed character positions `rectStart` and `rectEnd` on the line beginning at `startPos`.

If `charsInserted` and `charsDeleted` are not NULL, the number of characters inserted and deleted in the operation (beginning at `startPos`) are returned in these arguments.

30.88.3.33 `void Fl_Text_Buffer::overlay_rectangular_ (int startPos, int rectStart, int rectEnd, const char * insText, int * nDeleted, int * nInserted, int * endPos) [protected]`

Overlay a rectangular area of text without calling the modify callbacks.

`nDeleted` and `nInserted` return the number of characters deleted and inserted beginning at the start of the line containing `startPos`. `endPos` returns buffer position of the lower left edge of the inserted column (as a hint for routines which need to set a cursor position).

30.88.3.34 `Fl_Text_Selection* Fl_Text_Buffer::primary_selection () [inline]`

Returns the primary selection.

30.88.3.35 `void Fl_Text_Buffer::rectangular_selection_boundaries (int lineStartPos, int rectStart, int rectEnd, int * selStart, int * selEnd) [protected]`

Finds the first and last character position in a line within a rectangular selection (for copying).

Includes tabs which cross `rectStart`, but not control characters which do so. Leaves off tabs which cross `rectEnd`.

Technically, the calling routine should convert tab characters which cross the right boundary of the selection to spaces which line up with the edge of the selection. Unfortunately, the additional memory management required in the parent routine to allow for the changes in string size is not worth all the extra work just for a couple of shifted characters, so if a tab protrudes, just lop it off and hope that there are other characters in the selection to establish the right margin for subsequent columnar pastes of this data.

30.88.3.36 void Fl_Text_Buffer::remove (int start, int end)

Deletes a range of characters in the buffer.

30.88.3.37 void Fl_Text_Buffer::remove_ (int start, int end) [protected]

Internal (non-redisplaying) version of BufRemove.

Removes the contents of the buffer between start and end (and moves the gap to the site of the delete).

30.88.3.38 void Fl_Text_Buffer::remove_modify_callback (Fl_Text_Modify_Cb bufModifiedCB, void * cbArg)

Removes a modify callback.

30.88.3.39 void Fl_Text_Buffer::remove_predelete_callback (Fl_Text_Predelete_Cb bufPreDeleteCB, void * cbArg)

Removes a callback routine bufPreDeleteCB associated with argument cbArg to be called before text is deleted from the buffer.

30.88.3.40 void Fl_Text_Buffer::remove_rectangular_ (int start, int end, int rectStart, int rectEnd, int * replaceLen, int * endPos) [protected]

Deletes a rectangle of text without calling the modify callbacks.

Returns the number of characters replacing those between start and end. Note that in some pathological cases, deleting can actually increase the size of the buffer because of tab expansions. endPos returns the buffer position of the point in the last line where the text was removed (as a hint for routines which need to position the cursor after a delete operation)

30.88.3.41 void Fl_Text_Buffer::remove_secondary_selection ()

Removes the text from the buffer corresponding to the secondary text selection object.

30.88.3.42 void Fl_Text_Buffer::remove_selection ()

Removes the text in the primary selection.

30.88.3.43 void Fl_Text_Buffer::remove_selection_ (Fl_Text_Selection * sel) [protected]

Removes the text from the buffer corresponding to sel.

30.88.3.44 void Fl_Text_Buffer::replace_rectangular (int *start*, int *end*, int *rectStart*, int *rectEnd*, const char * *text*)

Replaces a rectangular area in the buffer, given by *start*, *end*, *rectStart*, and *rectEnd*, with *text*. If *text* is vertically longer than the rectangle, add extra lines to make room for it.

30.88.3.45 void Fl_Text_Buffer::replace_secondary_selection (const char * *text*)

Replaces the text from the buffer corresponding to the secondary text selection object with the new string *text*.

30.88.3.46 void Fl_Text_Buffer::replace_selection (const char * *text*)

Replaces the text in the primary selection.

30.88.3.47 void Fl_Text_Buffer::replace_selection_ (Fl_Text_Selection * *sel*, const char * *text*)
[protected]

Replaces the *text* in selection *sel*.

30.88.3.48 int Fl_Text_Buffer::rewind_lines (int *startPos*, int *nLines*)

Finds and returns the position of the first character of the line *nLines* backwards from *startPos* (not counting the character pointed to by *startpos* if that is a newline) in the buffer.

nLines == 0 means find the beginning of the line

30.88.3.49 void Fl_Text_Buffer::secondary_select (int *start*, int *end*)

Selects a range of characters in the secondary selection.

30.88.3.50 Fl_Text_Selection* Fl_Text_Buffer::secondary_selection () [inline]

Returns the secondary selection.

30.88.3.51 int Fl_Text_Buffer::secondary_selection_position (int * *start*, int * *end*, int * *isRect*, int * *rectStart*, int * *rectEnd*)

Returns the current selection in the secondary text selection object.

30.88.3.52 int Fl_Text_Buffer::secondary_selection_position (int * *start*, int * *end*)

Returns the current selection in the secondary text selection object.

30.88.3.53 `char * Fl_Text_Buffer::secondary_selection_text ()`

Returns the text in the secondary selection.

When you are done with the text, free it using the `free()` function.

30.88.3.54 `void Fl_Text_Buffer::secondary_unselect ()`

Clears any selection in the secondary text selection object.

Cancels any previous selection on the primary text selection object.

30.88.3.55 `void Fl_Text_Buffer::select (int start, int end)`

Selects a range of characters in the buffer.

30.88.3.56 `char * Fl_Text_Buffer::selection_text ()`

Returns the currently selected text.

When you are done with the text, free it using the `free()` function.

30.88.3.57 `int Fl_Text_Buffer::substitute_null_characters (char * string, int len)`

The primary routine for integrating new text into a text buffer with substitution of another character for ascii nuls.

This substitutes null characters in the string in preparation for being copied or replaced into the buffer, and if necessary, adjusts the buffer as well, in the event that the string contains the character it is currently using for substitution. Returns 0, if substitution is no longer possible because all non-printable characters are already in use.

30.88.3.58 `int Fl_Text_Buffer::tab_distance () [inline]`

Gets the tab width.

30.88.3.59 `char * Fl_Text_Buffer::text ()`

Get the entire contents of a text buffer.

Memory is allocated to contain the returned string, which the caller must free.

30.88.3.60 `char * Fl_Text_Buffer::text_in_rectangle (int start, int end, int rectStart, int rectEnd)`

Returns the text from the given rectangle.

When you are done with the text, free it using the `free()` function.

30.88.3.61 `char * Fl_Text_Buffer::text_range (int start, int end)`

Return a copy of the text between `start` and `end` character positions from text buffer `buf`.

Positions start at 0, and the range does not include the character pointed to by `end`. When you are done with the text, free it using the `free()` function.

30.88.3.62 `void Fl_Text_Buffer::unhighlight ()`

Unhighlights text in the buffer.

30.88.3.63 `void Fl_Text_Buffer::unsubstitute_null_characters (char * string)`

Converts strings obtained from buffers which contain null characters, which have been substituted for by a special substitution character, back to a null-containing string.

There is no time penalty for calling this routine if no substitution has been done.

30.88.3.64 `int Fl_Text_Buffer::word_end (int pos)`

Returns the position corresponding to the end of the word.

30.88.4 Member Data Documentation

30.88.4.1 `Fl_Text_Modify_Cb* Fl_Text_Buffer::mNodifyProcs` [protected]

< procedures to call when buffer is
modified to redisplay contents

30.88.4.2 `char Fl_Text_Buffer::mNullSubsChar` [protected]

NEdit is based on C null-terminated strings, so `ascii-nul` characters must be substituted with something else.

This is the else, but of course, things get quite messy when you use it

30.88.4.3 `Fl_Text_Predelete_Cb* Fl_Text_Buffer::mPredeleteProcs` [protected]

< procedure to call before text is deleted
from the buffer; at most one is supported.

30.88.4.4 `int Fl_Text_Buffer::mTabDist` [protected]

equiv.

number of characters in a tab

The documentation for this class was generated from the following files:

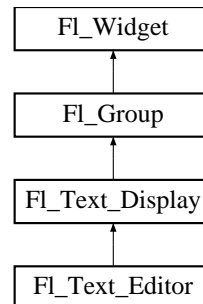
- `Fl_Text_Buffer.H`
- `Fl_Text_Buffer.cxx`

30.89 FL_Text_Display Class Reference

This is the FLTK text display widget.

```
#include <Fl_Text_Display.H>
```

Inheritance diagram for FL_Text_Display::



Classes

- struct [Style_Table_Entry](#)

This structure associates the color,font,size of a string to draw with an attribute mask matching attr.

Public Types

- enum {
NORMAL_CURSOR, CARET_CURSOR, DIM_CURSOR, BLOCK_CURSOR,
HEAVY_CURSOR }
text display cursor shapes enumeration
- enum { **CURSOR_POS, CHARACTER_POS }**
- enum { **DRAG_CHAR = 0, DRAG_WORD = 1, DRAG_LINE = 2 }**
drag types- they match [Fl::event_clicks\(\)](#) so that single clicking to start a collection selects by character, double clicking selects by word and triple clicking selects by line.
- enum { **ATTR_NONE = 0, ATTR_UNDERLINE = 1, ATTR_HIDDEN = 2 }**
style attributes - currently not implemented!
- typedef void(* **Unfinished_Style_Cb**)(int, void *)

Public Member Functions

- [Fl_Text_Buffer](#) * [buffer](#) ()
Gets the current text buffer associated with the text widget.
- void [buffer](#) ([Fl_Text_Buffer](#) &buf)
Sets or gets the current text buffer associated with the text widget.

- void [buffer](#) ([FL_Text_Buffer](#) *buf)
Attach a text buffer to display, replacing the current buffer (if any).
- int [count_lines](#) (int start, int end, bool start_pos_is_line_start)
Same as BufCountLines, but takes in to account wrapping if wrapping is turned on.
- void [cursor_color](#) ([FL_Color](#) n)
Sets or gets the text cursor color.
- [FL_Color](#) [cursor_color](#) () const
Sets or gets the text cursor color.
- void [cursor_style](#) (int style)
Sets the text cursor style to one of the following:.
- [FL_Text_Display](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new text display widget.
- virtual int [handle](#) (int e)
Handles the specified event.
- void [hide_cursor](#) ()
Hides the text cursor.
- void [highlight_data](#) ([FL_Text_Buffer](#) *styleBuffer, const [Style_Table_Entry](#) *styleTable, int nStyles, char unfinishedStyle, Unfinished_Style_Cb unfinishedHighlightCB, void *cbArg)
Attach (or remove) highlight information in text display and redisplay.
- int [in_selection](#) (int x, int y)
Return 1 if position (X, Y) is inside of the primary [FL_Text_Selection](#).
- void [insert](#) (const char *text)
Inserts "text" at the current cursor location.
- int [insert_position](#) ()
Gets the position of the text insertion cursor for text display.
- void [insert_position](#) (int newPos)
Sets the position of the text insertion cursor for text display.
- int [line_end](#) (int pos, bool start_pos_is_line_start)
Same as BufEndOfLine, but takes in to account line breaks when wrapping is turned on.
- int [line_start](#) (int pos)
Same as BufStartOfLine, but returns the character after last wrap point rather than the last newline.
- int [move_down](#) ()
Moves the current insert position down one line.
- int [move_left](#) ()

Moves the current insert position left one character.

- `int move_right ()`
Moves the current insert position right one character.
- `int move_up ()`
Moves the current insert position up one line.
- `void next_word (void)`
Moves the current insert position right one word.
- `void overstrike (const char *text)`
Replaces text at the current insert position.
- `int position_style (int lineStartPos, int lineLen, int lineIndex, int dispIndex)`
Determine the drawing method to use to draw a specific character from "buf".
- `void previous_word (void)`
Moves the current insert position left one word.
- `void redisplay_range (int start, int end)`
Marks text from start to end as needing a redraw.
- `virtual void resize (int X, int Y, int W, int H)`
Change the size of the displayed text area.
- `int rewind_lines (int startPos, int nLines)`
Same as BufCountBackwardNLines, but takes in to account line breaks when wrapping is turned on.
- `void scroll (int topLineNum, int horizOffset)`
Scrolls the current buffer to start at the specified line and column.
- `void scrollbar_align (Fl_Align a)`
Sets the scrollbar alignment type.
- `Fl_Align scrollbar_align ()`
Gets the scrollbar alignment type.
- `void scrollbar_width (int W)`
Sets or gets the width/height of the scrollbars.
- `int scrollbar_width ()`
Sets or gets the width/height of the scrollbars.
- `void shortcut (int s)`
- `int shortcut () const`
- `void show_cursor (int b=1)`
Shows the text cursor.
- `void show_insert_position ()`

Scrolls the text buffer to show the current insert position.

- `int skip_lines (int startPos, int nLines, bool startPosIsLineStart)`
Same as BufCountForwardNLines, but takes in to account line breaks when wrapping is turned on.
- `void textcolor (unsigned n)`
Sets the default color of text in the widget.
- `Fl_Color textcolor () const`
Gets the default color of text in the widget.
- `void textfont (Fl_Font s)`
Sets the default font used when drawing text in the widget.
- `Fl_Font textfont () const`
Gets the default font used when drawing text in the widget.
- `void textsize (Fl_Fonsize s)`
Sets the default size of text in the widget.
- `Fl_Fonsize textsize () const`
Gets the default size of text in the widget.
- `int word_end (int pos)`
Moves the insert position to the end of the current word.
- `int word_start (int pos)`
Moves the insert position to the beginning of the current word.
- `void wrap_mode (int wrap, int wrap_margin)`
If mode is not zero, this call enables automatic word wrapping at column pos.
- `int wrapped_column (int row, int column)`
Correct a column number based on an unconstrained position (as returned by TextDXYToUnconstrainedPosition) to be relative to the last actual newline in the buffer before the row and column position given, rather than the last line start created by line wrapping.
- `int wrapped_row (int row)`
Correct a row number from an unconstrained position (as returned by TextDXYToUnconstrainedPosition) to a straight number of newlines from the top line of the display.
- `~Fl_Text_Display ()`
Free a text display and release its associated memory.

Protected Member Functions

- `void absolute_top_line_number (int oldFirstChar)`
Re-calculate absolute top line number for a change in scroll position.

- void [calc_last_char](#) ()
Given a [FL_Text_Display](#) with a complete, up-to-date `lineStarts` array, update the `lastChar` entry to point to the last buffer position displayed.
- void [calc_line_starts](#) (int startLine, int endLine)
Scan through the text in the `"textD"`'s buffer and recalculate the line starts array values beginning at index `"startLine"` and continuing through (including) `"endLine"`.
- void [clear_rect](#) (int style, int x, int y, int width, int height)
Clear a rectangle with the appropriate background color for `"style"`.
- void [display_insert](#) ()
Scroll the display to bring insertion cursor into view.
- virtual void [draw](#) ()
Draws the widget.
- void [draw_cursor](#) (int, int)
Draw a cursor with top center at `X`, `y`.
- void [draw_line_numbers](#) (bool clearAll)
Refresh the line number area.
- void [draw_range](#) (int start, int end)
Refresh all of the text between buffer positions `"start"` and `"end"` not including the character at the position `"end"`.
- void [draw_string](#) (int style, int x, int y, int toX, const char *string, int nChars)
Draw a string or blank area according to parameter `"style"`, using the appropriate colors and drawing method for that style, with top left corner at `X`, `y`.
- void [draw_text](#) (int X, int Y, int W, int H)
Refresh a rectangle of the text display.
- void [draw_vline](#) (int visLineNum, int leftClip, int rightClip, int leftCharIndex, int rightCharIndex)
Draw the text on a single line represented by `"visLineNum"` (the number of lines down from the top of the display), limited by `"leftClip"` and `"rightClip"` window coordinates and `"leftCharIndex"` and `"rightCharIndex"` character positions (not including the character at position `"rightCharIndex"`).
- int [empty_vlines](#) ()
Return true if there are lines visible with no corresponding buffer text.
- void [extend_range_for_styles](#) (int *start, int *end)
*Extend the range of a redraw request (from `*start` to `*end`) with additional redraw requests resulting from changes to the attached style buffer (which contains auxiliary information for coloring or styling text).*
- void [find_line_end](#) (int pos, bool start_pos_is_line_start, int *lineEnd, int *nextLineStart)
Finds both the end of the current line and the start of the next line.
- void [find_wrap_range](#) (const char *deletedText, int pos, int nInserted, int nDeleted, int *modRangeStart, int *modRangeEnd, int *linesInserted, int *linesDeleted)

When continuous wrap is on, and the user inserts or deletes characters, wrapping can happen before and beyond the changed position.

- `int get_absolute_top_line_number ()`
Returns the absolute (non-wrapped) line number of the first line displayed.
- `int longest_vline ()`
- `void maintain_absolute_top_line_number (int state)`
In continuous wrap mode, internal line numbers are calculated after wrapping.
- `int maintaining_absolute_top_line_number ()`
Return true if a separate absolute top line number is being maintained (for displaying line numbers or showing in the statistics line).
- `void measure_deleted_lines (int pos, int nDeleted)`
This is a stripped-down version of the findWrapRange() function above, intended to be used to calculate the number of "deleted" lines during a buffer modification.
- `int measure_proportional_character (char c, int colNum, int pos)`
Measure the width in pixels of a character "c" at a particular column "colNum" and buffer position "pos".
- `int measure_vline (int visLineNum)`
Return the width in pixels of the displayed line pointed to by "visLineNum".
- `void offset_line_starts (int newTopLineNum)`
Offset the line starts array, mTopLineNum, mFirstChar and lastChar, for a new vertical scroll position given by newTopLineNum.
- `int position_to_line (int pos, int *lineNum)`
Find the line number of position "pos" relative to the first line of displayed text.
- `int position_to_linecol (int pos, int *lineNum, int *column)`
Find the line number of position "pos".
- `int position_to_xy (int pos, int *x, int *y)`
Translate a buffer text position to the XY location where the top left of the cursor would be positioned to point to that character.
- `int range_touces_selection (Fl_Text_Selection *sel, int rangeStart, int rangeEnd)`
Return true if the selection "sel" is rectangular, and touches a buffer position withing "rangeStart" to "rangeEnd".
- `void reset_absolute_top_line_number ()`
Count lines from the beginning of the buffer to reestablish the absolute (non-wrapped) top line number.
- `void scroll_ (int topLineNum, int horizOffset)`
- `int string_width (const char *string, int length, int style)`
Find the width of a string in the font of a particular style.
- `void update_h_scrollbar ()`
Update the minimum, maximum, slider size, page increment, and value for the horizontal scroll bar.

- void [update_line_starts](#) (int pos, int charsInserted, int charsDeleted, int linesInserted, int linesDeleted, int *scrolled)
Update the line starts array, mTopLineNum, mFirstChar and lastChar for text display "textD" after a modification to the text buffer, given by the position where the change began "pos", and the nmubers of characters and lines inserted and deleted.
- void [update_v_scrollbar](#) ()
Update the minimum, maximum, slider size, page increment, and value for vertical scroll bar.
- int [vline_length](#) (int visLineNum)
Return the length of a line (number of displayable characters) by examining entries in the line starts array rather than by scanning for newlines.
- int [wrap_uses_character](#) (int lineEndPos)
Line breaks in continuous wrap mode usually happen at newlines or whitespace.
- void [wrapped_line_counter](#) (Fl_Text_Buffer *buf, int startPos, int maxPos, int maxLines, bool startPosIsLineStart, int styleBufOffset, int *retPos, int *retLines, int *retLineStart, int *retLineEnd, bool countLastLineMissingNewLine=true)
Count forward from startPos to either maxPos or maxLines (whichever is reached first), and return all relevant positions and line count.
- int [xy_to_position](#) (int x, int y, int PosType=CHARACTER_POS)
Translate window coordinates to the nearest (insert cursor or character cell) text position.
- void [xy_to_rowcol](#) (int x, int y, int *row, int *column, int PosType=CHARACTER_POS)
Translate window coordinates to the nearest row and column number for positioning the cursor.

Static Protected Member Functions

- static void [buffer_modified_cb](#) (int pos, int nInserted, int nDeleted, int nRestyled, const char *deletedText, void *cbArg)
Callback attached to the text buffer to receive modification information.
- static void [buffer_predelete_cb](#) (int pos, int nDeleted, void *cbArg)
Callback attached to the text buffer to receive delete information before the modifications are actually made.
- static void [h_scrollbar_cb](#) (Fl_Scrollbar *w, Fl_Text_Display *d)
- static void [scroll_timer_cb](#) (void *)
- static void [v_scrollbar_cb](#) (Fl_Scrollbar *w, Fl_Text_Display *d)
Callbacks for drag or valueChanged on scroll bars.

Friends

- void [fl_text_drag_me](#) (int pos, Fl_Text_Display *d)

30.89.1 Detailed Description

This is the FLTK text display widget.

It allows the user to view multiple lines of text and supports highlighting and scrolling. The buffer that is displayed in the widget is managed by the [FL_Text_Buffer](#) class.

30.89.2 Constructor & Destructor Documentation

30.89.2.1 FL_Text_Display::FL_Text_Display (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new text display widget.

30.89.2.2 FL_Text_Display::~~FL_Text_Display ()

Free a text display and release its associated memory.

Note, the text BUFFER that the text display displays is a separate entity and is not freed, nor are the style buffer or style table.

30.89.3 Member Function Documentation

30.89.3.1 FL_Text_Buffer* FL_Text_Display::buffer () [inline]

Gets the current text buffer associated with the text widget.

Multiple text widgets can be associated with the same text buffer.

30.89.3.2 void FL_Text_Display::buffer (FL_Text_Buffer & *buf*) [inline]

Sets or gets the current text buffer associated with the text widget.

Multiple text widgets can be associated with the same text buffer.

30.89.3.3 void FL_Text_Display::calc_line_starts (int *startLine*, int *endLine*) [protected]

Scan through the text in the "textD"'s buffer and recalculate the line starts array values beginning at index "startLine" and continuing through (including) "endLine".

It assumes that the line starts entry preceding "startLine" (or mFirstChar if startLine is 0) is good, and re-counts newlines to fill in the requested entries. Out of range values for "startLine" and "endLine" are acceptable.

30.89.3.4 int FL_Text_Display::count_lines (int *startPos*, int *endPos*, bool *startPosIsLineStart*)

Same as BufCountLines, but takes in to account wrapping if wrapping is turned on.

If the caller knows that startPos is at a line start, it can pass "startPosIsLineStart" as True to make the call more efficient by avoiding the additional step of scanning back to the last newline.

30.89.3.5 void Fl_Text_Display::cursor_color (Fl_Color *n*) [inline]

Sets or gets the text cursor color.

30.89.3.6 Fl_Color Fl_Text_Display::cursor_color () const [inline]

Sets or gets the text cursor color.

30.89.3.7 void Fl_Text_Display::cursor_style (int *style*)

Sets the text cursor style to one of the following:

- Fl_Text_Display::NORMAL_CURSOR - Shows an I beam.
- Fl_Text_Display::CARET_CURSOR - Shows a caret under the text.
- Fl_Text_Display::DIM_CURSOR - Shows a dimmed I beam.
- Fl_Text_Display::BLOCK_CURSOR - Shows an unfilled box around the current character.
- Fl_Text_Display::HEAVY_CURSOR - Shows a thick I beam.

30.89.3.8 void Fl_Text_Display::display_insert () [protected]

Scroll the display to bring insertion cursor into view.

Note: it would be nice to be able to do this without counting lines twice (scroll_() counts them too) and/or to count from the most efficient starting point, but the efficiency of this routine is not as important to the overall performance of the text display.

30.89.3.9 void Fl_Text_Display::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to Fl_Widget* and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Group](#).

30.89.3.10 void Fl_Text_Display::draw_line_numbers (bool *clearAll*) [protected]

Refresh the line number area.

If clearAll is False, writes only over the character cell areas. Setting clearAll to True will clear out any stray marks outside of the character cell area, which might have been left from before a resize or font change.

30.89.3.11 void Fl_Text_Display::draw_range (int *startpos*, int *endpos*) [protected]

Refresh all of the text between buffer positions "start" and "end" not including the character at the position "end".

If end points beyond the end of the buffer, refresh the whole display after pos, including blank lines which are not technically part of any range of characters.

30.89.3.12 void Fl_Text_Display::draw_string (int *style*, int *X*, int *Y*, int *toX*, const char * *string*, int *nChars*) [protected]

Draw a string or blank area according to parameter "style", using the appropriate colors and drawing method for that style, with top left corner at X, y.

If style says to draw text, use "string" as source of characters, and draw "nChars", if style is FILL, erase rectangle where text would have drawn from X to toX and from Y to the maximum Y extent of the current font(s).

30.89.3.13 void Fl_Text_Display::draw_text (int *left*, int *top*, int *width*, int *height*) [protected]

Refresh a rectangle of the text display.

left and top are in coordinates of the text drawing window

30.89.3.14 void Fl_Text_Display::find_line_end (int *startPos*, bool *startPosIsLineStart*, int * *lineEnd*, int * *nextLineStart*) [protected]

Finds both the end of the current line and the start of the next line.

Why? In continuous wrap mode, if you need to know both, figuring out one from the other can be expensive or error prone. The problem comes when there's a trailing space or tab just before the end of the buffer. To translate an end of line value to or from the next lines start value, you need to know whether the trailing space or tab is being used as a line break or just a normal character, and to find that out would otherwise require counting all the way back to the beginning of the line.

30.89.3.15 void Fl_Text_Display::find_wrap_range (const char * *deletedText*, int *pos*, int *nInserted*, int *nDeleted*, int * *modRangeStart*, int * *modRangeEnd*, int * *linesInserted*, int * *linesDeleted*) [protected]

When continuous wrap is on, and the user inserts or deletes characters, wrapping can happen before and beyond the changed position.

This routine finds the extent of the changes, and counts the deleted and inserted lines over that range. It also attempts to minimize the size of the range to what has to be counted and re-displayed, so the results can be useful both for delimiting where the line starts need to be recalculated, and for deciding what part of the text to redisplay.

30.89.3.16 int Fl_Text_Display::get_absolute_top_line_number () [protected]

Returns the absolute (non-wrapped) line number of the first line displayed.

Returns 0 if the absolute top line number is not being maintained.

30.89.3.17 `int Fl_Text_Display::handle (int event)` [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Group](#).

Reimplemented in [Fl_Text_Editor](#).

30.89.3.18 `void Fl_Text_Display::highlight_data (Fl_Text_Buffer * styleBuffer, const Style_Table_Entry * styleTable, int nStyles, char unfinishedStyle, Unfinished_Style_Cb unfinishedHighlightCB, void * cbArg)`

Attach (or remove) highlight information in text display and redisplay.

Highlighting information consists of a style buffer which parallels the normal text buffer, but codes font and color information for the display; a style table which translates style buffer codes (indexed by buffer character - 'A') into fonts and colors; and a callback mechanism for as-needed highlighting, triggered by a style buffer entry of "unfinishedStyle". Style buffer can trigger additional redisplay during a normal buffer modification if the buffer contains a primary [Fl_Text_Selection](#) (see [extendRangeForStyleMods](#) for more information on this protocol).

Style buffers, tables and their associated memory are managed by the caller.

30.89.3.19 `void Fl_Text_Display::insert (const char * text)`

Inserts "text" at the current cursor location.

This has the same effect as inserting the text into the buffer using [BufInsert](#) and then moving the insert position after the newly inserted text, except that it's optimized to do less redrawing.

30.89.3.20 `int Fl_Text_Display::line_end (int pos, bool startPosIsLineStart)`

Same as [BufEndOfLine](#), but takes in to account line breaks when wrapping is turned on.

If the caller knows that `startPos` is at a line start, it can pass `"startPosIsLineStart"` as `True` to make the call more efficient by avoiding the additional step of scanning back to the last newline.

Note that the definition of the end of a line is less clear when continuous wrap is on. With continuous wrap off, it's just a pointer to the newline that ends the line. When it's on, it's the character beyond the last DISPLAYABLE character on the line, where a whitespace character which has been "converted" to a newline for wrapping is not considered displayable. Also note that, a line can be wrapped at a non-whitespace character if the line had no whitespace. In this case, this routine returns a pointer to the start of the next line. This is also consistent with the model used by `visLineLength`.

30.89.3.21 void Fl_Text_Display::maintain_absolute_top_line_number (int state) [protected]

In continuous wrap mode, internal line numbers are calculated after wrapping.

A separate non-wrapped line count is maintained when line numbering is turned on. There is some performance cost to maintaining this line count, so normally absolute line numbers are not tracked if line numbering is off. This routine allows callers to specify that they still want this line count maintained (for use via `TextDPosToLineAndCol`). More specifically, this allows the line number reported in the statistics line to be calibrated in absolute lines, rather than post-wrapped lines.

30.89.3.22 void Fl_Text_Display::measure_deleted_lines (int pos, int nDeleted) [protected]

This is a stripped-down version of the `findWrapRange()` function above, intended to be used to calculate the number of "deleted" lines during a buffer modification.

It is called `_before_` the modification takes place.

This function should only be called in continuous wrap mode with a non-fixed font width. In that case, it is impossible to calculate the number of deleted lines, because the necessary style information is no longer available `_after_` the modification. In other cases, we can still perform the calculation afterwards (possibly even more efficiently).

30.89.3.23 int Fl_Text_Display::measure_proportional_character (char c, int colNum, int pos) [protected]

Measure the width in pixels of a character `"c"` at a particular column `"colNum"` and buffer position `"pos"`.

This is for measuring characters in proportional or mixed-width highlighting fonts.

A note about proportional and mixed-width fonts: the mixed width and proportional font code in `nedit` does not get much use in general editing, because `nedit` doesn't allow per-language-mode fonts, and editing programs in a proportional font is usually a bad idea, so very few users would choose a proportional font as a default. There are still probably mixed-width syntax highlighting cases where things don't redraw properly for insertion/deletion, though static display and wrapping and resizing should now be solid because they are now used for online help display.

30.89.3.24 int Fl_Text_Display::move_down ()

Moves the current insert position down one line.

30.89.3.25 int Fl_Text_Display::move_left ()

Moves the current insert position left one character.

30.89.3.26 int Fl_Text_Display::move_right ()

Moves the current insert position right one character.

30.89.3.27 int Fl_Text_Display::move_up ()

Moves the current insert position up one line.

30.89.3.28 void Fl_Text_Display::next_word (void)

Moves the current insert position right one word.

30.89.3.29 void Fl_Text_Display::offset_line_starts (int *newTopLineNum*) [protected]

Offset the line starts array, mTopLineNum, mFirstChar and lastChar, for a new vertical scroll position given by newTopLineNum.

If any currently displayed lines will still be visible, salvage the line starts values, otherwise, count lines from the nearest known line start (start or end of buffer, or the closest value in the mLineStarts array)

30.89.3.30 void Fl_Text_Display::overstrike (const char * *text*)

Replaces text at the current insert position.

30.89.3.31 int Fl_Text_Display::position_style (int *lineStartPos*, int *lineLen*, int *lineIndex*, int *dispIndex*)

Determine the drawing method to use to draw a specific character from "buf".

"lineStartPos" gives the character index where the line begins, "lineIndex", the number of characters past the beginning of the line, and "dispIndex", the number of displayed characters past the beginning of the line. Passing lineStartPos of -1 returns the drawing style for "no text".

Why not just: position_style(pos)? Because style applies to blank areas of the window beyond the text boundaries, and because this routine must also decide whether a position is inside of a rectangular [Fl_Text_Selection](#), and do so efficiently, without re-counting character positions from the start of the line.

Note that style is a somewhat incorrect name, drawing method would be more appropriate.

30.89.3.32 int Fl_Text_Display::position_to_line (int *pos*, int * *lineNum*) [protected]

Find the line number of position "pos" relative to the first line of displayed text.

Returns 0 if the line is not displayed.

30.89.3.33 int Fl_Text_Display::position_to_linecol (int *pos*, int * *lineNum*, int * *column*)
[protected]

Find the line number of position "pos".

Note: this only works for displayed lines. If the line is not displayed, the function returns 0 (without the mLineStarts array it could turn in to very long calculation involving scanning large amounts of text in the

buffer). If continuous wrap mode is on, returns the absolute line number (as opposed to the wrapped line number which is used for scrolling).

30.89.3.34 `int Fl_Text_Display::position_to_xy (int pos, int * X, int * Y)` [protected]

Translate a buffer text position to the XY location where the top left of the cursor would be positioned to point to that character.

Returns 0 if the position is not displayed because it is VERTICALLY out of view. If the position is horizontally out of view, returns the X coordinate where the position would be if it were visible.

30.89.3.35 `void Fl_Text_Display::previous_word (void)`

Moves the current insert position left one word.

30.89.3.36 `void Fl_Text_Display::redisplay_range (int startpos, int endpos)`

Marks text from start to end as needing a redraw.

30.89.3.37 `void Fl_Text_Display::reset_absolute_top_line_number ()` [protected]

Count lines from the beginning of the buffer to reestablish the absolute (non-wrapped) top line number.

If mode is not continuous wrap, or the number is not being maintained, does nothing.

30.89.3.38 `void Fl_Text_Display::scroll (int topLineNum, int horizOffset)`

Scrolls the current buffer to start at the specified line and column.

30.89.3.39 `void Fl_Text_Display::scrollbar_width (int W)` [inline]

Sets or gets the width/height of the scrollbars.

30.89.3.40 `int Fl_Text_Display::scrollbar_width ()` [inline]

Sets or gets the width/height of the scrollbars.

30.89.3.41 `void Fl_Text_Display::shortcut (int s)` [inline]

Todo

FIXME : get set methods pointing on `shortcut_` have no effects as `shortcut_` is unused in this class and derived!

30.89.3.42 `int Fl_Text_Display::shortcut () const` [inline]

Todo

FIXME : get set methods pointing on `shortcut_` have no effects as `shortcut_` is unused in this class and derived!

30.89.3.43 void Fl_Text_Display::show_insert_position ()

Scrolls the text buffer to show the current insert position.

30.89.3.44 int Fl_Text_Display::skip_lines (int startPos, int nLines, bool startPosIsLineStart)

Same as BufCountForwardNLines, but takes in to account line breaks when wrapping is turned on.

If the caller knows that startPos is at a line start, it can pass "startPosIsLineStart" as True to make the call more efficient by avoiding the additional step of scanning back to the last newline.

30.89.3.45 void Fl_Text_Display::textcolor (unsigned n) [inline]

Sets the default color of text in the widget.

30.89.3.46 Fl_Color Fl_Text_Display::textcolor () const [inline]

Gets the default color of text in the widget.

30.89.3.47 void Fl_Text_Display::textfont (Fl_Font s) [inline]

Sets the default font used when drawing text in the widget.

30.89.3.48 Fl_Font Fl_Text_Display::textfont () const [inline]

Gets the default font used when drawing text in the widget.

30.89.3.49 void Fl_Text_Display::textsize (Fl_Fontsize s) [inline]

Sets the default size of text in the widget.

30.89.3.50 Fl_Fontsize Fl_Text_Display::textsize () const [inline]

Gets the default size of text in the widget.

30.89.3.51 int Fl_Text_Display::word_end (int pos) [inline]

Moves the insert position to the end of the current word.

30.89.3.52 int Fl_Text_Display::word_start (int pos) [inline]

Moves the insert position to the beginning of the current word.

30.89.3.53 void Fl_Text_Display::wrap_mode (int wrap, int wrapMargin)

If *mode* is not zero, this call enables automatic word wrapping at column *pos*.

Word-wrapping does not change the text buffer itself, only the way that the text is displayed.

30.89.3.54 `int Fl_Text_Display::wrap_uses_character (int lineEndPos)` [protected]

Line breaks in continuous wrap mode usually happen at newlines or whitespace.

This line-terminating character is not included in line width measurements and has a special status as a non-visible character. However, lines with no whitespace are wrapped without the benefit of a line terminating character, and this distinction causes endless trouble with all of the text display code which was originally written without continuous wrap mode and always expects to wrap at a newline character.

Given the position of the end of the line, as returned by `TextDEndOfLine` or `BufEndOfLine`, this returns true if there is a line terminating character, and false if there's not. On the last character in the buffer, this function can't tell for certain whether a trailing space was used as a wrap point, and just guesses that it wasn't. So if an exact accounting is necessary, don't use this function.

30.89.3.55 `int Fl_Text_Display::wrapped_column (int row, int column)`

Correct a column number based on an unconstrained position (as returned by `TextDXYToUnconstrainedPosition`) to be relative to the last actual newline in the buffer before the row and column position given, rather than the last line start created by line wrapping.

This is an adapter for rectangular selections and code written before continuous wrap mode, which thinks that the unconstrained column is the number of characters from the last newline. Obviously this is time consuming, because it involves character re-counting.

30.89.3.56 `void Fl_Text_Display::wrapped_line_counter (Fl_Text_Buffer * buf, int startPos, int maxPos, int maxLines, bool startPosIsLineStart, int styleBufOffset, int * retPos, int * retLines, int * retLineStart, int * retLineEnd, bool countLastLineMissingNewLine = true)` [protected]

Count forward from `startPos` to either `maxPos` or `maxLines` (whichever is reached first), and return all relevant positions and line count.

The provided `textBuffer` may differ from the actual text buffer of the widget. In that case it must be a (partial) copy of the actual text buffer and the `styleBufOffset` argument must indicate the starting position of the copy, to take into account the correct style information.

Returned values:

`retPos`: Position where counting ended. When counting lines, the position returned is the start of the line "maxLines" lines beyond "startPos". `retLines`: Number of line breaks counted `retLineStart`: Start of the line where counting ended `retLineEnd`: End position of the last line traversed

30.89.3.57 `int Fl_Text_Display::wrapped_row (int row)`

Correct a row number from an unconstrained position (as returned by `TextDXYToUnconstrainedPosition`) to a straight number of newlines from the top line of the display.

Because rectangular selections are based on newlines, rather than display wrapping, and anywhere a rectangular selection needs a row, it needs it in terms of un-wrapped lines.

30.89.3.58 `int Fl_Text_Display::xy_to_position (int X, int Y, int posType = CHARACTER_POS)` [protected]

Translate window coordinates to the nearest (insert cursor or character cell) text position.

The parameter `posType` specifies how to interpret the position: `CURSOR_POS` means translate the coordinates to the nearest cursor position, and `CHARACTER_POS` means return the position of the character closest to (X, Y).

30.89.3.59 `void Fl_Text_Display::xy_to_rowcol (int X, int Y, int *row, int *column, int posType = CHARACTER_POS)` [protected]

Translate window coordinates to the nearest row and column number for positioning the cursor.

This, of course, makes no sense when the font is proportional, since there are no absolute columns. The parameter `posType` specifies how to interpret the position: `CURSOR_POS` means translate the coordinates to the nearest position between characters, and `CHARACTER_POS` means translate the position to the nearest character cell.

The documentation for this class was generated from the following files:

- `Fl_Text_Display.H`
- `Fl_Text_Display.cxx`

30.90 Fl_Text_Display::Style_Table_Entry Struct Reference

This structure associates the color,font,size of a string to draw with an attribute mask matching attr.

```
#include <Fl_Text_Display.H>
```

Public Attributes

- unsigned **attr**
- [Fl_Color](#) **color**
- [Fl_Font](#) **font**
- int **size**

30.90.1 Detailed Description

This structure associates the color,font,size of a string to draw with an attribute mask matching attr.

The documentation for this struct was generated from the following file:

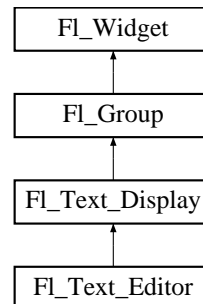
- Fl_Text_Display.H

30.91 FL_Text_Editor Class Reference

This is the FLTK text editor widget.

```
#include <Fl_Text_Editor.H>
```

Inheritance diagram for FL_Text_Editor::



Classes

- struct [Key_Binding](#)
Simple linked list associating a key/state to a function.

Public Types

- typedef int(* [Key_Func](#))(int key, [FL_Text_Editor](#) *editor)
Key function binding callback type.

Public Member Functions

- void [add_default_key_bindings](#) ([Key_Binding](#) **list)
Adds all of the default editor key bindings to the specified key binding list.
- void [add_key_binding](#) (int key, int state, [Key_Func](#) f)
Adds a key of state "state" with the function "function".
- void [add_key_binding](#) (int key, int state, [Key_Func](#) f, [Key_Binding](#) **list)
Adds a key of state "state" with the function "function".
- [Key_Func](#) [bound_key_function](#) (int key, int state)
Returns the function associated with a key binding.
- [Key_Func](#) [bound_key_function](#) (int key, int state, [Key_Binding](#) *list)
Returns the function associated with a key binding.
- void [default_key_function](#) ([Key_Func](#) f)
Sets the default key function for unassigned keys.

- [Fl_Text_Editor](#) (int X, int Y, int W, int H, const char *l=0)
The constructor creates a new text editor widget.
- virtual int [handle](#) (int e)
Handles the specified event.
- int [insert_mode](#) ()
Gets the current insert mode; if non-zero, new text is inserted before the current cursor position.
- void [insert_mode](#) (int b)
Sets the current insert mode; if non-zero, new text is inserted before the current cursor position.
- void [remove_all_key_bindings](#) ()
Removes all of the key bindings associated with the text editor or list.
- void [remove_all_key_bindings](#) ([Key_Binding](#) **list)
Removes all of the key bindings associated with the text editor or list.
- void [remove_key_binding](#) (int key, int state)
Removes the key binding associated with the key "key" of state "state".
- void [remove_key_binding](#) (int key, int state, [Key_Binding](#) **list)
Removes the key binding associated with the key "key" of state "state".

Static Public Member Functions

- static int [kf_backspace](#) (int c, [Fl_Text_Editor](#) *e)
Does a backspace in the current buffer.
- static int [kf_c_s_move](#) (int c, [Fl_Text_Editor](#) *e)
Extends the current selection in the direction indicated by control key c.
- static int [kf_copy](#) (int c, [Fl_Text_Editor](#) *e)
Does a copy of selected text or the current character in the current buffer.
- static int [kf_ctrl_move](#) (int c, [Fl_Text_Editor](#) *e)
Moves the current text cursor in the direction indicated by control key.
- static int [kf_cut](#) (int c, [Fl_Text_Editor](#) *e)
Does a cut of selected text in the current buffer.
- static int [kf_default](#) (int c, [Fl_Text_Editor](#) *e)
Inserts the text associated with the key.
- static int [kf_delete](#) (int c, [Fl_Text_Editor](#) *e)
Does a delete of selected text or the current character in the current buffer.

- static int [kf_down](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor one line down.
- static int [kf_end](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor to the end of the current line.
- static int [kf_enter](#) (int c, [FL_Text_Editor](#) *e)
Inserts a newline at the current cursor position.
- static int [kf_home](#) (int, [FL_Text_Editor](#) *e)
Moves the text cursor to the beginning of the current line.
- static int [kf_ignore](#) (int c, [FL_Text_Editor](#) *e)
Ignores the keypress.
- static int [kf_insert](#) (int c, [FL_Text_Editor](#) *e)
Toggles the insert mode in the text editor.
- static int [kf_left](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor one character to the left.
- static int [kf_move](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor in the direction indicated by key c.
- static int [kf_page_down](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor down one page.
- static int [kf_page_up](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor up one page.
- static int [kf_paste](#) (int c, [FL_Text_Editor](#) *e)
Does a paste of selected text in the current buffer.
- static int [kf_right](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor one character to the right.
- static int [kf_select_all](#) (int c, [FL_Text_Editor](#) *e)
Selects all text in the current buffer.
- static int [kf_shift_move](#) (int c, [FL_Text_Editor](#) *e)
Extends the current selection in the direction of key c.
- static int [kf_undo](#) (int c, [FL_Text_Editor](#) *e)
Undo last edit in the current buffer.
- static int [kf_up](#) (int c, [FL_Text_Editor](#) *e)
Moves the text cursor one line up.

Protected Member Functions

- `int handle_key ()`
Handles a key press in the editor.
- `void maybe_do_callback ()`
does or does not a callback according to `changed()` and `when()` settings

30.91.1 Detailed Description

This is the FLTK text editor widget.

It allows the user to edit multiple lines of text and supports highlighting and scrolling. The buffer that is displayed in the widget is managed by the [FL_Text_Buffer](#) class.

30.91.2 Constructor & Destructor Documentation

30.91.2.1 `FL_Text_Editor::FL_Text_Editor (int X, int Y, int W, int H, const char *l = 0)`

The constructor creates a new text editor widget.

30.91.3 Member Function Documentation

30.91.3.1 `void FL_Text_Editor::add_default_key_bindings (Key_Binding ** list)`

Adds all of the default editor key bindings to the specified key binding list.

30.91.3.2 `Key_Func FL_Text_Editor::bound_key_function (int key, int state) [inline]`

Returns the function associated with a key binding.

30.91.3.3 `FL_Text_Editor::Key_Func FL_Text_Editor::bound_key_function (int key, int state, Key_Binding * list)`

Returns the function associated with a key binding.

30.91.3.4 `void FL_Text_Editor::default_key_function (Key_Func f) [inline]`

Sets the default key function for unassigned keys.

30.91.3.5 `int FL_Text_Editor::handle (int event) [virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Text_Display](#).

30.91.3.6 int Fl_Text_Editor::insert_mode () [inline]

Gets the current insert mode; if non-zero, new text is inserted before the current cursor position. Otherwise, new text replaces text at the current cursor position.

30.91.3.7 void Fl_Text_Editor::insert_mode (int *b*) [inline]

Sets the current insert mode; if non-zero, new text is inserted before the current cursor position. Otherwise, new text replaces text at the current cursor position.

30.91.3.8 int Fl_Text_Editor::kf_backspace (int *c*, Fl_Text_Editor * *e*) [static]

Does a backspace in the current buffer.

30.91.3.9 int Fl_Text_Editor::kf_c_s_move (int *c*, Fl_Text_Editor * *e*) [static]

Extends the current selection in the direction indicated by control key *c*.

30.91.3.10 int Fl_Text_Editor::kf_copy (int *c*, Fl_Text_Editor * *e*) [static]

Does a copy of selected text or the current character in the current buffer.

30.91.3.11 int Fl_Text_Editor::kf_cut (int *c*, Fl_Text_Editor * *e*) [static]

Does a cut of selected text in the current buffer.

30.91.3.12 int Fl_Text_Editor::kf_delete (int *c*, Fl_Text_Editor * *e*) [static]

Does a delete of selected text or the current character in the current buffer.

30.91.3.13 `int Fl_Text_Editor::kf_down (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor one line down.

30.91.3.14 `int Fl_Text_Editor::kf_end (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor to the end of the current line.

30.91.3.15 `int Fl_Text_Editor::kf_home (int, Fl_Text_Editor * e)` `[static]`

Moves the text cursor to the beginning of the current line.

30.91.3.16 `int Fl_Text_Editor::kf_insert (int c, Fl_Text_Editor * e)` `[static]`

Toggles the insert mode in the text editor.

30.91.3.17 `int Fl_Text_Editor::kf_left (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor one character to the left.

30.91.3.18 `int Fl_Text_Editor::kf_move (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor in the direction indicated by key *c*.

30.91.3.19 `int Fl_Text_Editor::kf_page_down (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor down one page.

30.91.3.20 `int Fl_Text_Editor::kf_page_up (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor up one page.

30.91.3.21 `int Fl_Text_Editor::kf_paste (int c, Fl_Text_Editor * e)` `[static]`

Does a paste of selected text in the current buffer.

30.91.3.22 `int Fl_Text_Editor::kf_right (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor one character to the right.

30.91.3.23 `int Fl_Text_Editor::kf_select_all (int c, Fl_Text_Editor * e)` `[static]`

Selects all text in the current buffer.

30.91.3.24 `int Fl_Text_Editor::kf_shift_move (int c, Fl_Text_Editor * e)` `[static]`

Extends the current selection in the direction of key *c*.

30.91.3.25 `int Fl_Text_Editor::kf_undo (int c, Fl_Text_Editor * e)` `[static]`

Undo last edit in the current buffer.

Also deselect previous selection.

30.91.3.26 `int Fl_Text_Editor::kf_up (int c, Fl_Text_Editor * e)` `[static]`

Moves the text cursor one line up.

30.91.3.27 `void Fl_Text_Editor::remove_all_key_bindings ()` `[inline]`

Removes all of the key bindings associated with the text editor or list.

30.91.3.28 `void Fl_Text_Editor::remove_all_key_bindings (Key_Binding ** list)`

Removes all of the key bindings associated with the text editor or list.

30.91.3.29 `void Fl_Text_Editor::remove_key_binding (int key, int state)` `[inline]`

Removes the key binding associated with the key "*key*" of state "*state*".

The documentation for this class was generated from the following files:

- `Fl_Text_Editor.H`
- `Fl_Text_Editor.cxx`

30.92 Fl_Text_Editor::Key_Binding Struct Reference

Simple linked list associating a key/state to a function.

```
#include <Fl_Text_Editor.H>
```

Public Attributes

- [Key_Func](#) function
associated function
- [int](#) `key`
the key pressed
- [Key_Binding](#) * `next`
next key binding in the list
- [int](#) `state`
the state of key modifiers

30.92.1 Detailed Description

Simple linked list associating a key/state to a function.

The documentation for this struct was generated from the following file:

- Fl_Text_Editor.H

30.93 Fl_Text_Selection Class Reference

This is an internal class for [Fl_Text_Buffer](#) to manage text selections.

```
#include <Fl_Text_Buffer.H>
```

Public Member Functions

- **int end ()**
- **int includes** (int pos, int lineStartPos, int dispIndex)
Return true if position pos with indentation dispIndex is in the [Fl_Text_Selection](#).
- **int position** (int *start, int *end, int *isRect, int *rectStart, int *rectEnd)
- **int position** (int *start, int *end)
- **int rect_end ()**
- **int rect_start ()**
- **char rectangular ()**
- **void selected** (char b)
- **char selected ()**
Returns a non-zero number if any text has been selected, or 0 if no text is selected.
- **void set** (int start, int end)
- **void set_rectangular** (int start, int end, int rectStart, int rectEnd)
- **int start ()**
- **void update** (int pos, int nDeleted, int nInserted)
Updates an individual selection for changes in the corresponding text.

Protected Attributes

- **int mEnd**
- **char mRectangular**
- **int mRectEnd**
- **int mRectStart**
- **char mSelected**
- **int mStart**

Friends

- class [Fl_Text_Buffer](#)

30.93.1 Detailed Description

This is an internal class for [Fl_Text_Buffer](#) to manage text selections.

Todo

members must be documented

The documentation for this class was generated from the following files:

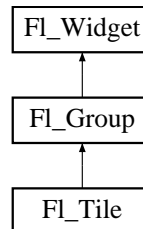
- Fl_Text_Buffer.H
- Fl_Text_Buffer.cxx

30.94 FL_Tile Class Reference

The [FL_Tile](#) class lets you resize the children by dragging the border between them:.

```
#include <Fl_Tile.H>
```

Inheritance diagram for [FL_Tile](#)::



Public Member Functions

- [FL_Tile](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [FL_Tile](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- void [position](#) (int, int, int, int)
Drag the intersection at from_x,from_y to to_x,to_y.
- void [resize](#) (int, int, int, int)
Resizes the [FL_Group](#) widget and all of its children.

30.94.1 Detailed Description

The [FL_Tile](#) class lets you resize the children by dragging the border between them:.

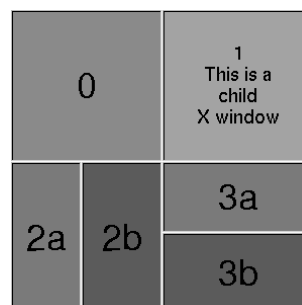


Figure 30.31: [FL_Tile](#)

For the tiling to work correctly, the children of an [FL_Tile](#) must cover the entire area of the widget, but not overlap. This means that all children must touch each other at their edges, and no gaps can't be left inside the [FL_Tile](#).

[FL_Tile](#) does not normailly draw any graphics of its own. The "borders" which can be seen in the snapshot above are actually part of the children. Their boxtypes have been set to `FL_DOWN_BOX` creating the impression of "ridges" where the boxes touch. What you see are actually two adjacent `FL_DOWN_BOX`'s drawn next to each other. All neighboring widgets share the same edge - the widget's thick borders make it appear as though the widgets aren't actually touching, but they are. If the edges of adjacent widgets do not touch, then it will be impossible to drag the corresponding edges.

[FL_Tile](#) allows objects to be resized to zero dimensions. To prevent this you can use the [resizable\(\)](#) to limit where corners can be dragged to.

Even though objects can be resized to zero sizes, they must initially have non-zero sizes so the [FL_Tile](#) can figure out their layout. If desired, call [position\(\)](#) after creating the children but before displaying the window to set the borders where you want.

Note on [resizable\(FL_Widget &w\)](#) : The "resizable" child widget (which should be invisible) limits where the border can be dragged to. If you don't set it, it will be possible to drag the borders right to the edge, and thus resize objects on the edge to zero width or height. The [resizable\(\)](#) widget is not resized by dragging any borders. See also void [FL_Group::resizable\(FL_Widget &w\)](#)

30.94.2 Constructor & Destructor Documentation

30.94.2.1 `FL_Tile::FL_Tile (int X, int Y, int W, int H, const char *l = 0)` `[inline]`

Creates a new [FL_Tile](#) widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the [FL_Tile](#) and all of it's children can be automatic (local) variables, but you must declare the [FL_Tile](#) *first*, so that it is destroyed last.

30.94.3 Member Function Documentation

30.94.3.1 `int FL_Tile::handle (int event)` `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Group](#).

30.94.3.2 void FL_Tile::position (int *oix*, int *oiy*, int *newx*, int *newy*)

Drag the intersection at from_x,from_y to to_x,to_y.

This redraws all the necessary children.

30.94.3.3 void FL_Tile::resize (int *X*, int *Y*, int *W*, int *H*) [virtual]

Resizes the [FL_Group](#) widget and all of its children.

The [FL_Group](#) widget first resizes itself, and then it moves and resizes all its children according to the rules documented for [FL_Group::resizable\(FL_Widget*\)](#)

See also:

[FL_Group::resizable\(FL_Widget*\)](#)
[FL_Group::resizable\(\)](#)
[FL_Widget::resize\(int,int,int,int\)](#)

Reimplemented from [FL_Group](#).

The documentation for this class was generated from the following files:

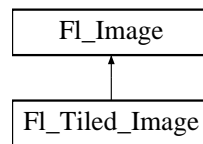
- [FL_Tile.H](#)
- [FL_Tile.cxx](#)

30.95 Fl_Tiled_Image Class Reference

This class supports tiling of images over a specified area.

```
#include <Fl_Tiled_Image.H>
```

Inheritance diagram for Fl_Tiled_Image::



Public Member Functions

- virtual void [color_average](#) ([Fl_Color](#) c, float i)
The [color_average\(\)](#) method averages the colors in the image with the FLTK color value c.
- [Fl_Image](#) * [copy](#) ()
The [copy\(\)](#) method creates a copy of the specified image.
- virtual [Fl_Image](#) * [copy](#) (int W, int H)
The [copy\(\)](#) method creates a copy of the specified image.
- virtual void [desaturate](#) ()
The [desaturate\(\)](#) method converts an image to grayscale.
- void [draw](#) (int X, int Y)
The [draw\(\)](#) methods draw the image.
- virtual void [draw](#) (int X, int Y, int W, int H, int cx, int cy)
The [draw\(\)](#) methods draw the image.
- [Fl_Tiled_Image](#) ([Fl_Image](#) *i, int W=0, int H=0)
The constructors create a new tiled image containing the specified image.
- [Fl_Image](#) * [image](#) ()
Gets The image that is shared.
- virtual [~Fl_Tiled_Image](#) ()
The destructor frees all memory and server resources that are used by the tiled image.

Protected Attributes

- int [alloc_image_](#)
- [Fl_Image](#) * [image_](#)

30.95.1 Detailed Description

This class supports tiling of images over a specified area.

The source (tile) image is **not** copied unless you call the [color_average\(\)](#), [desaturate\(\)](#), or [inactive\(\)](#) methods.

30.95.2 Constructor & Destructor Documentation

30.95.2.1 `Fl_Tiled_Image::Fl_Tiled_Image (Fl_Image * i, int W = 0, int H = 0)`

The constructors create a new tiled image containing the specified image.

Use a width and height of 0 to tile the whole window/widget.

30.95.3 Member Function Documentation

30.95.3.1 `void Fl_Tiled_Image::color_average (Fl_Color c, float i) [virtual]`

The [color_average\(\)](#) method averages the colors in the image with the FLTK color value *c*.

The *i* argument specifies the amount of the original image to combine with the color, so a value of 1.0 results in no color blend, and a value of 0.0 results in a constant image of the specified color. *The original image data is not altered by this method.*

Reimplemented from [Fl_Image](#).

30.95.3.2 `Fl_Image* Fl_Tiled_Image::copy () [inline]`

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [Fl_Shared_Image](#), released) when you are done with it.

Reimplemented from [Fl_Image](#).

30.95.3.3 `Fl_Image * Fl_Tiled_Image::copy (int W, int H) [virtual]`

The [copy\(\)](#) method creates a copy of the specified image.

If the width and height are provided, the image is resized to the specified size. The image should be deleted (or in the case of [Fl_Shared_Image](#), released) when you are done with it.

Reimplemented from [Fl_Image](#).

30.95.3.4 `void Fl_Tiled_Image::desaturate () [virtual]`

The [desaturate\(\)](#) method converts an image to grayscale.

If the image contains an alpha channel (depth = 4), the alpha channel is preserved. *This method does not alter the original image data.*

Reimplemented from [Fl_Image](#).

30.95.3.5 void FL_Tiled_Image::draw (int X, int Y) [inline]

The [draw\(\)](#) methods draw the image.

This form specifies the upper-lefthand corner of the image

Reimplemented from [FL_Image](#).

30.95.3.6 void FL_Tiled_Image::draw (int X, int Y, int W, int H, int cx, int cy) [virtual]

The [draw\(\)](#) methods draw the image.

This form specifies a bounding box for the image, with the origin (upper-lefthand corner) of the image offset by the cx and cy arguments.

Reimplemented from [FL_Image](#).

The documentation for this class was generated from the following files:

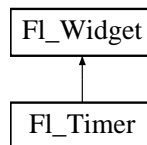
- FL_Tiled_Image.H
- FL_Tiled_Image.cxx

30.96 FL_Timer Class Reference

This is provided only to emulate the Forms Timer widget.

```
#include <Fl_Timer.H>
```

Inheritance diagram for FL_Timer::



Public Member Functions

- void [direction](#) (char d)
Gets or sets the direction of the timer.
- char [direction](#) () const
Gets or sets the direction of the timer.
- [FL_Timer](#) (uchar t, int x, int y, int w, int h, const char *l)
Creates a new [FL_Timer](#) widget using the given type, position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- void [suspended](#) (char d)
Gets or sets whether the timer is suspended.
- char [suspended](#) () const
Gets or sets whether the timer is suspended.
- double [value](#) () const
See void [FL_Timer::value\(double\)](#).
- void [value](#) (double)
Sets the current timer value.
- [~FL_Timer](#) ()
Destroys the timer and removes the timeout.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.96.1 Detailed Description

This is provided only to emulate the Forms Timer widget.

It works by making a timeout callback every 1/5 second. This is wasteful and inaccurate if you just want something to happen a fixed time in the future. You should directly call [Fl::add_timeout\(\)](#) instead.

30.96.2 Constructor & Destructor Documentation

30.96.2.1 `Fl_Timer::Fl_Timer (uchar t, int X, int Y, int W, int H, const char * l)`

Creates a new [Fl_Timer](#) widget using the given type, position, size, and label string.

The type parameter can be any of the following symbolic constants:

- `FL_NORMAL_TIMER` - The timer just does the callback and displays the string "Timer" in the widget.
- `FL_VALUE_TIMER` - The timer does the callback and displays the current timer value in the widget.
- `FL_HIDDEN_TIMER` - The timer just does the callback and does not display anything.

30.96.3 Member Function Documentation

30.96.3.1 `void Fl_Timer::direction (char d) [inline]`

Gets or sets the direction of the timer.

If the direction is zero then the timer will count up, otherwise it will count down from the initial [value\(\)](#).

30.96.3.2 `char Fl_Timer::direction () const [inline]`

Gets or sets the direction of the timer.

If the direction is zero then the timer will count up, otherwise it will count down from the initial [value\(\)](#).

30.96.3.3 `void Fl_Timer::draw () [protected, virtual]`

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.96.3.4 `int FL_Timer::handle (int event)` `[virtual]`

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Widget](#).

30.96.3.5 `void FL_Timer::suspended (char d)`

Gets or sets whether the timer is suspended.

30.96.3.6 `char FL_Timer::suspended () const` `[inline]`

Gets or sets whether the timer is suspended.

The documentation for this class was generated from the following files:

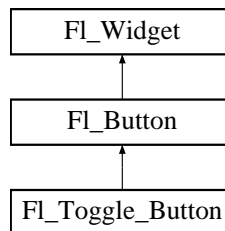
- `FL_Timer.H`
- `forms_timer.cxx`

30.97 Fl_Toggle_Button Class Reference

The toggle button is a push button that needs to be clicked once to toggle on, and one more time to toggle off.

```
#include <Fl_Toggle_Button.H>
```

Inheritance diagram for Fl_Toggle_Button::



Public Member Functions

- [Fl_Toggle_Button](#) (int X, int Y, int W, int H, const char *l=0)
Creates a new [Fl_Toggle_Button](#) widget using the given position, size, and label string.

30.97.1 Detailed Description

The toggle button is a push button that needs to be clicked once to toggle on, and one more time to toggle off.

The [Fl_Toggle_Button](#) subclass displays the "on" state by drawing a pushed-in button.

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for [type\(\)](#) and [when\(\)](#).

30.97.2 Constructor & Destructor Documentation

30.97.2.1 [Fl_Toggle_Button::Fl_Toggle_Button](#) (int X, int Y, int W, int H, const char *l = 0) [inline]

Creates a new [Fl_Toggle_Button](#) widget using the given position, size, and label string.

The inherited destructor deletes the toggle button.

The documentation for this class was generated from the following file:

- [Fl_Toggle_Button.H](#)

30.98 FL_Tooltip Class Reference

The [FL_Tooltip](#) class provides tooltip support for all FLTK widgets.

```
#include <Fl_Tooltip.H>
```

Static Public Member Functions

- static void [color](#) (unsigned c)
Sets the background color for tooltips.
- static [FL_Color](#) [color](#) ()
Gets the background color for tooltips.
- static void [current](#) ([FL_Widget](#) *)
Sets the current widget target.
- static [FL_Widget](#) * [current](#) ()
Gets the current widget target.
- static void [delay](#) (float f)
Sets the tooltip delay.
- static float [delay](#) ()
Gets the tooltip delay.
- static void [disable](#) ()
Same as [enable\(0\)](#), disables tooltips on all widgets.
- static void [enable](#) (int b=1)
Enables tooltips on all widgets (or disables if b is false).
- static int [enabled](#) ()
Returns non-zero if tooltips are enabled.
- static void [enter_area](#) ([FL_Widget](#) *w, int X, int Y, int W, int H, const char *tip)
You may be able to use this to provide tooltips for internal pieces of your widget.
- static void [font](#) ([FL_Font](#) i)
Sets the typeface for the tooltip text.
- static [FL_Font](#) [font](#) ()
Gets the typeface for the tooltip text.
- static void [hoverdelay](#) (float f)
Sets the tooltip hover delay, the delay between tooltips.
- static float [hoverdelay](#) ()
Gets the tooltip hover delay, the delay between tooltips.

- static void [size](#) ([FL_Fontsize](#) s)
Sets the size of the tooltip text.
- static [FL_Fontsize](#) [size](#) ()
Gets the size of the tooltip text.
- static void [textcolor](#) (unsigned c)
Sets the color of the text in the tooltip.
- static [FL_Color](#) [textcolor](#) ()
Gets the color of the text in the tooltip.

Static Public Attributes

- static void(* [enter](#))([FL_Widget](#) *w) = nothing
- static void(* [exit](#))([FL_Widget](#) *w) = nothing

Friends

- void [FL_Widget::tooltip](#) (const char *)

30.98.1 Detailed Description

The [FL_Tooltip](#) class provides tooltip support for all FLTK widgets.

It contains only static methods.

30.98.2 Member Function Documentation

30.98.2.1 static void [FL_Tooltip::color](#) (unsigned c) [inline, static]

Sets the background color for tooltips.

The default background color is a pale yellow.

30.98.2.2 static [FL_Color](#) [FL_Tooltip::color](#) () [inline, static]

Gets the background color for tooltips.

The default background color is a pale yellow.

30.98.2.3 void [FL_Tooltip::current](#) ([FL_Widget](#) * w) [static]

Sets the current widget target.

Acts as though [enter](#)(widget) was done but does not pop up a tooltip. This is useful to prevent a tooltip from reappearing when a modal overlapping window is deleted. FLTK does this automatically when you click the mouse button.

30.98.2.4 static void Fl_Tooltip::delay (float *f*) [inline, static]

Sets the tooltip delay.

The default delay is 1.0 seconds.

30.98.2.5 static float Fl_Tooltip::delay () [inline, static]

Gets the tooltip delay.

The default delay is 1.0 seconds.

30.98.2.6 static void Fl_Tooltip::disable () [inline, static]

Same as enable(0), disables tooltips on all widgets.

30.98.2.7 static void Fl_Tooltip::enable (int *b* = 1) [inline, static]

Enables tooltips on all widgets (or disables if *b* is false).

30.98.2.8 static int Fl_Tooltip::enabled () [inline, static]

Returns non-zero if tooltips are enabled.

30.98.2.9 void Fl_Tooltip::enter_area (Fl_Widget * *wid*, int *x*, int *y*, int *w*, int *h*, const char * *t*)
[static]

You may be able to use this to provide tooltips for internal pieces of your widget.

Call this after setting [Fl::belowmouse\(\)](#) to your widget (because that calls the above enter() method). Then figure out what thing the mouse is pointing at, and call this with the widget (this pointer is used to remove the tooltip if the widget is deleted or hidden, and to locate the tooltip), the rectangle surrounding the area, relative to the top-left corner of the widget (used to calculate where to put the tooltip), and the text of the tooltip (which must be a pointer to static data as it is not copied).

30.98.2.10 static void Fl_Tooltip::font (Fl_Font *i*) [inline, static]

Sets the typeface for the tooltip text.

30.98.2.11 static Fl_Font Fl_Tooltip::font () [inline, static]

Gets the typeface for the tooltip text.

30.98.2.12 static void Fl_Tooltip::hoverdelay (float *f*) [inline, static]

Sets the tooltip hover delay, the delay between tooltips.

The default delay is 0.2 seconds.

30.98.2.13 static float FL_Tooltip::hoverdelay () [inline, static]

Gets the tooltip hover delay, the delay between tooltips.

The default delay is 0.2 seconds.

30.98.2.14 static void FL_Tooltip::size (FL_Fontsize s) [inline, static]

Sets the size of the tooltip text.

30.98.2.15 static FL_Fontsize FL_Tooltip::size () [inline, static]

Gets the size of the tooltip text.

30.98.2.16 static void FL_Tooltip::textcolor (unsigned c) [inline, static]

Sets the color of the text in the tooltip.

The default is black.

30.98.2.17 static FL_Color FL_Tooltip::textcolor () [inline, static]

Gets the color of the text in the tooltip.

The default is black.

The documentation for this class was generated from the following files:

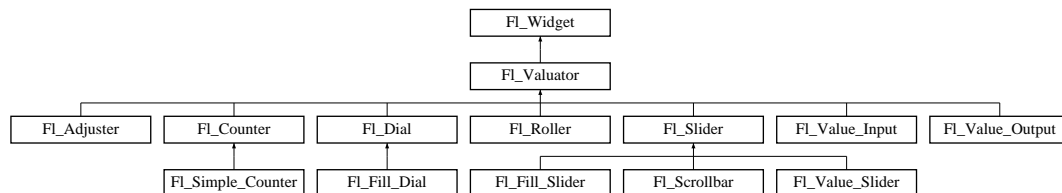
- FL_Tooltip.H
- FL.cxx
- FL_Tooltip.cxx

30.99 FL_Valuator Class Reference

The [FL_Valuator](#) class controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object.

```
#include <Fl_Valuator.H>
```

Inheritance diagram for FL_Valuator::



Public Member Functions

- void [bounds](#) (double a, double b)
Sets the minimum (a) and maximum (b) values for the valuator widget.
- double [clamp](#) (double)
Clamps the passed value to the valuator range.
- virtual int [format](#) (char *)
Uses internal rules to format the fields numerical value into the character array pointed to by the passed parameter.
- double [increment](#) (double, int)
Adds n times the step value to the passed value.
- void [maximum](#) (double a)
Sets the maximum value for the valuator.
- double [maximum](#) () const
Gets the maximum value for the valuator.
- void [minimum](#) (double a)
Sets the minimum value for the valuator.
- double [minimum](#) () const
Gets the minimum value for the valuator.
- void [precision](#) (int)
Sets the step value to $1/10^{\text{digits}}$.
- void [range](#) (double a, double b)
Sets the minimum and maximum values for the valuator.
- double [round](#) (double)

Round the passed value to the nearest step increment.

- double [step](#) () const
Gets or sets the step value.
- void [step](#) (double s)
See double [FL_Valuator::step\(\)](#) const.
- void [step](#) (double a, int b)
See double [FL_Valuator::step\(\)](#) const.
- void [step](#) (int a)
See double [FL_Valuator::step\(\)](#) const.
- int [value](#) (double)
Sets the current value.
- double [value](#) () const
Gets the floating point(double) value.

Protected Member Functions

- [FL_Valuator](#) (int X, int Y, int W, int H, const char *L)
Creates a new [FL_Valuator](#) widget using the given position, size, and label string.
- void [handle_drag](#) (double newvalue)
Called during a drag operation, after an [FL_WHEN_CHANGED](#) event is received and before the callback.
- void [handle_push](#) ()
Stores the current value in the previous value.
- void [handle_release](#) ()
Called after an [FL_WHEN_RELEASE](#) event is received and before the callback.
- int [horizontal](#) () const
Tells if the valuator is an [FL_HORIZONTAL](#) one.
- double [previous_value](#) () const
Gets the previous floating point value before an event changed it.
- void [set_value](#) (double v)
Sets the current floating point value.
- double [softclamp](#) (double)
Clamps the value, but accepts v if the previous value is not already out of range.
- virtual void [value_damage](#) ()
Asks for partial redraw.

30.99.1 Detailed Description

The [FL_Valuator](#) class controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object.

There are probably more of these classes in FLTK than any others:

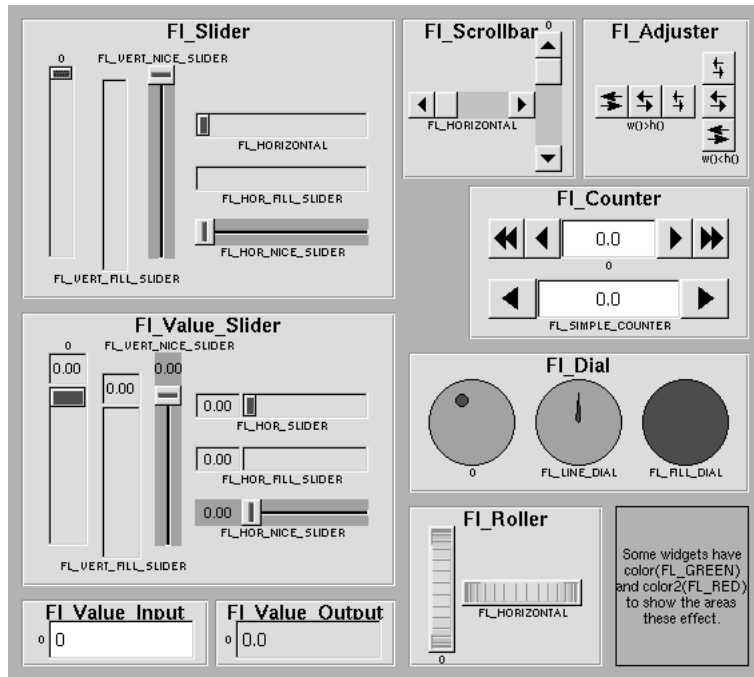


Figure 30.32: Valuators derived from FL_Valuators

In the above diagram each box surrounds an actual subclass. These are further differentiated by setting the [type\(\)](#) of the widget to the symbolic value labeling the widget. The ones labelled "0" are the default versions with a [type\(0\)](#). For consistency the symbol `FL_VERTICAL` is defined as zero.

30.99.2 Constructor & Destructor Documentation

30.99.2.1 `FL_Valuator::FL_Valuator (int X, int Y, int W, int H, const char * L)` [protected]

Creates a new [FL_Valuator](#) widget using the given position, size, and label string.

The default boxtype is `FL_NO_BOX`.

30.99.3 Member Function Documentation

30.99.3.1 `void FL_Valuator::bounds (double a, double b)` [inline]

Sets the minimum (a) and maximum (b) values for the valuator widget.

Reimplemented in [FL_Slider](#).

30.99.3.2 double FL_Valuator::clamp (double *v*)

Clamps the passed value to the valuator range.

30.99.3.3 int FL_Valuator::format (char * *buffer*) [virtual]

Uses internal rules to format the fields numerical value into the character array pointed to by the passed parameter.

The actual format used depends on the current step value. If the step value has been set to zero then a g format is used. If the step value is non-zero, then a %.*f* format is used, where the precision is calculated to show sufficient digits for the current step value. An integer step value, such as 1 or 1.0, gives a precision of 0, so the formatted value will appear as an integer.

This method is used by the FL_Value_... group of widgets to format the current value into a text string. The return value is the length of the formatted text. The formatted value is written into in *buffer*. *buffer* should have space for at least 128 bytes.

You may override this function to create your own text formatting.

30.99.3.4 void FL_Valuator::handle_drag (double *v*) [protected]

Called during a drag operation, after an FL_WHEN_CHANGED event is received and before the callback.

30.99.3.5 void FL_Valuator::handle_release () [protected]

Called after an FL_WHEN_RELEASE event is received and before the callback.

30.99.3.6 double FL_Valuator::increment (double *v*, int *n*)

Adds *n* times the step value to the passed value.

If step was set to zero it uses fabs([maximum\(\)](#) - [minimum\(\)](#)) / 100.

30.99.3.7 void FL_Valuator::maximum (double *a*) [inline]

Sets the maximum value for the valuator.

30.99.3.8 double FL_Valuator::maximum () const [inline]

Gets the maximum value for the valuator.

30.99.3.9 void FL_Valuator::minimum (double *a*) [inline]

Sets the minimum value for the valuator.

30.99.3.10 double FL_Valuator::minimum () const [inline]

Gets the minimum value for the valuator.

30.99.3.11 void FL_Valuator::precision (int *p*)

Sets the step value to $1/10^{\text{digits}}$.

30.99.3.12 void FL_Valuator::range (double *a*, double *b*) [inline]

Sets the minimum and maximum values for the valuator.

When the user manipulates the widget, the value is limited to this range. This clamping is done *after* rounding to the step value (this makes a difference if the range is not a multiple of the step).

The minimum may be greater than the maximum. This has the effect of "reversing" the object so the larger values are in the opposite direction. This also switches which end of the filled sliders is filled.

Some widgets consider this a "soft" range. This means they will stop at the range, but if the user releases and grabs the control again and tries to move it further, it is allowed.

The range may affect the display. You must [redraw\(\)](#) the widget after changing the range.

30.99.3.13 double FL_Valuator::round (double *v*)

Round the passed value to the nearest step increment.

Does nothing if step is zero.

30.99.3.14 void FL_Valuator::set_value (double *v*) [inline, protected]

Sets the current floating point value.

30.99.3.15 double FL_Valuator::step () const [inline]

Gets or sets the step value.

As the user moves the mouse the value is rounded to the nearest multiple of the step value. This is done *before* clamping it to the range. For most widgets the default step is zero.

For precision the step is stored as the ratio of two integers, A/B. You can set these integers directly. Currently setting a floating point value sets the nearest A/1 or 1/B value possible.

30.99.3.16 int FL_Valuator::value (double *v*)

Sets the current value.

The new value is *not* clamped or otherwise changed before storing it. Use [clamp\(\)](#) or [round\(\)](#) to modify the value before calling [value\(\)](#). The widget is redrawn if the new value is different than the current one. The initial value is zero.

[changed\(\)](#) will return true if the user has moved the slider, but it will be turned off by [value\(x\)](#) and just before doing a callback (the callback can turn it back on if desired).

30.99.3.17 double FL_Valuator::value () const [inline]

Gets the floating point(double) value.

See int [value\(double\)](#)

Reimplemented in [Fl_Scrollbar](#).

The documentation for this class was generated from the following files:

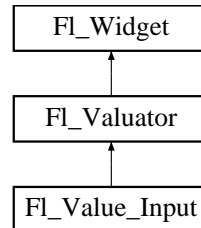
- Fl_Valuator.H
- Fl_Valuator.cxx

30.100 Fl_Value_Input Class Reference

The [Fl_Value_Input](#) widget displays a numeric value.

```
#include <Fl_Value_Input.H>
```

Inheritance diagram for `Fl_Value_Input`:



Public Member Functions

- void [cursor_color](#) (unsigned n)
Sets the color of the text cursor.
- [Fl_Color](#) [cursor_color](#) () const
Gets the color of the text cursor.
- [Fl_Value_Input](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [Fl_Value_Input](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- void [resize](#) (int, int, int, int)
Changes the size or position of the widget.
- void [shortcut](#) (int s)
See int [Fl_Value_Input::shortcut\(\)](#) const.
- int [shortcut](#) () const
The first form returns the current shortcut key for the Input.
- char [soft](#) () const
If "soft" is turned on, the user is allowed to drag the value outside the range.
- void [soft](#) (char s)
See void [Fl_Value_Input::soft\(char s\)](#).
- void [textcolor](#) (unsigned n)
Sets the color of the text in the value box.
- [Fl_Color](#) [textcolor](#) () const
Gets the color of the text in the value box.

- void [textfont](#) ([Fl_Font](#) s)

Sets the typeface of the text in the value box.

- [Fl_Font](#) [textfont](#) () const

Gets the typeface of the text in the value box.

- void [textsize](#) ([Fl_Fonsize](#) s)

Sets the size of the text in the value box.

- [Fl_Fonsize](#) [textsize](#) () const

Gets the size of the text in the value box.

Public Attributes

- [Fl_Input](#) [input](#)

Protected Member Functions

- void [draw](#) ()

Draws the widget.

30.100.1 Detailed Description

The [Fl_Value_Input](#) widget displays a numeric value.

The user can click in the text field and edit it - there is in fact a hidden [Fl_Input](#) widget with type(FL_FLOAT_INPUT) or type(FL_INT_INPUT) in there - and when they hit return or tab the value updates to what they typed and the callback is done.

If [step\(\)](#) is non-zero and integral, then the range of numbers is limited to integers instead of floating point numbers. As well as displaying the value as an integer, typed input is also limited to integer values, even if the hidden [Fl_Input](#) widget is of type(FL_FLOAT_INPUT).

If [step\(\)](#) is non-zero, the user can also drag the mouse across the object and thus slide the value. The left button moves one [step\(\)](#) per pixel, the middle by 10 [step\(\)](#), and the right button by 100 * [step\(\)](#). It is therefore impossible to select text by dragging across it, although clicking can still move the insertion cursor.

If [step\(\)](#) is non-zero and integral, then the range of numbers are limited to integers instead of floating point values.

Figure 30.33: Fl_Value_Input

30.100.2 Constructor & Destructor Documentation

30.100.2.1 Fl_Value_Input::Fl_Value_Input (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [Fl_Value_Input](#) widget using the given position, size, and label string.

The default boxtype is FL_DOWN_BOX.

30.100.3 Member Function Documentation

30.100.3.1 void Fl_Value_Input::cursor_color (unsigned *n*) [inline]

Sets the color of the text cursor.

The text cursor is black by default.

30.100.3.2 Fl_Color Fl_Value_Input::cursor_color () const [inline]

Gets the color of the text cursor.

The text cursor is black by default.

30.100.3.3 void Fl_Value_Input::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.100.3.4 int Fl_Value_Input::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

30.100.3.5 void Fl_Value_Input::resize (int x, int y, int w, int h) [virtual]

Changes the size or position of the widget.

This is a virtual function so that the widget may implement its own handling of resizing. The default version does *not* call the [redraw\(\)](#) method, but instead relies on the parent widget to do so because the parent may know a faster way to update the display, such as scrolling from the old position.

Some window managers under X11 call [resize\(\)](#) a lot more often than needed. Please verify that the position or size of a widget did actually change before doing any extensive calculations.

`position(X, Y)` is a shortcut for `resize(X, Y, w(), h())`, and `size(W, H)` is a shortcut for `resize(x(), y(), W, H)`.

Parameters:

← *x,y* new position relative to the parent window

← *w,h* new size

See also:

[position\(int,int\)](#), [size\(int,int\)](#)

Reimplemented from [Fl_Widget](#).

30.100.3.6 int Fl_Value_Input::shortcut () const [inline]

The first form returns the current shortcut key for the Input.

The second form sets the shortcut key to key. Setting this overrides the use of '&' in the [label\(\)](#). The value is a bitwise OR of a key and a set of shift flags, for example `FL_ALT | 'a'`, `FL_ALT | (FL_F + 10)`, or just `'a'`. A value of 0 disables the shortcut.

The key can be any value returned by [Fl::event_key\(\)](#), but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by [Fl::event_state\(\)](#). If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

30.100.3.7 char Fl_Value_Input::soft () const [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. The default is true.

30.100.3.8 void Fl_Value_Input::textcolor (unsigned *n*) [inline]

Sets the color of the text in the value box.

30.100.3.9 Fl_Color Fl_Value_Input::textcolor () const [inline]

Gets the color of the text in the value box.

30.100.3.10 void Fl_Value_Input::textfont (Fl_Font *s*) [inline]

Sets the typeface of the text in the value box.

30.100.3.11 Fl_Font Fl_Value_Input::textfont () const [inline]

Gets the typeface of the text in the value box.

30.100.3.12 void Fl_Value_Input::textsize (Fl_Fontsize *s*) [inline]

Sets the size of the text in the value box.

30.100.3.13 Fl_Fontsize Fl_Value_Input::textsize () const [inline]

Gets the size of the text in the value box.

The documentation for this class was generated from the following files:

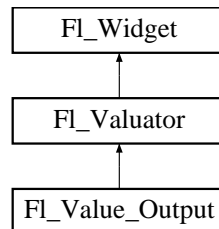
- Fl_Value_Input.H
- Fl_Value_Input.cxx

30.101 Fl_Value_Output Class Reference

The [Fl_Value_Output](#) widget displays a floating point value.

```
#include <Fl_Value_Output.H>
```

Inheritance diagram for `Fl_Value_Output`:



Public Member Functions

- [Fl_Value_Output](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [Fl_Value_Output](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- [uchar soft](#) () const
If "soft" is turned on, the user is allowed to drag the value outside the range.
- void [soft](#) ([uchar s](#))
If "soft" is turned on, the user is allowed to drag the value outside the range.
- void [textcolor](#) (unsigned s)
Gets the color of the text in the value box.
- [Fl_Color textcolor](#) () const
Sets the color of the text in the value box.
- void [textfont](#) ([Fl_Font s](#))
Sets the typeface of the text in the value box.
- [Fl_Font textfont](#) () const
Gets the typeface of the text in the value box.
- void [textsize](#) ([Fl_Fontsize s](#))
- [Fl_Fontsize textsize](#) () const
Gets the size of the text in the value box.

Protected Member Functions

- void [draw](#) ()

Draws the widget.

30.101.1 Detailed Description

The [FL_Value_Output](#) widget displays a floating point value.

If [step\(\)](#) is not zero, the user can adjust the value by dragging the mouse left and right. The left button moves one [step\(\)](#) per pixel, the middle by $10 * \text{step}()$, and the right button by $100 * \text{step}()$.

This is much lighter-weight than [FL_Value_Input](#) because it contains no text editing code or character buffer.

Figure 30.34: FL_Value_Output

30.101.2 Constructor & Destructor Documentation

30.101.2.1 [FL_Value_Output::FL_Value_Output](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [FL_Value_Output](#) widget using the given position, size, and label string.

The default boxtype is FL_NO_BOX.

Inherited destructor destroys the Valuator.

30.101.3 Member Function Documentation

30.101.3.1 void Fl_Value_Output::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;          // scroll is an embedded Fl_Scrollbar
s->draw();                        // calls Fl_Scrollbar::draw()
```

Implements [Fl_Widget](#).

30.101.3.2 int Fl_Value_Output::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Widget](#).

30.101.3.3 uchar Fl_Value_Output::soft () const [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

30.101.3.4 void Fl_Value_Output::soft (uchar s) [inline]

If "soft" is turned on, the user is allowed to drag the value outside the range.

If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

30.101.3.5 void Fl_Value_Output::textcolor (unsigned s) [inline]

Gets the color of the text in the value box.

30.101.3.6 Fl_Color Fl_Value_Output::textcolor () const [inline]

Sets the color of the text in the value box.

30.101.3.7 void Fl_Value_Output::textfont (Fl_Font s) [inline]

Sets the typeface of the text in the value box.

30.101.3.8 Fl_Font Fl_Value_Output::textfont () const [inline]

Gets the typeface of the text in the value box.

30.101.3.9 Fl_Fonsize Fl_Value_Output::textsize () const [inline]

Gets the size of the text in the value box.

The documentation for this class was generated from the following files:

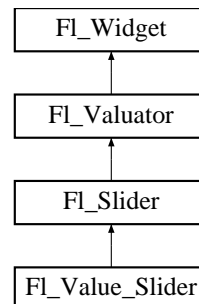
- Fl_Value_Output.H
- Fl_Value_Output.cxx

30.102 FL_Value_Slider Class Reference

The [FL_Value_Slider](#) widget is a [FL_Slider](#) widget with a box displaying the current value.

```
#include <Fl_Value_Slider.H>
```

Inheritance diagram for `FL_Value_Slider`:



Public Member Functions

- [FL_Value_Slider](#) (int x, int y, int w, int h, const char *l=0)
Creates a new [FL_Value_Slider](#) widget using the given position, size, and label string.
- int [handle](#) (int)
Handles the specified event.
- void [textcolor](#) (unsigned s)
Sets the color of the text in the value box.
- [FL_Color textcolor](#) () const
Gets the color of the text in the value box.
- void [textfont](#) ([FL_Font](#) s)
Sets the typeface of the text in the value box.
- [FL_Font textfont](#) () const
Gets the typeface of the text in the value box.
- void [textsize](#) ([FL_Fontsize](#) s)
Sets the size of the text in the value box.
- [FL_Fontsize textsize](#) () const
Gets the size of the text in the value box.

Protected Member Functions

- void [draw](#) ()
Draws the widget.

30.102.1 Detailed Description

The [Fl_Value_Slider](#) widget is a [Fl_Slider](#) widget with a box displaying the current value.

Figure 30.35: Fl_Value_Slider

30.102.2 Constructor & Destructor Documentation

30.102.2.1 [Fl_Value_Slider::Fl_Value_Slider](#) (int *X*, int *Y*, int *W*, int *H*, const char * *l* = 0)

Creates a new [Fl_Value_Slider](#) widget using the given position, size, and label string.

The default boxtype is `FL_DOWN_BOX`.

30.102.3 Member Function Documentation

30.102.3.1 [void Fl_Value_Slider::draw](#) () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to `Fl_Widget*` and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [Fl_Slider](#).

30.102.3.2 [int Fl_Value_Slider::handle](#) (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood
1 if the event was used and can be deleted

See also:

[Fl_Event](#)

Reimplemented from [Fl_Slider](#).

30.102.3.3 void Fl_Value_Slider::textcolor (unsigned s) [inline]

Sets the color of the text in the value box.

30.102.3.4 Fl_Color Fl_Value_Slider::textcolor () const [inline]

Gets the color of the text in the value box.

30.102.3.5 void Fl_Value_Slider::textfont (Fl_Font s) [inline]

Sets the typeface of the text in the value box.

30.102.3.6 Fl_Font Fl_Value_Slider::textfont () const [inline]

Gets the typeface of the text in the value box.

30.102.3.7 void Fl_Value_Slider::textsize (Fl_Fonsize s) [inline]

Sets the size of the text in the value box.

30.102.3.8 Fl_Fonsize Fl_Value_Slider::textsize () const [inline]

Gets the size of the text in the value box.

The documentation for this class was generated from the following files:

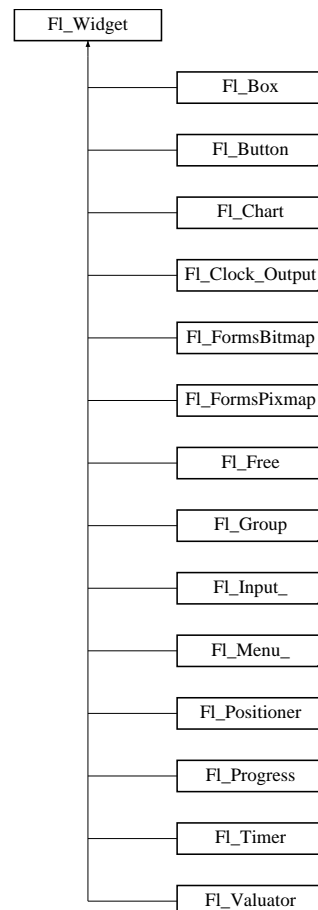
- [Fl_Value_Slider.H](#)
- [Fl_Value_Slider.cxx](#)

30.103 Fl_Widget Class Reference

[Fl_Widget](#) is the base class for all widgets in FLTK.

```
#include <Fl_Widget.H>
```

Inheritance diagram for Fl_Widget::



Public Member Functions

- void [activate](#) ()
Activates the widget.
- int [active](#) () const
Returns whether the widget is active.
- int [active_r](#) () const
Returns whether the widget and all of its parents are active.
- void [align](#) ([Fl_Align](#) alignment)
Sets the label alignment.

- `Fl_Align align () const`
Gets the label alignment.
- `void argument (long v)`
Sets the current user data (long) argument that is passed to the callback function.
- `long argument () const`
Gets the current user data (long) argument that is passed to the callback function.
- `void box (Fl_Boxtype new_box)`
Sets the box type for the widget.
- `Fl_Boxtype box () const`
Gets the box type of the widget.
- `void callback (Fl_Callback1 *cb, long p=0)`
Sets the current callback function for the widget.
- `void callback (Fl_Callback0 *cb)`
Sets the current callback function for the widget.
- `void callback (Fl_Callback *cb)`
Sets the current callback function for the widget.
- `void callback (Fl_Callback *cb, void *p)`
Sets the current callback function for the widget.
- `Fl_Callback_p callback () const`
Gets the current callback function for the widget.
- `int changed () const`
Checks if the widget value changed since the last callback.
- `void clear_changed ()`
Marks the value of the widget as unchanged.
- `void clear_damage (uchar c=0)`
Clears the damage flags.
- `void clear_output ()`
Sets a widget to accept input.
- `void clear_visible ()`
Hides the widget.
- `void clear_visible_focus ()`
Disables keyboard focus navigation with this widget.
- `void color (unsigned bg, unsigned sel)`
Sets the background and selection color of the widget.

- void `color` (unsigned bg)
Sets the background color of the widget.
- `FL_Color color` () const
Gets the background color of the widget.
- void `color2` (unsigned a)
For back compatibility only.
- `FL_Color color2` () const
For back compatibility only.
- int `contains` (const `FL_Widget *w`) const
Checks if w is a child of this widget.
- void `copy_label` (`FL_CString new_label`)
Sets the current label.
- void `damage` (`uchar c`, int x, int y, int w, int h)
Sets the damage bits for an area inside the widget.
- void `damage` (`uchar c`)
Sets the damage bits for the widget.
- `uchar damage` () const
Returns non-zero if `draw()` needs to be called.
- int `damage_resize` (int, int, int, int)
Internal use only.
- void `deactivate` ()
Deactivates the widget.
- void `deimage` (`FL_Image &img`)
Sets the image to use as part of the widget label.
- void `deimage` (`FL_Image *img`)
Sets the image to use as part of the widget label.
- `FL_Image * deimage` ()
Gets the image that is used as part of the widget label.
- void `do_callback` (`FL_Widget *o`, void *arg=0)
Calls the widget callback.
- void `do_callback` (`FL_Widget *o`, long arg)
Calls the widget callback.
- void `do_callback` ()

Calls the widget callback.

- virtual void [draw](#) ()=0

Draws the widget.

- void [draw_label](#) (int, int, int, int, [Fl_Align](#)) const

Draws the label in an arbitrary bounding box with an arbitrary alignment.

- int [h](#) () const

Gets the widget height.

- virtual int [handle](#) (int event)

Handles the specified event.

- virtual void [hide](#) ()

Makes a widget invisible.

- void [image](#) ([Fl_Image](#) &img)

Sets the image to use as part of the widget label.

- void [image](#) ([Fl_Image](#) *img)

Sets the image to use as part of the widget label.

- [Fl_Image](#) * [image](#) ()

Gets the image that is used as part of the widget label.

- int [inside](#) (const [Fl_Widget](#) *w) const

Checks if this widget is a child of w.

- void [label](#) ([Fl_Labeltype](#) a, const char *b)

Shortcut to set the label text and type in one call.

- void [label](#) (const char *text)

Sets the current label pointer.

- const char * [label](#) () const

Gets the current label text.

- void [labelcolor](#) (unsigned c)

Sets the label color.

- [Fl_Color](#) [labelcolor](#) () const

Gets the label color.

- void [labelfont](#) ([Fl_Font](#) f)

Sets the font to use.

- [Fl_Font](#) [labelfont](#) () const

Gets the font to use.

- void `labelsize` (`Fl_Fontsize` pix)
Sets the font size in pixels.
- `Fl_Fontsize` `labelsize` () const
Gets the font size in pixels.
- void `labeltype` (`Fl_Labeltype` a)
Sets the label type.
- `Fl_Labeltype` `labeltype` () const
Gets the label type.
- void `measure_label` (int &ww, int &hh)
Sets width ww and height hh accordingly with the label size.
- int `output` () const
Returns if a widget is used for output only.
- void `parent` (`Fl_Group` *p)
Internal use only - "for hacks only".
- `Fl_Group` * `parent` () const
Returns a pointer to the parent widget.
- void `position` (int X, int Y)
Repositions the window or widget.
- void `redraw` ()
Schedules the drawing of the widget.
- void `redraw_label` ()
Schedules the drawing of the label.
- virtual void `resize` (int x, int y, int w, int h)
Changes the size or position of the widget.
- void `selection_color` (unsigned a)
Sets the selection color.
- `Fl_Color` `selection_color` () const
Gets the selection color.
- void `set_changed` ()
Marks the value of the widget as changed.
- void `set_output` ()
Sets a widget to output only.
- void `set_visible` ()
Makes the widget visible.

- void [set_visible_focus](#) ()
Enables keyboard focus navigation with this widget.
- virtual void [show](#) ()
Makes a widget visible.
- void [size](#) (int W, int H)
Changes the size of the widget.
- int [take_focus](#) ()
Gives the widget the keyboard focus.
- int [takeevents](#) () const
Returns if the widget is able to take events.
- int [test_shortcut](#) ()
Internal use only.
- void [tooltip](#) (const char *t)
Sets the current tooltip text.
- const char * [tooltip](#) () const
Gets the current tooltip text.
- void [type](#) (uchar t)
Sets the widget type.
- uchar [type](#) () const
Gets the widget type.
- void [user_data](#) (void *v)
Sets the user data for this widget.
- void * [user_data](#) () const
Gets the user data for this widget.
- int [visible](#) () const
Returns whether a widget is visible.
- int [visible_focus](#) ()
Checks whether this widget has a visible focus.
- void [visible_focus](#) (int v)
Modifies keyboard focus navigation.
- int [visible_r](#) () const
Returns whether a widget and all its parents are visible.
- int [w](#) () const

Gets the widget width.

- void `when (uchar i)`
Sets the flags used to decide when a callback is called.
- `Fl_When when () const`
Returns the conditions under which the callback is called.
- `Fl_Window * window () const`
Returns a pointer to the primary `Fl_Window` widget.
- `int x () const`
Gets the widget position in its window.
- `int y () const`
Gets the widget position in its window.
- virtual `~Fl_Widget ()`
Destroys the widget.

Static Public Member Functions

- static void `default_callback (Fl_Widget *cb, void *d)`
Sets the default callback for all widgets.
- static char `label_shortcut (const char *t)`
Internal use only.
- static int `test_shortcut (const char *)`
Internal use only.

Protected Types

- enum {
 `INACTIVE` = 1, `INVISIBLE` = 2, `OUTPUT` = 4, `SHORTCUT_LABEL` = 64,
 `CHANGED` = 128, `VISIBLE_FOCUS` = 512, `COPIED_LABEL` = 1024 }
flags possible values enumeration.

Protected Member Functions

- void `clear_flag (int c)`
Clears a flag in the flags mask.
- void `draw_box (Fl_Boxtype t, int x, int y, int w, int h, Fl_Color c) const`
Draws a box of type `t`, of color `c` at the position `X,Y` and size `W,H`.

- void [draw_box](#) ([Fl_Boxtype](#) t, [Fl_Color](#) c) const
Draws a box of type t, of color c at the widget's position and size.
- void [draw_box](#) () const
Draws the widget box according its box style.
- void [draw_focus](#) ([Fl_Boxtype](#) t, int x, int y, int w, int h) const
Draws a focus box for the widget at the given position and size.
- void [draw_focus](#) ()
draws a focus rectangle around the widget
- void [draw_label](#) (int, int, int, int) const
Draws the label in an arbitrary bounding box.
- void [draw_label](#) () const
Draws the widget's label at the defined label position.
- [Fl_Widget](#) (int x, int y, int w, int h, [Fl_CString](#) label=0L)
Creates a widget at the given position and size.
- int [flags](#) () const
Gets the widget flags mask.
- void [h](#) (int v)
Internal use only.
- void [set_flag](#) (int c)
Sets a flag in the flags mask.
- void [w](#) (int v)
Internal use only.
- void [x](#) (int v)
Internal use only.
- void [y](#) (int v)
Internal use only.

Friends

- class [Fl_Group](#)

30.103.1 Detailed Description

[FL_Widget](#) is the base class for all widgets in FLTK.

You can't create one of these because the constructor is not public. However you can subclass it.

All "property" accessing methods, such as [color\(\)](#), [parent\(\)](#), or [argument\(\)](#) are implemented as trivial inline functions and thus are as fast and small as accessing fields in a structure. Unless otherwise noted, the property setting methods such as [color\(n\)](#) or [label\(s\)](#) are also trivial inline functions, even if they change the widget's appearance. It is up to the user code to call [redraw\(\)](#) after these.

30.103.2 Member Enumeration Documentation

30.103.2.1 `anonymous enum` [protected]

flags possible values enumeration.

See [activate\(\)](#), [output\(\)](#), [visible\(\)](#), [changed\(\)](#), [set_visible_focus\(\)](#)

Enumerator:

INACTIVE the widget can't receive focus, and is disabled but potentially visible

INVISIBLE the widget is not drawn but can receive events

OUTPUT for output only

SHORTCUT_LABEL the label contains a shortcut we need to draw

CHANGED the widget value changed

VISIBLE_FOCUS accepts keyboard focus navigation if the widget can have the focus

COPIED_LABEL the widget label is internally copied, its destruction is handled by the widget

30.103.3 Constructor & Destructor Documentation

30.103.3.1 `FL_Widget::FL_Widget (int x, int y, int w, int h, FL_CString label = 0L)` [protected]

Creates a widget at the given position and size.

The [FL_Widget](#) is a protected constructor, but all derived widgets have a matching public constructor. It takes a value for [x\(\)](#), [y\(\)](#), [w\(\)](#), [h\(\)](#), and an optional value for [label\(\)](#).

Parameters:

- ← **x,y** the position of the widget relative to the enclosing window
- ← **w,h** size of the widget in pixels
- ← **label** optional text for the widget label

30.103.3.2 `FL_Widget::~~FL_Widget ()` [virtual]

Destroys the widget.

Destroys the widget, taking care of throwing focus before if any.

Destroying single widgets is not very common. You almost always want to destroy the parent group instead, which will destroy all of the child widgets and groups in that group.

Since:

FLTK 1.3, the widget's destructor removes the widget from its parent group, if it is member of a group.

Destruction removes the widget from any parent group! And groups when destroyed destroy all their children. This is convenient and fast.

30.103.4 Member Function Documentation

30.103.4.1 void Fl_Widget::activate ()

Activates the widget.

Changing this value will send FL_ACTIVATE to the widget if [active_r\(\)](#) is true.

See also:

[active\(\)](#), [active_r\(\)](#), [deactivate\(\)](#)

30.103.4.2 int Fl_Widget::active () const [inline]

Returns whether the widget is active.

Return values:

0 if the widget is inactive

See also:

[active_r\(\)](#), [activate\(\)](#), [deactivate\(\)](#)

30.103.4.3 int Fl_Widget::active_r () const

Returns whether the widget and all of its parents are active.

Return values:

0 if this or any of the parent widgets are inactive

See also:

[active\(\)](#), [activate\(\)](#), [deactivate\(\)](#)

30.103.4.4 void Fl_Widget::align (Fl_Align alignment) [inline]

Sets the label alignment.

This controls how the label is displayed next to or inside the widget. The default value is FL_ALIGN_CENTER, which centers the label inside the widget.

Parameters:

← *alignment* new label alignment

See also:

[align\(\)](#), [Fl_Align](#)

30.103.4.5 `Fl_Align Fl_Widget::align () const` [inline]

Gets the label alignment.

Returns:

label alignment

See also:

[label\(\)](#), [align\(Fl_Align\)](#), [Fl_Align](#)

Todo

This function should not take uchar as an argument. Apart from the fact that uchar is too short with only 8 bits, it does not provide type safety (in which case we don't need to declare Fl_Align an enum to begin with). NOTE* The current (FLTK 1.3) implementation (Dec 2008) is such that Fl_Align is (typedef'd to be) "unsigned" (int), but Fl_Widget's "align_" member variable is a bit field of 8 bits only !

30.103.4.6 `void Fl_Widget::argument (long v)` [inline]

Sets the current user data (long) argument that is passed to the callback function.

Todo

The user data value must be implemented using a *union* to avoid 64 bit machine incompatibilities.

30.103.4.7 `void Fl_Widget::box (Fl_Boxtype new_box)` [inline]

Sets the box type for the widget.

This identifies a routine that draws the background of the widget. See Fl_Boxtype for the available types. The default depends on the widget, but is usually FL_NO_BOX or FL_UP_BOX.

Parameters:

← *new_box* the new box type

See also:

[box\(\)](#), [Fl_Boxtype](#)

30.103.4.8 `Fl_Boxtype Fl_Widget::box () const` [inline]

Gets the box type of the widget.

Returns:

the current box type

See also:

[box\(Fl_Boxtype\)](#), [Fl_Boxtype](#)

30.103.4.9 void Fl_Widget::callback (Fl_Callback1 * *cb*, long *p* = 0) [inline]

Sets the current callback function for the widget.

Each widget has a single callback.

Parameters:

← *cb* new callback

← *p* user data

30.103.4.10 void Fl_Widget::callback (Fl_Callback0 * *cb*) [inline]

Sets the current callback function for the widget.

Each widget has a single callback.

Parameters:

← *cb* new callback

30.103.4.11 void Fl_Widget::callback (Fl_Callback * *cb*) [inline]

Sets the current callback function for the widget.

Each widget has a single callback.

Parameters:

← *cb* new callback

30.103.4.12 void Fl_Widget::callback (Fl_Callback * *cb*, void * *p*) [inline]

Sets the current callback function for the widget.

Each widget has a single callback.

Parameters:

← *cb* new callback

← *p* user data

30.103.4.13 `FL_Callback_p FL_Widget::callback () const` `[inline]`

Gets the current callback function for the widget.

Each widget has a single callback.

Returns:

current callback

30.103.4.14 `int FL_Widget::changed () const` `[inline]`

Checks if the widget value changed since the last callback.

"Changed" is a flag that is turned on when the user changes the value stored in the widget. This is only used by subclasses of [FL_Widget](#) that store values, but is in the base class so it is easier to scan all the widgets in a panel and [do_callback\(\)](#) on the changed ones in response to an "OK" button.

Most widgets turn this flag off when they do the callback, and when the program sets the stored value.

Return values:

0 if the value did not change

See also:

[set_changed\(\)](#), [clear_changed\(\)](#)

Reimplemented in [FL_Input_Choice](#).

30.103.4.15 `void FL_Widget::clear_changed ()` `[inline]`

Marks the value of the widget as unchanged.

See also:

[changed\(\)](#), [set_changed\(\)](#)

Reimplemented in [FL_Input_Choice](#).

30.103.4.16 `void FL_Widget::clear_damage (uchar c = 0)` `[inline]`

Clears the damage flags.

Damage flags are cleared when parts of the widget drawing is repaired.

Parameters:

← *c* bitmask of flags to clear

See also:

[damage\(uchar\)](#), [damage\(\)](#)

30.103.4.17 void FL_Widget::clear_output () [inline]

Sets a widget to accept input.

See also:

[set_output\(\)](#), [output\(\)](#)

30.103.4.18 void FL_Widget::clear_visible () [inline]

Hides the widget.

You must still redraw the parent to see a change in the window. Normally you want to use the [hide\(\)](#) method instead.

30.103.4.19 void FL_Widget::clear_visible_focus () [inline]

Disables keyboard focus navigation with this widget.

Normally, all widgets participate in keyboard focus navigation.

See also:

[set_visible_focus\(\)](#), [visible_focus\(\)](#), [visible_focus\(int\)](#)

30.103.4.20 void FL_Widget::color (unsigned *bg*, unsigned *sel*) [inline]

Sets the background and selection color of the widget.

The two color form sets both the background and selection colors.

Parameters:

← *bg* background color

← *sel* selection color

See also:

[color\(unsigned\)](#), [selection_color\(unsigned\)](#)

30.103.4.21 void FL_Widget::color (unsigned *bg*) [inline]

Sets the background color of the widget.

The color is passed to the box routine. The color is either an index into an internal table of RGB colors or an RGB color value generated using [fl_rgb_color\(\)](#).

The default for most widgets is FL_BACKGROUND_COLOR. Use [FL::set_color\(\)](#) to redefine colors in the color map.

Parameters:

← *bg* background color

See also:

[color\(\)](#), [color\(unsigned, unsigned\)](#), [selection_color\(unsigned\)](#)

30.103.4.22 FL_Color FL_Widget::color () const [inline]

Gets the background color of the widget.

Returns:

current background color

See also:

[color\(unsigned\)](#), [color\(unsigned, unsigned\)](#)

30.103.4.23 void FL_Widget::color2 (unsigned *a*) [inline]

For back compatibility only.

Deprecated

Use [selection_color\(unsigned\)](#) instead.

30.103.4.24 FL_Color FL_Widget::color2 () const [inline]

For back compatibility only.

Deprecated

Use [selection_color\(\)](#) instead.

30.103.4.25 int FL_Widget::contains (const FL_Widget * *w*) const

Checks if *w* is a child of this widget.

Parameters:

← *w* potential child widget

Returns:

Returns 1 if *w* is a child of this widget, or is equal to this widget. Returns 0 if *w* is NULL.

30.103.4.26 void FL_Widget::copy_label (FL_CString *new_label*)

Sets the current label.

Unlike [label\(\)](#), this method allocates a copy of the label string instead of using the original string pointer.

The internal copy will automatically be freed whenever you assign a new label or when the widget is destroyed.

Parameters:

← *new_label* the new label text

See also:

[label\(\)](#)

Reimplemented in [Fl_Window](#).

30.103.4.27 void Fl_Widget::damage (uchar *c*, int *x*, int *y*, int *w*, int *h*)

Sets the damage bits for an area inside the widget.

Setting damage bits will schedule the widget for the next redraw.

Parameters:

← *c* bitmask of flags to set

← *x,y,w,h* size of damaged area

See also:

[damage\(\)](#), [clear_damage\(uchar\)](#)

30.103.4.28 void Fl_Widget::damage (uchar *c*)

Sets the damage bits for the widget.

Setting damage bits will schedule the widget for the next redraw.

Parameters:

← *c* bitmask of flags to set

See also:

[damage\(\)](#), [clear_damage\(uchar\)](#)

30.103.4.29 uchar Fl_Widget::damage () const [inline]

Returns non-zero if [draw\(\)](#) needs to be called.

The damage value is actually a bit field that the widget subclass can use to figure out what parts to draw.

Returns:

a bitmap of flags describing the kind of damage to the widget

See also:

[damage\(uchar\)](#), [clear_damage\(uchar\)](#)

30.103.4.30 int Fl_Widget::damage_resize (int *X*, int *Y*, int *W*, int *H*)

Internal use only.

30.103.4.31 void FL_Widget::deactivate ()

Deactivates the widget.

Inactive widgets will be drawn "grayed out", e.g. with less contrast than the active widget. Inactive widgets will not receive any keyboard or mouse button events. Other events (including FL_ENTER, FL_MOVE, FL_LEAVE, FL_SHORTCUT, and others) will still be sent. A widget is only active if [active\(\)](#) is true on it *and all of its parents*.

Changing this value will send FL_DEACTIVATE to the widget if [active_r\(\)](#) is true.

Currently you cannot deactivate [FL_Window](#) widgets.

See also:

[activate\(\)](#), [active\(\)](#), [active_r\(\)](#)

Reimplemented in [FL_Repeat_Button](#).

30.103.4.32 void FL_Widget::default_callback (FL_Widget * *cb*, void * *d*) [static]

Sets the default callback for all widgets.

Sets the default callback, which puts a pointer to the widget on the queue returned by [FL::readqueue\(\)](#). You may want to call this from your own callback.

Parameters:

- ← *cb* the new callback
- ← *d* user data associated with that callback

See also:

[callback\(\)](#), [do_callback\(\)](#), [FL::readqueue\(\)](#)

30.103.4.33 void FL_Widget::deimage (FL_Image & *img*) [inline]

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the inactive state.

Parameters:

- ← *img* the new image for the deactivated widget

30.103.4.34 void FL_Widget::deimage (FL_Image * *img*) [inline]

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the inactive state.

Parameters:

- ← *img* the new image for the deactivated widget

30.103.4.35 `Fl_Image* Fl_Widget::deimage ()` `[inline]`

Gets the image that is used as part of the widget label.

This image is used when drawing the widget in the inactive state.

Returns:

the current image for the deactivated widget

30.103.4.36 `void Fl_Widget::do_callback (Fl_Widget * o, void * arg = 0)`

Calls the widget callback.

Causes a widget to invoke its callback function with arbitrary arguments.

Parameters:

- ← *o* call the callback with *o* as the widget argument
- ← *arg* use *arg* as the user data argument

See also:

[callback\(\)](#)

30.103.4.37 `void Fl_Widget::do_callback (Fl_Widget * o, long arg)` `[inline]`

Calls the widget callback.

Causes a widget to invoke its callback function with arbitrary arguments.

Parameters:

- ← *o* call the callback with *o* as the widget argument
- ← *arg* call the callback with *arg* as the user data argument

See also:

[callback\(\)](#)

30.103.4.38 `void Fl_Widget::do_callback ()` `[inline]`

Calls the widget callback.

Causes a widget to invoke its callback function with default arguments.

See also:

[callback\(\)](#)

30.103.4.39 virtual void FL_Widget::draw () [pure virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to FL_Widget* and call the virtual [draw\(\)](#) method.

```
FL_Widget *s = &scroll;           // scroll is an embedded FL_Scrollbar
s->draw();                         // calls FL_Scrollbar::draw()
```

Implemented in [FL_Adjuster](#), [FL_Box](#), [FL_Browser](#), [FL_Button](#), [FL_Cairo_Window](#), [FL_Chart](#), [FL_Choice](#), [FL_Clock_Output](#), [FL_Counter](#), [FL_Dial](#), [FL_File_Input](#), [FL_FormsBitmap](#), [FL_FormsPixmap](#), [FL_Free](#), [FL_Gl_Window](#), [FL_Group](#), [FL_Input](#), [FL_Light_Button](#), [FL_Menu_Bar](#), [FL_Menu_Button](#), [FL_Pack](#), [FL_Positioner](#), [FL_Progress](#), [FL_Return_Button](#), [FL_Roller](#), [FL_Scroll](#), [FL_Scrollbar](#), [FL_Slider](#), [FL_Tabs](#), [FL_Text_Display](#), [FL_Timer](#), [FL_Value_Input](#), [FL_Value_Output](#), [FL_Value_Slider](#), [FL_Window](#), and [FL_Glut_Window](#).

30.103.4.40 void FL_Widget::draw_box (FL_Boxtype t, int X, int Y, int W, int H, FL_Color c) const [protected]

Draws a box of type t, of color c at the position X,Y and size W,H.

30.103.4.41 void FL_Widget::draw_box (FL_Boxtype t, FL_Color c) const [protected]

Draws a box of type t, of color c at the widget's position and size.

30.103.4.42 void FL_Widget::draw_label (int X, int Y, int W, int H, FL_Align a) const

Draws the label in an arbitrary bounding box with an arbitrary alignment.

Anybody can call this to force the label to draw anywhere.

30.103.4.43 void FL_Widget::draw_label (int X, int Y, int W, int H) const [protected]

Draws the label in an arbitrary bounding box.

[draw\(\)](#) can use this instead of [draw_label\(void\)](#) to change the bounding box

30.103.4.44 void FL_Widget::draw_label (void) const [protected]

Draws the widget's label at the defined label position.

This is the normal call for a widget's [draw\(\)](#) method.

30.103.4.45 int FL_Widget::h () const [inline]

Gets the widget height.

Returns:

the height of the widget in pixels.

30.103.4.46 void FL_Widget::h (int v) [inline, protected]

Internal use only.

Use [position\(int,int\)](#), [size\(int,int\)](#) or [resize\(int,int,int,int\)](#) instead.

30.103.4.47 int FL_Widget::handle (int event) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

0 if the event was not used or understood

1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented in [FL_Adjuster](#), [FL_Box](#), [FL_Browser_](#), [FL_Button](#), [FL_Check_Browser](#), [FL_Choice](#), [FL_Clock](#), [FL_Counter](#), [FL_Dial](#), [FL_File_Input](#), [FL_Free](#), [FL_Group](#), [FL_Input](#), [FL_Light_Button](#), [FL_Menu_Bar](#), [FL_Menu_Button](#), [FL_Positioner](#), [FL_Repeat_Button](#), [FL_Return_Button](#), [FL_Roller](#), [FL_Scroll](#), [FL_Scrollbar](#), [FL_Slider](#), [FL_Spinner](#), [FL_Tabs](#), [FL_Text_Display](#), [FL_Text_Editor](#), [FL_Tile](#), [FL_Timer](#), [FL_Value_Input](#), [FL_Value_Output](#), [FL_Value_Slider](#), [FL_Window](#), and [FL_Glut_Window](#).

30.103.4.48 void FL_Widget::hide () [virtual]

Makes a widget invisible.

See also:

[show\(\)](#), [visible\(\)](#), [visible_r\(\)](#)

Reimplemented in [FL_Browser](#), [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Menu_Window](#), [FL_Overlay_Window](#), and [FL_Window](#).

30.103.4.49 void FL_Widget::image (FL_Image & *img*) [inline]

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the active state.

Parameters:

← *img* the new image for the label

30.103.4.50 void FL_Widget::image (FL_Image * *img*) [inline]

Sets the image to use as part of the widget label.

This image is used when drawing the widget in the active state.

Parameters:

← *img* the new image for the label

30.103.4.51 FL_Image* FL_Widget::image () [inline]

Gets the image that is used as part of the widget label.

This image is used when drawing the widget in the active state.

Returns:

the current image

30.103.4.52 int FL_Widget::inside (const FL_Widget * *w*) const [inline]

Checks if this widget is a child of *w*.

Returns 1 if this widget is a child of *w*, or is equal to *w*. Returns 0 if *w* is NULL.

Parameters:

← *w* the possible parent widget.

See also:

[contains\(\)](#)

30.103.4.53 void FL_Widget::label (FL_Labeltype *a*, const char * *b*) [inline]

Shortcut to set the label text and type in one call.

See also:

[label\(FL_CString\)](#), [labeltype\(FL_Labeltype\)](#)

30.103.4.54 void Fl_Widget::label (const char * *text*)

Sets the current label pointer.

The label is shown somewhere on or next to the widget. The passed pointer is stored unchanged in the widget (the string is *not* copied), so if you need to set the label to a formatted value, make sure the buffer is static, global, or allocated. The [copy_label\(\)](#) method can be used to make a copy of the label string automatically.

Parameters:

← *text* pointer to new label text

See also:

[copy_label\(\)](#)

Reimplemented in [Fl_Window](#).

30.103.4.55 const char* Fl_Widget::label () const [inline]

Gets the current label text.

Returns:

a pointer to the current label text

See also:

[label\(Fl_CString\)](#), [copy_label\(Fl_CString\)](#)

Reimplemented in [Fl_Window](#).

30.103.4.56 char Fl_Widget::label_shortcut (const char * *t*) [static]

Internal use only.

30.103.4.57 void Fl_Widget::labelcolor (unsigned *c*) [inline]

Sets the label color.

The default color is FL_FOREGROUND_COLOR.

Parameters:

← *c* the new label color

30.103.4.58 Fl_Color Fl_Widget::labelcolor () const [inline]

Gets the label color.

The default color is FL_FOREGROUND_COLOR.

Returns:

the current label color

30.103.4.59 void FL_Widget::labelfont (FL_Font *f*) [inline]

Sets the font to use.

Fonts are identified by indexes into a table. The default value uses a Helvetica typeface (Arial for Microsoft®Windows®). The function [FL::set_font\(\)](#) can define new typefaces.

Parameters:

← *f* the new font for the label

See also:

[FL_Font](#)

30.103.4.60 FL_Font FL_Widget::labelfont () const [inline]

Gets the font to use.

Fonts are identified by indexes into a table. The default value uses a Helvetica typeface (Arial for Microsoft®Windows®). The function [FL::set_font\(\)](#) can define new typefaces.

Returns:

current font used by the label

See also:

[FL_Font](#)

30.103.4.61 void FL_Widget::labelsize (FL_Fontsize *pix*) [inline]

Sets the font size in pixels.

Parameters:

← *pix* the new font size

See also:

[FL_Fontsize labelsize\(\)](#)

30.103.4.62 FL_Fontsize FL_Widget::labelsize () const [inline]

Gets the font size in pixels.

The default size is 14 pixels.

Returns:

the current font size

30.103.4.63 void Fl_Widget::labeltype (Fl_Labeltype *a*) [inline]

Sets the label type.

The label type identifies the function that draws the label of the widget. This is generally used for special effects such as embossing or for using the [label\(\)](#) pointer as another form of data such as an icon. The value FL_NORMAL_LABEL prints the label as plain text.

Parameters:

← *a* new label type

See also:

[Fl_Labeltype](#)

30.103.4.64 Fl_Labeltype Fl_Widget::labeltype () const [inline]

Gets the label type.

Returns:

the current label type.

See also:

[Fl_Labeltype](#)

30.103.4.65 void Fl_Widget::measure_label (int & *ww*, int & *hh*) [inline]

Sets width *ww* and height *hh* accordingly with the label size.

Labels with images will return [w\(\)](#) and [h\(\)](#) of the image.

30.103.4.66 int Fl_Widget::output () const [inline]

Returns if a widget is used for output only.

[output\(\)](#) means the same as [!active\(\)](#) except it does not change how the widget is drawn. The widget will not receive any events. This is useful for making scrollbars or buttons that work as displays rather than input devices.

Return values:

0 if the widget is used for input and output

See also:

[set_output\(\)](#), [clear_output\(\)](#)

30.103.4.67 void FL_Widget::parent (FL_Group * p) [inline]

Internal use only - "for hacks only".

It is **STRONGLY recommended** not to use this method, because it short-circuits FL_Group's normal widget adding and removing methods, if the widget is already a child widget of another [FL_Group](#).

Use [FL_Group::add\(FL_Widget*\)](#) and/or [FL_Group::remove\(FL_Widget*\)](#) instead.

30.103.4.68 FL_Group* FL_Widget::parent () const [inline]

Returns a pointer to the parent widget.

Usually this is a [FL_Group](#) or [FL_Window](#).

Return values:

NULL if the widget has no parent

See also:

[FL_Group::add\(FL_Widget*\)](#)

30.103.4.69 void FL_Widget::position (int X, int Y) [inline]

Repositions the window or widget.

position(X, Y) is a shortcut for [resize\(X, Y, w\(\), h\(\)\)](#).

Parameters:

← *X,Y* new position relative to the parent window

See also:

[resize\(int,int,int,int\)](#), [size\(int,int\)](#)

Reimplemented in [FL_Input_](#).

30.103.4.70 void FL_Widget::redraw ()

Schedules the drawing of the widget.

Marks the widget as needing its [draw\(\)](#) routine called.

30.103.4.71 void FL_Widget::redraw_label ()

Schedules the drawing of the label.

Marks the widget or the parent as needing a redraw for the label area of a widget.

30.103.4.72 void FL_Widget::resize (int *x*, int *y*, int *w*, int *h*) [virtual]

Changes the size or position of the widget.

This is a virtual function so that the widget may implement its own handling of resizing. The default version does *not* call the [redraw\(\)](#) method, but instead relies on the parent widget to do so because the parent may know a faster way to update the display, such as scrolling from the old position.

Some window managers under X11 call [resize\(\)](#) a lot more often than needed. Please verify that the position or size of a widget did actually change before doing any extensive calculations.

[position\(X, Y\)](#) is a shortcut for [resize\(X, Y, w\(\), h\(\)\)](#), and [size\(W, H\)](#) is a shortcut for [resize\(x\(\), y\(\), W, H\)](#).

Parameters:

← *x,y* new position relative to the parent window

← *w,h* new size

See also:

[position\(int,int\)](#), [size\(int,int\)](#)

Reimplemented in [FL_Browser_](#), [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Group](#), [FL_Help_View](#), [FL_Input_](#), [FL_Input_Choice](#), [FL_Overlay_Window](#), [FL_Scroll](#), [FL_Spinner](#), [FL_Text_Display](#), [FL_Tile](#), [FL_Value_Input](#), and [FL_Window](#).

30.103.4.73 void FL_Widget::selection_color (unsigned *a*) [inline]

Sets the selection color.

The selection color is defined for Forms compatibility and is usually used to color the widget when it is selected, although some widgets use this color for other purposes. You can set both colors at once with [color\(unsigned bg, unsigned sel\)](#).

Parameters:

← *a* the new selection color

See also:

[selection_color\(\)](#), [color\(unsigned, unsigned\)](#)

30.103.4.74 FL_Color FL_Widget::selection_color () const [inline]

Gets the selection color.

Returns:

the current selection color

See also:

[selection_color\(unsigned\)](#), [color\(unsigned, unsigned\)](#)

30.103.4.75 void FL_Widget::set_changed () [inline]

Marks the value of the widget as changed.

See also:

[changed\(\)](#), [clear_changed\(\)](#)

Reimplemented in [FL_Input_Choice](#).

30.103.4.76 void FL_Widget::set_output () [inline]

Sets a widget to output only.

See also:

[output\(\)](#), [clear_output\(\)](#)

30.103.4.77 void FL_Widget::set_visible () [inline]

Makes the widget visible.

You must still redraw the parent widget to see a change in the window. Normally you want to use the [show\(\)](#) method instead.

30.103.4.78 void FL_Widget::set_visible_focus () [inline]

Enables keyboard focus navigation with this widget.

Note, however, that this will not necessarily mean that the widget will accept focus, but for widgets that can accept focus, this method enables it if it has been disabled.

See also:

[visible_focus\(\)](#), [clear_visible_focus\(\)](#), [visible_focus\(int\)](#)

30.103.4.79 void FL_Widget::show () [virtual]

Makes a widget visible.

An invisible widget never gets redrawn and does not get events. The [visible\(\)](#) method returns true if the widget is set to be visible. The [visible_r\(\)](#) method returns true if the widget and all of its parents are visible. A widget is only visible if [visible\(\)](#) is true on it *and all of its parents*.

Changing it will send FL_SHOW or FL_HIDE events to the widget. *Do not change it if the parent is not visible, as this will send false FL_SHOW or FL_HIDE events to the widget.* [redraw\(\)](#) is called if necessary on this or the parent.

See also:

[hide\(\)](#), [visible\(\)](#), [visible_r\(\)](#)

Reimplemented in [FL_Browser](#), [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Menu_Window](#), [FL_Overlay_Window](#), [FL_Single_Window](#), and [FL_Window](#).

30.103.4.80 void Fl_Widget::size (int *W*, int *H*) [inline]

Changes the size of the widget.

size(*W*, *H*) is a shortcut for `resize(x(), y(), W, H)`.

Parameters:

← *W*, *H* new size

See also:

[position\(int,int\)](#), [resize\(int,int,int,int\)](#)

Reimplemented in [Fl_Browser](#), [Fl_Chart](#), [Fl_Help_View](#), [Fl_Input_](#), and [Fl_Menu_](#).

30.103.4.81 int Fl_Widget::take_focus ()

Gives the widget the keyboard focus.

Tries to make this widget be the [Fl::focus\(\)](#) widget, by first sending it an FL_FOCUS event, and if it returns non-zero, setting [Fl::focus\(\)](#) to this widget. You should use this method to assign the focus to a widget.

Returns:

true if the widget accepted the focus.

30.103.4.82 int Fl_Widget::takeevents () const [inline]

Returns if the widget is able to take events.

This is the same as `(active\(\) && !output\(\) && visible\(\))` but is faster.

Return values:

0 if the widget takes no events

30.103.4.83 int Fl_Widget::test_shortcut (const char * *t*) [static]

Internal use only.

30.103.4.84 int Fl_Widget::test_shortcut ()

Internal use only.

Reimplemented in [Fl_Menu_](#).

30.103.4.85 void Fl_Widget::tooltip (const char * *t*)

Sets the current tooltip text.

Sets a string of text to display in a popup tooltip window when the user hovers the mouse over the widget. The string is *not* copied, so make sure any formatted string is stored in a static, global, or allocated buffer.

If no tooltip is set, the tooltip of the parent is inherited. Setting a tooltip for a group and setting no tooltip for a child will show the group's tooltip instead. To avoid this behavior, you can set the child's tooltip to an empty string ("").

Parameters:

← *t* new tooltip

30.103.4.86 `const char* FL_Widget::tooltip () const` [inline]

Gets the current tooltip text.

Returns:

a pointer to the tooltip text or NULL

30.103.4.87 `void FL_Widget::type (uchar t)` [inline]

Sets the widget type.

This is used for Forms compatibility.

Reimplemented in [FL_Spinner](#).

30.103.4.88 `uchar FL_Widget::type () const` [inline]

Gets the widget type.

Returns the widget type value, which is used for Forms compatibility and to simulate RTTI.

Todo

Explain "simulate RTTI" (currently only used to decide if a widget is a window, i.e. `type()>=FL_WINDOW` ?). Is `type()` really used in a way that ensures "Forms compatibility" ?

Reimplemented in [FL_Spinner](#).

30.103.4.89 `void FL_Widget::user_data (void *v)` [inline]

Sets the user data for this widget.

Sets the new user data (void *) argument that is passed to the callback function.

Parameters:

← *v* new user data

30.103.4.90 `void* FL_Widget::user_data () const` [inline]

Gets the user data for this widget.

Gets the current user data (void *) argument that is passed to the callback function.

Returns:

user data as a pointer

30.103.4.91 int Fl_Widget::visible () const [inline]

Returns whether a widget is visible.

Return values:

0 if the widget is not drawn and hence invisible.

See also:

[show\(\)](#), [hide\(\)](#), [visible_r\(\)](#)

30.103.4.92 int Fl_Widget::visible_focus () [inline]

Checks whether this widget has a visible focus.

Return values:

0 if this widget has no visible focus.

See also:

[visible_focus\(int\)](#), [set_visible_focus\(\)](#), [clear_visible_focus\(\)](#)

30.103.4.93 void Fl_Widget::visible_focus (int v) [inline]

Modifies keyboard focus navigation.

Parameters:

← v set or clear visible focus

See also:

[set_visible_focus\(\)](#), [clear_visible_focus\(\)](#), [visible_focus\(\)](#)

30.103.4.94 int Fl_Widget::visible_r () const

Returns whether a widget and all its parents are visible.

Return values:

0 if the widget or any of its parents are invisible.

See also:

[show\(\)](#), [hide\(\)](#), [visible\(\)](#)

30.103.4.95 `int FL_Widget::w () const` `[inline]`

Gets the widget width.

Returns:

the width of the widget in pixels.

30.103.4.96 `void FL_Widget::w (int v)` `[inline, protected]`

Internal use only.

Use [position\(int,int\)](#), [size\(int,int\)](#) or [resize\(int,int,int,int\)](#) instead.

30.103.4.97 `void FL_Widget::when (uchar i)` `[inline]`

Sets the flags used to decide when a callback is called.

This controls when callbacks are done. The following values are useful, the default value is FL_WHEN_RELEASE:

- 0: The callback is not done, but [changed\(\)](#) is turned on.
- FL_WHEN_CHANGED: The callback is done each time the text is changed by the user.
- FL_WHEN_RELEASE: The callback will be done when this widget loses the focus, including when the window is unmapped. This is a useful value for text fields in a panel where doing the callback on every change is wasteful. However the callback will also happen if the mouse is moved out of the window, which means it should not do anything visible (like pop up an error message). You might do better setting this to zero, and scanning all the items for [changed\(\)](#) when the OK button on a panel is pressed.
- FL_WHEN_ENTER_KEY: If the user types the Enter key, the entire text is selected, and the callback is done if the text has changed. Normally the Enter key will navigate to the next field (or insert a newline for a [FL_Multiline_Input](#)) - this changes the behavior.
- FL_WHEN_ENTER_KEY|FL_WHEN_NOT_CHANGED: The Enter key will do the callback even if the text has not changed. Useful for command fields. [FL_Widget::when\(\)](#) is a set of bitflags used by subclasses of [FL_Widget](#) to decide when to do the callback.

If the value is zero then the callback is never done. Other values are described in the individual widgets. This field is in the base class so that you can scan a panel and [do_callback\(\)](#) on all the ones that don't do their own callbacks in response to an "OK" button.

Parameters:

← *i* set of flags

30.103.4.98 `FL_Widget::when () const` `[inline]`

Returns the conditions under which the callback is called.

You can set the flags with [when\(uchar\)](#), the default value is FL_WHEN_RELEASE.

Returns:

set of flags

See also:

[when\(uchar\)](#)

30.103.4.99 `Fl_Window * Fl_Widget::window () const`

Returns a pointer to the primary [Fl_Window](#) widget.

Return values:

NULL if no window is associated with this widget.

Note:

for an [Fl_Window](#) widget, this returns its *parent* window (if any), not *this* window.

30.103.4.100 `int Fl_Widget::x () const` [inline]

Gets the widget position in its window.

Returns:

the x position relative to the window

30.103.4.101 `void Fl_Widget::x (int v)` [inline, protected]

Internal use only.

Use [position\(int,int\)](#), [size\(int,int\)](#) or [resize\(int,int,int,int\)](#) instead.

30.103.4.102 `int Fl_Widget::y () const` [inline]

Gets the widget position in its window.

Returns:

the y position relative to the window

30.103.4.103 `void Fl_Widget::y (int v)` [inline, protected]

Internal use only.

Use [position\(int,int\)](#), [size\(int,int\)](#) or [resize\(int,int,int,int\)](#) instead.

The documentation for this class was generated from the following files:

- `Fl_Widget.H`

- [Fl.cxx](#)
- [fl_boxttype.cxx](#)
- [fl_labeltype.cxx](#)
- [fl_shortcut.cxx](#)
- [Fl_Tooltip.cxx](#)
- [Fl_Widget.cxx](#)
- [Fl_Window.cxx](#)

30.104 FL_Widget_Tracker Class Reference

This class should be used to control safe widget deletion.

```
#include <Fl.H>
```

Public Member Functions

- `int deleted ()`
returns 1, if the watched widget has been deleted.
- `int exists ()`
returns 1, if the watched widget exists (has not been deleted).
- `FL_Widget_Tracker (FL_Widget *wi)`
The constructor adds a widget to the watch list.
- `FL_Widget * widget ()`
returns a pointer to the watched widget.
- `~FL_Widget_Tracker ()`
The destructor removes a widget from the watch list.

30.104.1 Detailed Description

This class should be used to control safe widget deletion.

You can use an `FL_Widget_Tracker` object to watch another widget, if you need to know, if this widget has been deleted during a callback.

This simplifies the use of the "safe widget deletion" methods `FL::watch_widget_pointer()` and `FL::release_widget_pointer()` and makes their use more reliable, because the destructor automatically releases the widget pointer from the widget watch list.

It is intended to be used as an automatic (local/stack) variable, such that the automatic destructor is called when the object's scope is left. This ensures that no stale widget pointers are left in the widget watch list (see example below).

You can also create `FL_Widget_Tracker` objects with `new`, but then it is your responsibility to delete the object (and thus remove the widget pointer from the watch list) when it is not needed any more.

Example:

```
int MyClass::handle (int event) {  
  
    if (...) {  
        FL_Widget_Tracker wp(this);    // watch myself  
        do_callback();                  // call the callback  
  
        if (wp.deleted()) return 1;     // exit, if deleted  
  
        // Now we are sure that the widget has not been deleted.  
        // It is safe to access the widget  
  
        clear_changed();                // access the widget  
    }  
}
```

```
    }  
}
```

30.104.2 Member Function Documentation

30.104.2.1 int FL_Widget_Tracker::deleted () [inline]

returns 1, if the watched widget has been deleted.

This is a convenience method. You can also use something like

if (wp.widget() == 0) // ...

where **wp** is an [FL_Widget_Tracker](#) object.

30.104.2.2 int FL_Widget_Tracker::exists () [inline]

returns 1, if the watched widget exists (has not been deleted).

This is a convenience method. You can also use something like

if (wp.widget() != 0) // ...

where **wp** is an [FL_Widget_Tracker](#) object.

30.104.2.3 FL_Widget* FL_Widget_Tracker::widget () [inline]

returns a pointer to the watched widget.

This pointer is NULL, if the widget has been deleted.

The documentation for this class was generated from the following files:

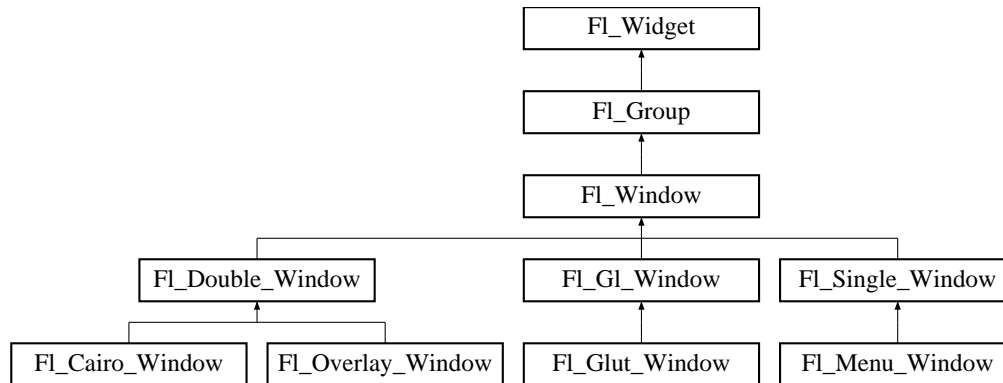
- FL.H
- FL.cxx

30.105 FL_Window Class Reference

This widget produces an actual window.

```
#include <FL_Window.H>
```

Inheritance diagram for FL_Window::



Public Member Functions

- int [border](#) () const
See int [FL_Window::border\(int\)](#).
- void [border](#) (int b)
Gets or sets whether or not the window manager border is around the window.
- void [clear_border](#) ()
Fast inline function to turn the border off.
- void [copy_label](#) (const char *a)
Sets the current label.
- void [cursor](#) (FL_Cursor, [FL_Color](#)=FL_BLACK, [FL_Color](#)=FL_WHITE)
Changes the cursor for this window.
- void [default_cursor](#) (FL_Cursor, [FL_Color](#)=FL_BLACK, [FL_Color](#)=FL_WHITE)
Sets the default window cursor as well as its color.
- [FL_Window](#) (int x, int y, int w, int h, const char *title=0)
Creates a window from the given position, size and title.
- [FL_Window](#) (int w, int h, const char *title=0)
Creates a window from the given size and title.
- void [free_position](#) ()
Undoes the effect of a previous [resize\(\)](#) or [show\(\)](#) so that the next time [show\(\)](#) is called the window manager is free to position the window.

- void `fullscreen` ()
Makes the window completely fill the screen, without any window manager border visible.
- void `fullscreen_off` (int, int, int, int)
Turns off any side effects of `fullscreen()` and does `resize(x,y,w,h)`.
- virtual int `handle` (int)
Handles the specified event.
- virtual void `hide` ()
Remove the window from the screen.
- void `hotspot` (const `FL_Widget` &p, int offscreen=0)
See void `FL_Window::hotspot(int x, int y, int offscreen = 0)`.
- void `hotspot` (const `FL_Widget` *, int offscreen=0)
See void `FL_Window::hotspot(int x, int y, int offscreen = 0)`.
- void `hotspot` (int x, int y, int offscreen=0)
Position the window so that the mouse is pointing at the given position, or at the center of the given widget, which may be the window itself.
- void `icon` (const void *ic)
Sets the current icon window target dependent data.
- const void * `icon` () const
Gets the current icon window target dependent data.
- void `iconize` ()
Iconifies the window.
- void `iconlabel` (const char *)
Sets the icon label.
- const char * `iconlabel` () const
See void `FL_Window::iconlabel(const char)`.*
- void `label` (const char *label, const char *iconlabel)
Gets or sets the icon label.
- void `label` (const char *)
Sets the window title bar label.
- const char * `label` () const
See void `FL_Window::label(const char)`.*
- void `make_current` ()
Sets things up so that the drawing functions in `<FL/fl_draw.H>` will go into this window.
- int `modal` () const

Returns true if this window is modal.

- `int non_modal () const`

Returns true if this window is modal or non-modal.

- `int override () const`

Returns non zero if `FL_OVERRIDE` flag is set, 0 otherwise.

- `virtual void resize (int, int, int, int)`

Changes the size and position of the window.

- `void set_modal ()`

A "modal" window, when `shown()`, will prevent any events from being delivered to other windows in the same program, and will also remain on top of the other windows (if the X window manager supports the "transient for" property).

- `void set_non_modal ()`

A "non-modal" window (terminology borrowed from Microsoft Windows) acts like a `modal()` one in that it remains on top, but it has no effect on event delivery.

- `void set_override ()`

Activate the flags `FL_NOBORDER|FL_OVERRIDE`.

- `void show (int, char **)`

See virtual void `FL_Window::show()`.

- `virtual void show ()`

Put the window on the screen.

- `int shown ()`

Returns non-zero if `show()` has been called (but not `hide()`).

- `void size_range (int a, int b, int c=0, int d=0, int e=0, int f=0, int g=0)`

Set the allowable range the user can resize this window to.

- `int x_root () const`

Gets the x position of the window on the screen.

- `void xclass (const char *c)`

A string used to tell the system what type of window this is.

- `const char * xclass () const`

See void `FL_Window::xclass(const char)`.*

- `int y_root () const`

Gets the y position of the window on the screen.

- `virtual ~FL_Window ()`

The destructor also deletes all the children.

Static Public Member Functions

- static [FL_Window](#) * [current](#) ()
Returns the last window that was made current.
- static void [default_callback](#) ([FL_Window](#) *, void *v)
Back compatibility: Sets the default callback v for win to call on close event.

Protected Member Functions

- virtual void [draw](#) ()
Draws the widget.
- virtual void [flush](#) ()
Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).

Static Protected Attributes

- static [FL_Window](#) * [current_](#)
Stores the last window that was made current.

Friends

- class [FL_X](#)

30.105.1 Detailed Description

This widget produces an actual window.

This can either be a main window, with a border and title and all the window management controls, or a "subwindow" inside a window. This is controlled by whether or not the window has a [parent\(\)](#).

Once you create a window, you usually add children [FL_Widget](#) 's to it by using `window->add(child)` for each new widget. See [FL_Group](#) for more information on how to add and remove children.

There are several subclasses of [FL_Window](#) that provide double-buffering, overlay, menu, and OpenGL support.

The window's callback is done if the user tries to close a window using the window manager and [FL::modal\(\)](#) is zero or equal to the window. [FL_Window](#) has a default callback that calls [FL_Window::hide\(\)](#).

30.105.2 Constructor & Destructor Documentation

30.105.2.1 [FL_Window::FL_Window](#) (int *w*, int *h*, const char * *title* = 0)

Creates a window from the given size and title.

If [FL_Group::current\(\)](#) is not NULL, the window is created as a subwindow of the parent window.

The first form of the constructor creates a top-level window and asks the window manager to position the window. The second form of the constructor either creates a subwindow or a top-level window at the specified location (x,y) , subject to window manager configuration. If you do not specify the position of the window, the window manager will pick a place to show the window or allow the user to pick a location. Use `position(x,y)` or `hotspot()` before calling `show()` to request a position on the screen. See `FL_Window::resize()` for some more details on positioning windows.

Top-level windows initially have `visible()` set to 0 and `parent()` set to NULL. Subwindows initially have `visible()` set to 1 and `parent()` set to the parent window pointer.

`FL_Widget::box()` defaults to FL_FLAT_BOX. If you plan to completely fill the window with children widgets you should change this to FL_NO_BOX. If you turn the window border off you may want to change this to FL_UP_BOX.

30.105.2.2 `FL_Window::FL_Window (int x, int y, int w, int h, const char * title = 0)`

Creates a window from the given position, size and title.

See `FL_Window::FL_Window(int w, int h, const char *title = 0)`

30.105.2.3 `FL_Window::~~FL_Window ()` [virtual]

The destructor *also deletes all the children*.

This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `FL_Window` and all of it's children can be automatic (local) variables, but you must declare the `FL_Window` *first* so that it is destroyed last.

30.105.3 Member Function Documentation

30.105.3.1 `void FL_Window::border (int b)`

Gets or sets whether or not the window manager border is around the window.

The default value is true. `border(n)` can be used to turn the border on and off, and returns non-zero if the value has been changed. *Under most X window managers this does not work after `show()` has been called, although SGI's 4DWM does work.*

30.105.3.2 `void FL_Window::clear_border ()` [inline]

Fast inline function to turn the border off.

It only works before `show()` is called.

30.105.3.3 `void FL_Window::copy_label (const char * new_label)`

Sets the current label.

Unlike `label()`, this method allocates a copy of the label string instead of using the original string pointer.

The internal copy will automatically be freed whenever you assign a new label or when the widget is destroyed.

Parameters:

← *new_label* the new label text

See also:

[label\(\)](#)

Reimplemented from [Fl_Widget](#).

30.105.3.4 Fl_Window * Fl_Window::current () [static]

Returns the last window that was made current.

See also:

[Fl_Window::make_current\(\)](#)

Reimplemented from [Fl_Group](#).

30.105.3.5 void Fl_Window::cursor (Fl_Cursor c, Fl_Color fg = FL_BLACK, Fl_Color bg = FL_WHITE)

Changes the cursor for this window.

This always calls the system, if you are changing the cursor a lot you may want to keep track of how you set it in a static variable and call this only if the new cursor is different.

The type Fl_Cursor is an enumeration defined in [<Enumerations.H>](#). (Under X you can get any XC_ cursor value by passing Fl_Cursor((XC_foo/2)+1)). The colors only work on X, they are not implemented on WIN32.

For back compatibility only.

30.105.3.6 void Fl_Window::default_cursor (Fl_Cursor c, Fl_Color fg = FL_BLACK, Fl_Color bg = FL_WHITE)

Sets the default window cursor as well as its color.

For back compatibility only.

30.105.3.7 void Fl_Window::draw () [protected, virtual]

Draws the widget.

Never call this function directly. FLTK will schedule redrawing whenever needed. If your widget must be redrawn as soon as possible, call [redraw\(\)](#) instead.

Override this function to draw your own widgets.

If you ever need to call another widget's draw method *from within your own draw method*, e.g. for an embedded scrollbar, you must downcast the embedded widget's pointer to Fl_Widget* and call the virtual [draw\(\)](#) method.

```
Fl_Widget *s = &scroll;           // scroll is an embedded Fl_Scrollbar
s->draw();                         // calls Fl_Scrollbar::draw()
```

Reimplemented from [FL_Group](#).

Reimplemented in [FL_Cairo_Window](#), [FL_Gl_Window](#), and [FL_Glut_Window](#).

30.105.3.8 void FL_Window::flush () [protected, virtual]

Forces the window to be drawn, this window is also made current and calls [draw\(\)](#).

Reimplemented in [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Menu_Window](#), [FL_Overlay_Window](#), and [FL_Single_Window](#).

30.105.3.9 void FL_Window::fullscreen ()

Makes the window completely fill the screen, without any window manager border visible.

You must use [fullscreen_off\(\)](#) to undo this. This may not work with all window managers.

30.105.3.10 int FL_Window::handle (int *event*) [virtual]

Handles the specified event.

You normally don't call this method directly, but instead let FLTK do it when the user interacts with the widget.

When implemented in a widget, this function must return 0 if the widget does not use the event or 1 otherwise.

Most of the time, you want to call the inherited [handle\(\)](#) method in your overridden method so that you don't short-circuit events that you don't handle. In this last case you should return the callee retval.

Parameters:

← *event* the kind of event received

Return values:

- 0 if the event was not used or understood
- 1 if the event was used and can be deleted

See also:

[FL_Event](#)

Reimplemented from [FL_Group](#).

Reimplemented in [FL_Glut_Window](#).

30.105.3.11 void FL_Window::hide () [virtual]

Remove the window from the screen.

If the window is already hidden or has not been shown then this does nothing and is harmless.

Reimplemented from [FL_Widget](#).

Reimplemented in [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Menu_Window](#), and [FL_Overlay_Window](#).

30.105.3.12 void FL_Window::hotspot (int x, int y, int offscreen = 0)

Position the window so that the mouse is pointing at the given position, or at the center of the given widget, which may be the window itself.

If the optional offscreen parameter is non-zero, then the window is allowed to extend off the screen (this does not work with some X window managers).

See also:

[position\(\)](#)

30.105.3.13 void FL_Window::iconize ()

Iconifies the window.

If you call this when [shown\(\)](#) is false it will [show\(\)](#) it as an icon. If the window is already iconified this does nothing.

Call [show\(\)](#) to restore the window.

When a window is iconified/restored (either by these calls or by the user) the [handle\(\)](#) method is called with FL_HIDE and FL_SHOW events and [visible\(\)](#) is turned on and off.

There is no way to control what is drawn in the icon except with the string passed to [FL_Window::xclass\(\)](#). You should not rely on window managers displaying the icons.

30.105.3.14 void FL_Window::iconlabel (const char * *iname*)

Sets the icon label.

30.105.3.15 void FL_Window::label (const char * *label*, const char * *iconlabel*)

Gets or sets the icon label.

30.105.3.16 void FL_Window::label (const char * *name*)

Sets the window title bar label.

Reimplemented from [FL_Widget](#).

30.105.3.17 void FL_Window::make_current ()

Sets things up so that the drawing functions in [<FL/fl_draw.H>](#) will go into this window.

This is useful for incremental update of windows, such as in an idle callback, which will make your program behave much better if it draws a slow graphic. **Danger: incremental update is very hard to debug and maintain!**

This method only works for the [FL_Window](#) and [FL_Gl_Window](#) derived classes.

Reimplemented in [FL_Gl_Window](#), [FL_Single_Window](#), and [FL_Glut_Window](#).

30.105.3.18 `int FL_Window::modal () const` [inline]

Returns true if this window is modal.

30.105.3.19 `int FL_Window::non_modal () const` [inline]

Returns true if this window is modal or non-modal.

30.105.3.20 `int FL_Window::override () const` [inline]

Returns non zero if FL_OVERRIDE flag is set, 0 otherwise.

30.105.3.21 `virtual void FL_Window::resize (int, int, int, int)` [virtual]

Changes the size and position of the window.

If [shown\(\)](#) is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If [shown\(\)](#) is false, the size and position are used when [show\(\)](#) is called. See [FL_Group](#) for the effect of resizing on the child widgets.

You can also call the [FL_Widget](#) methods [size\(x,y\)](#) and [position\(w,h\)](#), which are inline wrappers for this virtual function.

A top-level window can not force, but merely suggest a position and size to the operating system. The window manager may not be willing or able to display a window at the desired position or with the given dimensions. It is up to the application developer to verify window parameters after the resize request.

Reimplemented from [FL_Group](#).

Reimplemented in [FL_Double_Window](#), [FL_Gl_Window](#), and [FL_Overlay_Window](#).

30.105.3.22 `void FL_Window::set_modal ()` [inline]

A "modal" window, when [shown\(\)](#), will prevent any events from being delivered to other windows in the same program, and will also remain on top of the other windows (if the X window manager supports the "transient for" property).

Several modal windows may be shown at once, in which case only the last one shown gets events. You can See which window (if any) is modal by calling [FL::modal\(\)](#).

30.105.3.23 `void FL_Window::set_non_modal ()` [inline]

A "non-modal" window (terminology borrowed from Microsoft Windows) acts like a [modal\(\)](#) one in that it remains on top, but it has no effect on event delivery.

There are *three* states for a window: modal, non-modal, and normal.

30.105.3.24 `virtual void FL_Window::show ()` [virtual]

Put the window on the screen.

Usually this has the side effect of opening the display. The second form is used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call [show\(\)](#) at any time, even if the window is already up. It also means that [show\(\)](#) serves the purpose of [raise\(\)](#) in other toolkits.

Reimplemented from [FL_Widget](#).

Reimplemented in [FL_Double_Window](#), [FL_Gl_Window](#), [FL_Menu_Window](#), [FL_Overlay_Window](#), and [FL_Single_Window](#).

30.105.3.25 int FL_Window::shown () [inline]

Returns non-zero if [show\(\)](#) has been called (but not [hide\(\)](#)).

You can tell if a window is iconified with (w->[shown\(\)](#) &!w->[visible\(\)](#)).

30.105.3.26 void FL_Window::size_range (int a, int b, int c = 0, int d = 0, int e = 0, int f = 0, int g = 0) [inline]

Set the allowable range the user can resize this window to.

This only works for top-level windows.

- minw and minh are the smallest the window can be. Either value must be greater than 0.
- maxw and maxh are the largest the window can be. If either is *equal* to the minimum then you cannot resize in that direction. If either is zero then FLTK picks a maximum size in that direction such that the window will fill the screen.
- dw and dh are size increments. The window will be constrained to widths of minw + N * dw, where N is any non-negative integer. If these are less or equal to 1 they are ignored. (this is ignored on WIN32)
- aspect is a flag that indicates that the window should preserve it's aspect ratio. This only works if both the maximum and minimum have the same aspect ratio. (ignored on WIN32 and by many X window managers)

If this function is not called, FLTK tries to figure out the range from the setting of [resizable\(\)](#):

- If [resizable\(\)](#) is NULL (this is the default) then the window cannot be resized and the resize border and max-size control will not be displayed for the window.
- If either dimension of [resizable\(\)](#) is less than 100, then that is considered the minimum size. Otherwise the [resizable\(\)](#) has a minimum size of 100.
- If either dimension of [resizable\(\)](#) is zero, then that is also the maximum size (so the window cannot resize in that direction).

It is undefined what happens if the current size does not fit in the constraints passed to [size_range\(\)](#).

30.105.3.27 void FL_Window::xclass (const char * c) [inline]

A string used to tell the system what type of window this is.

Mostly this identifies the picture to draw in the icon. *Under X, this is turned into a XA_WM_CLASS pair by truncating at the first non-alphanumeric character and capitalizing the first character, and the second*

one if the first is 'x'. Thus "foo" turns into "foo, Foo", and "xprog.1" turns into "xprog, XProg". This only works if called *before* calling [show\(\)](#).

Under Microsoft Windows this string is used as the name of the WNDCLASS structure, though it is not clear if this can have any visible effect. The passed pointer is stored unchanged. The string is not copied.

30.105.4 Member Data Documentation

30.105.4.1 `Fl_Window* Fl_Window::current_` [static, protected]

Stores the last window that was made current.

See `current()` const

Reimplemented from [Fl_Group](#).

The documentation for this class was generated from the following files:

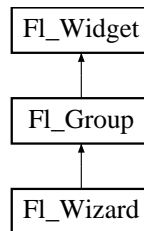
- `Fl_Window.H`
- `Fl.cxx`
- `Fl_arg.cxx`
- `fl_cursor.cxx`
- `Fl_Window.cxx`
- `Fl_Window_fullscreen.cxx`
- `Fl_Window_hotspot.cxx`
- `Fl_Window_iconize.cxx`

30.106 Fl_Wizard Class Reference

This widget is based off the [Fl_Tabs](#) widget, but instead of displaying tabs it only changes "tabs" under program control.

```
#include <Fl_Wizard.H>
```

Inheritance diagram for Fl_Wizard::



Public Member Functions

- [Fl_Wizard](#) (int, int, int, int, const char *=0)
The constructor creates the [Fl_Wizard](#) widget at the specified position and size.
- void [next](#) ()
This method shows the next child of the wizard.
- void [prev](#) ()
Shows the previous child.
- void [value](#) (Fl_Widget *)
Sets the child widget that is visible.
- Fl_Widget * [value](#) ()
Gets the current visible child widget.

30.106.1 Detailed Description

This widget is based off the [Fl_Tabs](#) widget, but instead of displaying tabs it only changes "tabs" under program control.

Its primary purpose is to support "wizards" that step a user through configuration or troubleshooting tasks.

As with [Fl_Tabs](#), wizard panes are composed of child (usually [Fl_Group](#)) widgets. Navigation buttons must be added separately.

30.106.2 Constructor & Destructor Documentation

30.106.2.1 Fl_Wizard::Fl_Wizard (int xx, int yy, int ww, int hh, const char * l = 0)

The constructor creates the [Fl_Wizard](#) widget at the specified position and size.

The inherited destructor destroys the widget and its children.

30.106.3 Member Function Documentation

30.106.3.1 void Fl_Wizard::next ()

This method shows the next child of the wizard.

If the last child is already visible, this function does nothing.

30.106.3.2 void Fl_Wizard::prev ()

Shows the previous child.

30.106.3.3 void Fl_Wizard::value (Fl_Widget * *kid*)

Sets the child widget that is visible.

30.106.3.4 Fl_Widget * Fl_Wizard::value ()

Gets the current visible child widget.

The documentation for this class was generated from the following files:

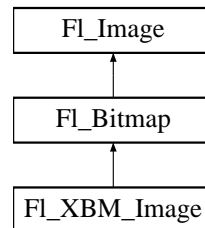
- Fl_Wizard.H
- Fl_Wizard.cxx

30.107 Fl_XBM_Image Class Reference

The [Fl_XBM_Image](#) class supports loading, caching, and drawing of X Bitmap (XBM) bitmap files.

```
#include <Fl_XBM_Image.H>
```

Inheritance diagram for Fl_XBM_Image::



Public Member Functions

- [Fl_XBM_Image](#) (const char *filename)

The constructor loads the named XBM file from the given name filename.

30.107.1 Detailed Description

The [Fl_XBM_Image](#) class supports loading, caching, and drawing of X Bitmap (XBM) bitmap files.

30.107.2 Constructor & Destructor Documentation

30.107.2.1 Fl_XBM_Image::Fl_XBM_Image (const char * *name*)

The constructor loads the named XBM file from the given name filename.

The destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

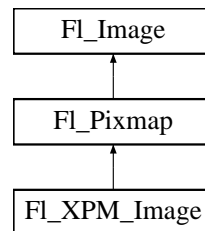
- Fl_XBM_Image.H
- Fl_XBM_Image.cxx

30.108 Fl_XPM_Image Class Reference

The [Fl_XPM_Image](#) class supports loading, caching, and drawing of X Pixmap (XPM) images, including transparency.

```
#include <Fl_XPM_Image.H>
```

Inheritance diagram for Fl_XPM_Image::



Public Member Functions

- [Fl_XPM_Image](#) (const char *filename)

The constructor loads the XPM image from the name filename.

30.108.1 Detailed Description

The [Fl_XPM_Image](#) class supports loading, caching, and drawing of X Pixmap (XPM) images, including transparency.

30.108.2 Constructor & Destructor Documentation

30.108.2.1 Fl_XPM_Image::Fl_XPM_Image (const char * name)

The constructor loads the XPM image from the name filename.

The destructor free all memory and server resources that are used by the image.

The documentation for this class was generated from the following files:

- Fl_XPM_Image.H
- Fl_XPM_Image.cxx

Chapter 31

File Documentation

31.1 Enumerations.H File Reference

This file contains type definitions and general enumerations.

```
#include "Fl_Export.H"  
#include "fl_types.h"
```

Defines

Event States

The following constants define bits in the *Fl::event_state()* value.

- #define **FL_ALT** 0x00080000
One of the alt keys is down.
- #define **FL_BUTTON**(n) (0x00800000<<(n))
Mouse button n (n > 0) is pushed.
- #define **FL_BUTTON1** 0x01000000
Mouse button 1 is pushed.
- #define **FL_BUTTON2** 0x02000000
Mouse button 2 is pushed.
- #define **FL_BUTTON3** 0x04000000
Mouse button 3 is pushed.
- #define **FL_BUTTONS** 0x7f000000
Any mouse button is pushed.
- #define **FL_CAPS_LOCK** 0x00020000
The caps lock is on.
- #define **FL_COMMAND** FL_CTRL
An alias for FL_CTRL on WIN32 and X11, or FL_META on MacOS X.

- #define [FL_CTRL](#) 0x00040000
One of the ctrl keys is down.
- #define [FL_META](#) 0x00400000
One of the meta/Windows keys is down.
- #define [FL_NUM_LOCK](#) 0x00100000
The num lock is on.
- #define [FL_SCROLL_LOCK](#) 0x00800000
The scroll lock is on.
- #define [FL_SHIFT](#) 0x00010000
One of the shift keys is down.

Mouse and Keyboard Events

This and the following constants define the non-ASCII keys on the keyboard for [FL_KEYBOARD](#) and [FL_SHORTCUT](#) events.

Todo

[FL_Button](#) and [FL_key...](#) constants could be structured better (use an enum or some doxygen grouping ?)

See also:

[Fl::event_key\(\)](#) and [Fl::get_key\(int\)](#) (use ascii letters for all other keys):

- #define [FL_Alt_L](#) 0xffe9
The left alt key.
- #define [FL_Alt_R](#) 0xffea
The right alt key.
- #define [FL_BackSpace](#) 0xff08
The backspace key.
- #define [FL_Button](#) 0xfee8
A mouse button; use [FL_Button](#) + n for mouse button n.
- #define [FL_Caps_Lock](#) 0xffe5
The caps lock key.
- #define [FL_Control_L](#) 0xffe3
The lefthand control key.
- #define [FL_Control_R](#) 0xffe4
The righthand control key.
- #define [FL_Delete](#) 0xffff
The delete key.
- #define [FL_Down](#) 0xff54
The down arrow key.

- #define [FL_End](#) 0xff57
The end key.
- #define [FL_Enter](#) 0xff0d
The enter key.
- #define [FL_Escape](#) 0xff1b
The escape key.
- #define [FL_F](#) 0xffbd
One of the function keys; use $FL_F + n$ for function key n .
- #define [FL_F_Last](#) 0xffe0
The last function key; use to range-check function keys.
- #define [FL_Help](#) 0xff68
The 'help' key on Mac keyboards.
- #define [FL_Home](#) 0xff50
The home key.
- #define [FL_Insert](#) 0xff63
The insert key.
- #define [FL_KP](#) 0xff80
One of the keypad numbers; use $FL_KP + n$ for number n .
- #define [FL_KP_Enter](#) 0xff8d
The enter key on the keypad, same as $FL_KP + '\r'$.
- #define [FL_KP_Last](#) 0xffbd
The last keypad key; use to range-check keypad.
- #define [FL_Left](#) 0xff51
The left arrow key.
- #define [FL_Menu](#) 0xff67
The menu key.
- #define [FL_Meta_L](#) 0xffe7
The left meta/Windows key.
- #define [FL_Meta_R](#) 0xffe8
The right meta/Windows key.
- #define [FL_Num_Lock](#) 0xff7f
The num lock key.
- #define [FL_Page_Down](#) 0xff56
The page-down key.
- #define [FL_Page_Up](#) 0xff55

The page-up key.

- #define `FL_Pause` 0xff13
The pause key.
- #define `FL_Print` 0xff61
The print (or print-screen) key.
- #define `FL_Right` 0xff53
The right arrow key.
- #define `FL_Scroll_Lock` 0xff14
The scroll lock key.
- #define `FL_Shift_L` 0xffe1
The lefthand shift key.
- #define `FL_Shift_R` 0xffe2
The righthand shift key.
- #define `FL_Tab` 0xff09
The tab key.
- #define `FL_Up` 0xff52
The up arrow key.

Mouse Buttons

These constants define the button numbers for `FL_PUSH` and `FL_RELEASE` events.

See also:

`Fl::event_button()`

- #define `FL_LEFT_MOUSE` 1
The left mouse button.
- #define `FL_MIDDLE_MOUSE` 2
The middle mouse button.
- #define `FL_RIGHT_MOUSE` 3
The right mouse button.

Version Numbers

FLTK defines some constants to help the programmer to find out, for which FLTK version a program is compiled.

The following constants are defined:

- #define `FL_MAJOR_VERSION` 1
The major release version of this FLTK library.
- #define `FL_MINOR_VERSION` 3
The minor release version for this library.

- `#define FL_PATCH_VERSION 0`
The patch version for this library.
- `#define FL_VERSION`
The FLTK version number as a double.

Typedefs

- `typedef int Fl_Fontsize`
Size of a font in pixels.

Enumerations

- `enum Fl_Event {`
`FL_NO_EVENT = 0, FL_PUSH = 1, FL_RELEASE = 2, FL_ENTER = 3,`
`FL_LEAVE = 4, FL_DRAG = 5, FL_FOCUS = 6, FL_UNFOCUS = 7,`
`FL_KEYDOWN = 8, FL_KEYBOARD = 8, FL_KEYUP = 9, FL_CLOSE = 10,`
`FL_MOVE = 11, FL_SHORTCUT = 12, FL_DEACTIVATE = 13, FL_ACTIVATE = 14,`
`FL_HIDE = 15, FL_SHOW = 16, FL_PASTE = 17, FL_SELECTIONCLEAR = 18,`
`FL_MOUSEWHEEL = 19, FL_DND_ENTER = 20, FL_DND_DRAG = 21, FL_DND_LEAVE =`
`22,`
`FL_DND_RELEASE = 23 }`
Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application.
- `enum Fl_Labeltype {`
`FL_NORMAL_LABEL = 0, FL_NO_LABEL, _FL_SHADOW_LABEL, _FL_ENGRAVED_`
`_LABEL,`
`_FL_EMBOSSSED_LABEL, _FL_MULTI_LABEL, _FL_ICON_LABEL, _FL_IMAGE_LABEL,`
`FL_FREE_LABELTYPE }`
The `labeltype()` method sets the type of the label.

When Conditions

- `enum Fl_When {`
`FL_WHEN_NEVER = 0, FL_WHEN_CHANGED = 1, FL_WHEN_NOT_CHANGED = 2, FL_`
`WHEN_RELEASE = 4,`
`FL_WHEN_RELEASE_ALWAYS = 6, FL_WHEN_ENTER_KEY = 8, FL_WHEN_ENTER_`
`KEY_ALWAYS = 10, FL_WHEN_ENTER_KEY_CHANGED = 11 }`
These constants determine when a callback is performed.

Variables

- FL_EXPORT [FL_Fontsize FL_NORMAL_SIZE](#)
normal font size

31.1.1 Detailed Description

This file contains type definitions and general enumerations.

31.1.2 Define Documentation

31.1.2.1 `#define FL_MAJOR_VERSION 1`

The major release version of this FLTK library.

See also:

[FL_VERSION](#)

31.1.2.2 `#define FL_MINOR_VERSION 3`

The minor release version for this library.

FLTK remains mostly source-code compatible between minor version changes.

31.1.2.3 `#define FL_PATCH_VERSION 0`

The patch version for this library.

FLTK remains binary compatible between patches.

31.1.2.4 `#define FL_VERSION`

Value:

```
((double)FL_MAJOR_VERSION + \  
                                (double)FL_MINOR_VERSION * 0.01 + \  
                                (double)FL_PATCH_VERSION * 0.0001)
```

The FLTK version number as a *double*.

This is changed slightly from the beta versions because the old "const double" definition would not allow for conditional compilation...

FL_VERSION is a double that describes the major and minor version numbers. Version 1.1 is actually stored as 1.01 to allow for more than 9 minor releases.

The FL_MAJOR_VERSION, FL_MINOR_VERSION, and FL_PATCH_VERSION constants give the integral values for the major, minor, and patch releases respectively.

31.1.3 Typedef Documentation

31.1.3.1 typedef unsigned Fl_Align

Flags to control the label alignment.

This controls how the label is displayed next to or inside the widget. The default value is `FL_ALIGN_CENTER` for most widgets, which centers the label inside the widget.

Flags can be or'd to achieve a combination of alignments.

See also:

[FL_ALIGN_CENTER](#), etc.

31.1.3.2 typedef int Fl_Font

A font number is an index into the internal font table.

The following constants define the standard FLTK fonts:

31.1.3.3 typedef int Fl_Fontsize

Size of a font in pixels.

This is the approximate height of a font in pixels.

31.1.4 Enumeration Type Documentation

31.1.4.1 enum Fl_Boxtype

Enumerator:

FL_NO_BOX nothing is drawn at all, this box is invisible

FL_FLAT_BOX a flat box

FL_UP_BOX see figure 1

FL_DOWN_BOX see figure 1

FL_UP_FRAME see figure 1

FL_DOWN_FRAME see figure 1

FL_THIN_UP_BOX see figure 1

FL_THIN_DOWN_BOX see figure 1

FL_THIN_UP_FRAME see figure 1

FL_THIN_DOWN_FRAME see figure 1

FL_ENGRAVED_BOX see figure 1

FL_EMBOSSSED_BOX see figure 1

FL_ENGRAVED_FRAME see figure 1

FL_EMBOSSSED_FRAME see figure 1

FL_BORDER_BOX see figure 1

_FL_SHADOW_BOX see figure 1

FL_BORDER_FRAME see figure 1
_FL_SHADOW_FRAME see figure 1
_FL_ROUNDED_BOX see figure 1
_FL_RSHADOW_BOX see figure 1
_FL_ROUNDED_FRAME see figure 1
_FL_RFLAT_BOX see figure 1
_FL_ROUND_UP_BOX see figure 1
_FL_ROUND_DOWN_BOX see figure 1
_FL_DIAMOND_UP_BOX see figure 1
_FL_DIAMOND_DOWN_BOX see figure 1
_FL_OVAL_BOX see figure 1
_FL_OSHADOW_BOX see figure 1
_FL_OVAL_FRAME see figure 1
_FL_OFLAT_BOX see figure 1
_FL_PLASTIC_UP_BOX plastic version of *FL_UP_BOX*
_FL_PLASTIC_DOWN_BOX plastic version of *FL_DOWN_BOX*
_FL_PLASTIC_UP_FRAME plastic version of *FL_UP_FRAME*
_FL_PLASTIC_DOWN_FRAME plastic version of *FL_DOWN_FRAME*
_FL_PLASTIC_THIN_UP_BOX plastic version of *FL_THIN_UP_BOX*
_FL_PLASTIC_THIN_DOWN_BOX plastic version of *FL_THIN_DOWN_BOX*
_FL_PLASTIC_ROUND_UP_BOX plastic version of *FL_ROUND_UP_BOX*
_FL_PLASTIC_ROUND_DOWN_BOX plastic version of *FL_ROUND_DOWN_BOX*
_FL_GTK_UP_BOX gtk+ version of *FL_UP_BOX*
_FL_GTK_DOWN_BOX gtk+ version of *FL_DOWN_BOX*
_FL_GTK_UP_FRAME gtk+ version of *FL_UP_FRAME*
_FL_GTK_DOWN_FRAME gtk+ version of *FL_DOWN_FRAME*
_FL_GTK_THIN_UP_BOX gtk+ version of *FL_THIN_UP_BOX*
_FL_GTK_THIN_DOWN_BOX gtk+ version of *FL_THIN_DOWN_BOX*
_FL_GTK_THIN_UP_FRAME gtk+ version of *FL_UP_FRAME*
_FL_GTK_THIN_DOWN_FRAME gtk+ version of *FL_THIN_DOWN_FRAME*
_FL_GTK_ROUND_UP_BOX gtk+ version of *FL_ROUND_UP_BOX*
_FL_GTK_ROUND_DOWN_BOX gtk+ version of *FL_ROUND_DOWN_BOX*
FL_FREE_BOXTYPE the first free box type for creation of new box types

31.1.4.2 enum *Fl_Color*

The *Fl_Color* enumeration type holds a FLTK color value.

Colors are either 8-bit indexes into a virtual colormap or 24-bit RGB color values.

Color indices occupy the lower 8 bits of the value, while RGB colors occupy the upper 24 bits, for a byte organization of RGBI.

Todo

enum `Fl_Color` needs some more comments for values, see `Fl/Enumerations.H`

Enumerator:

FL_FOREGROUND_COLOR the default foreground color (0) used for labels and text
FL_BACKGROUND2_COLOR the default background color for text, list, and valuator widgets
FL_INACTIVE_COLOR the inactive foreground color
FL_SELECTION_COLOR the default selection/highlight color

31.1.4.3 enum Fl_Event

Every time a user moves the mouse pointer, clicks a button, or presses a key, an event is generated and sent to your application.

Events can also come from other programs like the window manager.

Events are identified by the integer argument passed to the `Fl_Widget::handle()` virtual method. Other information about the most recent event is stored in static locations and acquired by calling the `Fl::event_*()` methods. This static information remains valid until the next event is read from the window system, so it is ok to look at it outside of the `handle()` method.

See also:

[Fl::event_text\(\)](#), [Fl::event_key\(\)](#), class [Fl::](#)

Enumerator:

FL_NO_EVENT No event.

FL_PUSH A mouse button has gone down with the mouse pointing at this widget.
 You can find out what button by calling [Fl::event_button\(\)](#). You find out the mouse position by calling [Fl::event_x\(\)](#) and [Fl::event_y\(\)](#).
 A widget indicates that it "wants" the mouse click by returning non-zero from its [Fl_Widget::handle\(\)](#) method. It will then become the [Fl::pushed\(\)](#) widget and will get `FL_DRAG` and the matching `FL_RELEASE` events. If [Fl_Widget::handle\(\)](#) returns zero then FLTK will try sending the `FL_PUSH` to another widget.

FL_RELEASE A mouse button has been released.
 You can find out what button by calling [Fl::event_button\(\)](#).
 In order to receive the `FL_RELEASE` event, the widget must return non-zero when handling `FL_PUSH`.

FL_ENTER The mouse has been moved to point at this widget.
 This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its `handle()` method. It then becomes the [Fl::belowmouse\(\)](#) widget and will receive `FL_MOVE` and `FL_LEAVE` events.

FL_LEAVE The mouse has moved out of the widget.
 In order to receive the `FL_LEAVE` event, the widget must return non-zero when handling `FL_ENTER`.

FL_DRAG The mouse has moved with a button held down.
 The current button state is in [Fl::event_state\(\)](#). The mouse position is in [Fl::event_x\(\)](#) and [Fl::event_y\(\)](#).
 In order to receive `FL_DRAG` events, the widget must return non-zero when handling `FL_PUSH`.

FL_FOCUS This indicates an *attempt* to give a widget the keyboard focus.

If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its `handle()` method. It then becomes the `Fl::focus()` widget and gets `FL_KEYDOWN`, `FL_KEYUP`, and `FL_UNFOCUS` events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check `Fl::event_key()` to figure out why it moved. For navigation it will be the key pressed and interaction with the window manager it will be zero.

FL_UNFOCUS This event is sent to the previous `Fl::focus()` widget when another widget gets the focus or the window loses focus.

FL_KEYDOWN A key was pressed or released.

The key can be found in `Fl::event_key()`. The text that the key should insert can be found with `Fl::event_text()` and its length is in `Fl::event_length()`. If you use the key `handle()` should return 1. If you return zero then FLTK assumes you ignored the key and will then attempt to send it to a parent widget. If none of them want it, it will change the event into a `FL_SHORTCUT` event. To receive `FL_KEYBOARD` events you must also respond to the `FL_FOCUS` and `FL_UNFOCUS` events.

If you are writing a text-editing widget you may also want to call the `Fl::compose()` function to translate individual keystrokes into foreign characters.

`FL_KEYUP` events are sent to the widget that currently has focus. This is not necessarily the same widget that received the corresponding `FL_KEYDOWN` event because focus may have changed between events.

FL_KEYBOARD Equivalent to `FL_KEYDOWN`.

See also:

`FL_KEYDOWN`

FL_KEYUP Key release event.

See also:

`FL_KEYDOWN`

FL_CLOSE The user clicked the close button of a window.

This event is used internally only to trigger the callback of `Fl_Window` derived classed. The default callback closes the window calling `Fl_Window::hide()`.

FL_MOVE The mouse has moved without any mouse buttons held down.

This event is sent to the `Fl::belowmouse()` widget.

In order to receive `FL_MOVE` events, the widget must return non-zero when handling `FL_ENTER`.

FL_SHORTCUT If the `Fl::focus()` widget is zero or ignores an `FL_KEYBOARD` event then FLTK tries sending this event to every widget it can, until one of them returns non-zero.

`FL_SHORTCUT` is first sent to the `Fl::belowmouse()` widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero. FLTK tries really hard to not to ignore any keystrokes!

You can also make "global" shortcuts by using `Fl::add_handler()`. A global shortcut will work no matter what windows are displayed or which one has the focus.

FL_DEACTIVATE This widget is no longer active, due to `Fl_Widget::deactivate()` being called on it or one of its parents.

`Fl_Widget::active()` may still be true after this, the widget is only active if `Fl_Widget::active()` is true on it and all its parents (use `Fl_Widget::active_r()` to check this).

FL_ACTIVATE This widget is now active, due to `Fl_Widget::activate()` being called on it or one of its parents.

FL_HIDE This widget is no longer visible, due to `Fl_Widget::hide()` being called on it or one of its parents, or due to a parent window being minimized.

`Fl_Widget::visible()` may still be true after this, but the widget is visible only if `visible()` is true for it and all its parents (use `Fl_Widget::visible_r()` to check this).

FL_SHOW This widget is visible again, due to `Fl_Widget::show()` being called on it or one of its parents, or due to a parent window being restored.

Child `Fl_Windows` respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass `FL_SHOW` to the base class `Fl_Widget::handle()` method!

FL_PASTE You should get this event some time after you call `Fl::paste()`.

The contents of `Fl::event_text()` is the text to insert and the number of characters is in `Fl::event_length()`.

FL_SELECTIONCLEAR The `Fl::selection_owner()` will get this event before the selection is moved to another widget.

This indicates that some other widget or program has claimed the selection. Motif programs used this to clear the selection indication. Most modern programs ignore this.

FL_MOUSEWHEEL The user has moved the mouse wheel.

The `Fl::event_dx()` and `Fl::event_dy()` methods can be used to find the amount to scroll horizontally and vertically.

FL_DND_ENTER The mouse has been moved to point at this widget.

A widget that is interested in receiving drag'n'drop data must return 1 to receive `FL_DND_DRAG`, `FL_DND_LEAVE` and `FL_DND_RELEASE` events.

FL_DND_DRAG The mouse has been moved inside a widget while dragging data.

A widget that is interested in receiving drag'n'drop data should indicate the possible drop position.

FL_DND_LEAVE The mouse has moved out of the widget.

FL_DND_RELEASE The user has released the mouse button dropping data into the widget.

If the widget returns 1, it will receive the data in the immediately following `FL_PASTE` event.

31.1.4.4 enum Fl_Labeltype

The `labeltype()` method sets the type of the label.

The following standard label types are included:

Todo

The doxygen comments are incomplete, and some labeltypes are starting with an underscore. Also, there are three external functions undocumented (yet):

- `fl_define_FL_SHADOW_LABEL()`
- `fl_define_FL_ENGRAVED_LABEL()`
- `fl_define_FL_EMBOSED_LABEL()`

Enumerator:

FL_NORMAL_LABEL draws the text (0)

FL_NO_LABEL does nothing

_FL_SHADOW_LABEL draws a drop shadow under the text

_FL_ENGRAVED_LABEL draws edges as though the text is engraved

_FL_EMBOSSSED_LABEL draws edges as though the text is raised

_FL_MULTI_LABEL ?

_FL_ICON_LABEL draws the icon associated with the text

_FL_IMAGE_LABEL ?

FL_FREE_LABELTYPE first free labeltype to use for creating own labeltypes

31.1.4.5 enum Fl_When

These constants determine when a callback is performed.

See also:

[Fl_Widget::when\(\);](#)

Todo

doxygen comments for values are incomplete and maybe wrong or unclear

Enumerator:

FL_WHEN_NEVER Never call the callback.

FL_WHEN_CHANGED Do the callback only when the widget value changes.

FL_WHEN_NOT_CHANGED Do the callback whenever the user interacts with the widget.

FL_WHEN_RELEASE Do the callback when the button or key is released and the value changes.

FL_WHEN_RELEASE_ALWAYS Do the callback when the button or key is released, even if the value doesn't change.

FL_WHEN_ENTER_KEY Do the callback when the user presses the ENTER key and the value changes.

FL_WHEN_ENTER_KEY_ALWAYS Do the callback when the user presses the ENTER key, even if the value doesn't change.

FL_WHEN_ENTER_KEY_CHANGED ?

31.1.5 Function Documentation

31.1.5.1 Fl_Boxtype fl_box (Fl_Boxtype *b*) [inline]

Get the filled version of a frame.

If no filled version of a given frame exists, the behavior of this function is undefined and some random box or frame is returned,

31.1.5.2 Fl_Boxtype fl_down (Fl_Boxtype *b*) [inline]

Get the "pressed" or "down" version of a box.

If no "down" version of a given box exists, the behavior of this function is undefined and some random box or frame is returned,

31.1.5.3 **Fl_Boxtype fl_frame (Fl_Boxtype *b*)** `[inline]`

Get the unfilled, frame only version of a box.

If no frame version of a given box exists, the behavior of this function is undefined and some random box or frame is returned,

31.1.6 Variable Documentation

31.1.6.1 **const Fl_Align FL_ALIGN_BOTTOM = (Fl_Align)2**

Align the label at the bottom of the widget.

31.1.6.2 **const Fl_Align FL_ALIGN_CENTER = (Fl_Align)0**

Align the label horizontally in the middle.

31.1.6.3 **const Fl_Align FL_ALIGN_CLIP = (Fl_Align)64**

All parts of the label that are larger than the widget will not be drawn .

31.1.6.4 **const Fl_Align FL_ALIGN_IMAGE_OVER_TEXT = (Fl_Align)0**

If the label contains an image, draw the text below the image.

31.1.6.5 **const Fl_Align FL_ALIGN_INSIDE = (Fl_Align)16**

Draw the label inside of the widget.

31.1.6.6 **const Fl_Align FL_ALIGN_LEFT = (Fl_Align)4**

Align the label at the left of the widget.

Inside labels appear left-justified starting at the left side of the widget, outside labels are right-justified and drawn to the left of the widget.

31.1.6.7 **const Fl_Align FL_ALIGN_RIGHT = (Fl_Align)8**

Align the label to the right of the widget.

31.1.6.8 **const Fl_Align FL_ALIGN_TEXT_OVER_IMAGE = (Fl_Align)32**

If the label contains an image, draw the text on top of the image.

31.1.6.9 **const Fl_Align FL_ALIGN_TOP = (Fl_Align)1**

Align the label at the top of the widget.

Inside labels appear below the top, outside labels are drawn on top of the widget.

31.1.6.10 const Fl_Align FL_ALIGN_WRAP = (Fl_Align)128

Wrap text that does not fit the width of the widget.

31.2 fl_arc.cxx File Reference

Utility functions for drawing arcs and circles.

```
#include <FL/fl_draw.H>
```

```
#include <FL/math.h>
```

Functions

- void [fl_arc](#) (double x, double y, double r, double start, double end)

Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using [scale](#) and rotate before calling [fl_arc\(\)](#).

31.2.1 Detailed Description

Utility functions for drawing arcs and circles.

31.3 fl_arci.cxx File Reference

Utility functions for drawing circles using integers.

```
#include <FL/fl_draw.H>
#include <FL/x.H>
#include <config.h>
```

Functions

- void [fl_arc](#) (int x, int y, int w, int h, double a1, double a2)
Draw ellipse sections using integer coordinates.
- void [fl_pie](#) (int x, int y, int w, int h, double a1, double a2)
Draw filled ellipse sections using integer coordinates.

31.3.1 Detailed Description

Utility functions for drawing circles using integers.

31.4 fl_boxtype.cxx File Reference

drawing code for common box types.

```
#include <FL/Fl.H>
#include <FL/Fl_Widget.H>
#include <FL/fl_draw.H>
#include <config.h>
```

Defines

- #define **D1** BORDER_WIDTH
- #define **D2** (BORDER_WIDTH+BORDER_WIDTH)
- #define **fl_border_box** fl_rectbound
allow consistent naming

Functions

- void **fl_border_frame** (int x, int y, int w, int h, **Fl_Color** c)
Draws a frame of type FL_BORDER_FRAME.
- void **fl_down_box** (int x, int y, int w, int h, **Fl_Color** c)
Draws a box of type FL_DOWN_BOX.
- void **fl_down_frame** (int x, int y, int w, int h, **Fl_Color**)
Draws a frame of type FL_DOWN_FRAME.
- void **fl_draw_box** (**Fl_Boxtype** t, int x, int y, int w, int h, **Fl_Color** c)
Draws a box using given type, position, size and color.
- void **fl_embossed_box** (int x, int y, int w, int h, **Fl_Color** c)
Draws a box of type FL_EMBOSSED_BOX.
- void **fl_embossed_frame** (int x, int y, int w, int h, **Fl_Color**)
Draws a frame of type FL_EMBOSSED_FRAME.
- void **fl_engraved_box** (int x, int y, int w, int h, **Fl_Color** c)
Draws a box of type FL_ENGRAVED_BOX.
- void **fl_engraved_frame** (int x, int y, int w, int h, **Fl_Color**)
Draws a frame of type FL_ENGRAVED_FRAME.
- void **fl_frame** (const char *s, int x, int y, int w, int h)
Draws a series of line segments around the given box.
- void **fl_frame2** (const char *s, int x, int y, int w, int h)
Draws a series of line segments around the given box.

- `uchar * fl_gray_ramp ()`
- `void fl_internal_boxtype (Fl_Boxtype t, Fl_Box_Draw_F *f)`
Sets the drawing function for a given box type.
- `void fl_no_box (int, int, int, int, Fl_Color)`
Draws a box of type FL_NO_BOX.
- `void fl_rectbound (int x, int y, int w, int h, Fl_Color bgcolor)`
Draws a bounded rectangle with a given position, size and color.
- `void fl_thin_down_box (int x, int y, int w, int h, Fl_Color c)`
Draws a box of type FL_THIN_DOWN_BOX.
- `void fl_thin_down_frame (int x, int y, int w, int h, Fl_Color)`
Draws a frame of type FL_THIN_DOWN_FRAME.
- `void fl_thin_up_box (int x, int y, int w, int h, Fl_Color c)`
Draws a box of type FL_THIN_UP_BOX.
- `void fl_thin_up_frame (int x, int y, int w, int h, Fl_Color)`
Draws a frame of type FL_THIN_UP_FRAME.
- `void fl_up_box (int x, int y, int w, int h, Fl_Color c)`
Draws a box of type FL_UP_BOX.
- `void fl_up_frame (int x, int y, int w, int h, Fl_Color)`
Draws a frame of type FL_UP_FRAME.

31.4.1 Detailed Description

drawing code for common box types.

31.4.2 Function Documentation

31.4.2.1 `void fl_internal_boxtype (Fl_Boxtype t, Fl_Box_Draw_F *f)`

Sets the drawing function for a given box type.

Parameters:

- ← *t* box type
- ← *f* box drawing function

31.4.2.2 `void fl_rectbound (int x, int y, int w, int h, Fl_Color bgcolor)`

Draws a bounded rectangle with a given position, size and color.

Equivalent to drawing a box of type FL_BORDER_BOX.

31.5 fl_color.cxx File Reference

Color handling.

```
#include "Fl_XColor.H"
#include <FL/Fl.H>
#include <FL/x.H>
#include <FL/fl_draw.H>
#include "fl_cmap.h"
```

Defines

- `#define fl_overlay 0`
HAVE_OVERLAY determines whether fl_overlay is variable or defined as 0.

Functions

- `FL_EXPORT void fl_color (Fl_Color i)`
Sets the color for all subsequent drawing operations.
- `FL_EXPORT void fl_color (uchar r, uchar g, uchar b)`
Set the color for all subsequent drawing operations.
- `Fl_Color fl_color_average (Fl_Color color1, Fl_Color color2, float weight)`
Returns the weighted average color between the two given colors.
- `Fl_Color fl_contrast (Fl_Color fg, Fl_Color bg)`
Returns a color that contrasts with the background color.
- `Fl_Color fl_inactive (Fl_Color c)`
Returns the inactive, dimmed version of the given color.
- `ulong fl_xpixel (Fl_Color i)`
Returns the X pixel number used to draw the given FLTK color index.
- `ulong fl_xpixel (uchar r, uchar g, uchar b)`
Returns the X pixel number used to draw the given rgb color.

Variables

- `uchar fl_bluemask`
color mask used in current color map handling
- `int fl_blueshift`
color shift used in current color map handling

- [Fl_Color fl_color_](#)
Current color for drawing operations.
- [int fl_extrashift](#)
color shift used in current color map handling
- [uchar fl_greenmask](#)
color mask used in current color map handling
- [int fl_greenshift](#)
color shift used in current color map handling
- [uchar fl_redmask](#)
color mask used in current color map handling
- [int fl_redshift](#)
color shift used in current color map handling
- [Fl_XColor fl_xmap](#) [1][256]
HAVE_OVERLAY determines whether fl_xmap is one or two planes.

31.5.1 Detailed Description

Color handling.

31.6 FL_Color_Chooser.H File Reference

[FL_Color_Chooser](#) widget .

```
#include <FL/Fl_Group.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Return_Button.H>
#include <FL/Fl_Choice.H>
#include <FL/Fl_Value_Input.H>
```

Classes

- class [FL_Color_Chooser](#)

The [FL_Color_Chooser](#) widget provides a standard RGB color chooser.

31.6.1 Detailed Description

[FL_Color_Chooser](#) widget .

31.7 fl_curve.cxx File Reference

Utility for drawing Bezier curves, adding the points to the current fl_begin/fl_vertex/fl_end path.

```
#include <FL/fl_draw.H>
#include <math.h>
```

Functions

- void [fl_curve](#) (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)

Add a series of points on a Bezier curve to the path.

31.7.1 Detailed Description

Utility for drawing Bezier curves, adding the points to the current fl_begin/fl_vertex/fl_end path.

Incremental math implementation: I very much doubt this is optimal! From Foley/vanDam page 511. If anybody has a better algorithm, please send it!

31.8 fl_draw.H File Reference

utility header to pull drawing functions together

```
#include "Enumerations.H"
#include "Fl_Window.H"
```

Defines

- #define `fl_clip` `fl_push_clip`
The `fl_clip()` name is deprecated and will be removed from future releases.

Typedefs

- typedef void(* `Fl_Draw_Image_Cb`)(void *data, int x, int y, int w, `uchar` *buf)
signature of image generation callback function.

Enumerations

- enum {
 `FL_SOLID` = 0, `FL_DASH` = 1, `FL_DOT` = 2, `FL_DASHDOT` = 3,
 `FL_DASHDOTDOT` = 4, `FL_CAP_FLAT` = 0x100, `FL_CAP_ROUND` = 0x200, `FL_CAP_SQUARE` = 0x300,
 `FL_JOIN_MITER` = 0x1000, `FL_JOIN_ROUND` = 0x2000, `FL_JOIN_BEVEL` = 0x3000 }

Functions

- `FL_EXPORT int fl_add_symbol` (const char *name, void(*drawit)(`Fl_Color`), int scalable)
Adds a symbol to the system.
- `FL_EXPORT void fl_arc` (double x, double y, double r, double start, double a)
Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using `scale` and `rotate` before calling `fl_arc()`.
- `FL_EXPORT void fl_arc` (int x, int y, int w, int h, double a1, double a2)
Draw ellipse sections using integer coordinates.
- `FL_EXPORT void fl_begin_complex_polygon` ()
Starts drawing a complex filled polygon.
- `FL_EXPORT void fl_begin_line` ()
Starts drawing a list of lines.
- `FL_EXPORT void fl_begin_loop` ()
Starts drawing a closed sequence of lines.

- FL_EXPORT void [fl_begin_points](#) ()
Starts drawing a list of points.
- FL_EXPORT void [fl_begin_polygon](#) ()
Starts drawing a convex filled polygon.
- FL_EXPORT char [fl_can_do_alpha_blending](#) ()
Checks whether platform supports true alpha blending for RGBA images.
- FL_EXPORT void [fl_chord](#) (int x, int y, int w, int h, double a1, double a2)
fl_chord declaration is a place holder - the function does not yet exist
- FL_EXPORT void [fl_circle](#) (double x, double y, double r)
fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- FL_EXPORT int [fl_clip_box](#) (int, int, int, int, int &x, int &y, int &w, int &h)
Intersects the rectangle with the current clip region and returns the bounding box of the result.
- [Fl_Color](#) [fl_color](#) ()
Returns the last fl_color() that was set.
- FL_EXPORT void [fl_color](#) (uchar r, uchar g, uchar b)
Set the color for all subsequent drawing operations.
- void [fl_color](#) (int c)
for back compatibility - use fl_color(Fl_Color c) instead
- FL_EXPORT void [fl_color](#) (Fl_Color i)
Sets the color for all subsequent drawing operations.
- FL_EXPORT void [fl_cursor](#) (Fl_Cursor, Fl_Color fg=FL_BLACK, Fl_Color bg=FL_WHITE)
Sets the cursor for the current window to the specified shape and colors.
- FL_EXPORT void [fl_curve](#) (double X0, double Y0, double X1, double Y1, double X2, double Y2, double X3, double Y3)
Add a series of points on a Bezier curve to the path.
- FL_EXPORT int [fl_descent](#) ()
Returns the recommended distance above the bottom of a fl_height() tall box to draw the text at so it looks centered vertically in that box.
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y, int w, int h, Fl_Align align, void(*callthis)(const char *, int, int, int), Fl_Image *img=0, int draw_symbols=1)
The same as fl_draw(const char,int,int,int,int,Fl_Align,Fl_Image*,int) with the addition of the callthis parameter, which is a pointer to a text drawing function such as fl_draw(const char*, int, int, int) to do the real work.*
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y, int w, int h, Fl_Align align, Fl_Image *img=0, int draw_symbols=1)
Fancy string drawing function which is used to draw all the labels.

- FL_EXPORT void [fl_draw](#) (const char *str, int n, int x, int y)
Draws an array of n characters starting at the given location.
- FL_EXPORT void [fl_draw](#) (const char *str, int x, int y)
Draws a nul-terminated string starting at the given location.
- FL_EXPORT void [fl_draw_box](#) (FL_Boxtype, int x, int y, int w, int h, FL_Color)
Draws a box using given type, position, size and color.
- FL_EXPORT void [fl_draw_image](#) (FL_Draw_Image_Cb cb, void *data, int X, int Y, int W, int H, int D=3)
Draw image using callback function to generate image data.
- FL_EXPORT void [fl_draw_image](#) (const uchar *buf, int X, int Y, int W, int H, int D=3, int L=0)
Draw an 8-bit per color RGB or luminance image.
- FL_EXPORT void [fl_draw_image_mono](#) (FL_Draw_Image_Cb cb, void *data, int X, int Y, int W, int H, int D=1)
Draw gray-scale image using callback function to generate image data.
- FL_EXPORT void [fl_draw_image_mono](#) (const uchar *buf, int X, int Y, int W, int H, int D=1, int L=0)
Draw a gray-scale (1 channel) image.
- FL_EXPORT int [fl_draw_pixmap](#) (const char *const *cdata, int x, int y, FL_Color=FL_GRAY)
Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int [fl_draw_pixmap](#) (char *const *data, int x, int y, FL_Color=FL_GRAY)
Draw XPM image data, with the top-left corner at the given position.
- FL_EXPORT int [fl_draw_symbol](#) (const char *label, int x, int y, int w, int h, FL_Color)
Draw the named symbol in the given rectangle using the given color.
- FL_EXPORT void [fl_end_complex_polygon](#) ()
Ends complex filled polygon, and draws.
- FL_EXPORT void [fl_end_line](#) ()
Ends list of lines, and draws.
- FL_EXPORT void [fl_end_loop](#) ()
Ends closed sequence of lines, and draws.
- FL_EXPORT void [fl_end_points](#) ()
Ends list of points, and draws.
- FL_EXPORT void [fl_end_polygon](#) ()
Ends convex filled polygon, and draws.

- FL_EXPORT const char * [fl_expand_text](#) (const char *from, char *buf, int maxbuf, double maxw, int &n, double &width, int wrap, int draw_symbols=0)
Copy from to buf, replacing unprintable characters with ^X and \nnn.
- [Fl_Font fl_font](#) ()
Returns the face set by the most recent call to [fl_font\(\)](#).
- FL_EXPORT void [fl_font](#) (Fl_Font face, Fl_Fontsize size)
Sets the current font, which is then used in various drawing routines.
- FL_EXPORT void [fl_frame](#) (const char *s, int x, int y, int w, int h)
Draws a series of line segments around the given box.
- FL_EXPORT void [fl_frame2](#) (const char *s, int x, int y, int w, int h)
Draws a series of line segments around the given box.
- FL_EXPORT void [fl_gap](#) ()
Call [fl_gap\(\)](#) to separate loops of the path.
- int [fl_height](#) (int, int size)
Dummy passthru function called only in [Fl_Text_Display](#) that simply returns the font height as given by the size parameter in the same call!
- FL_EXPORT int [fl_height](#) ()
Returns the recommended minimum line spacing for the current font.
- FL_EXPORT const char * [fl_latin1_to_local](#) (const char *t, int n=-1)
convert text from Windows/X11 latin1 character set to local encoding.
- FL_EXPORT void [fl_line](#) (int x, int y, int x1, int y1, int x2, int y2)
Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2).
- FL_EXPORT void [fl_line](#) (int x, int y, int x1, int y1)
Draws a line from (x,y) to (x1,y1).
- FL_EXPORT void [fl_line_style](#) (int style, int width=0, char *dashes=0)
Sets how to draw lines (the "pen").
- FL_EXPORT const char * [fl_local_to_latin1](#) (const char *t, int n=-1)
convert text from local encoding to Windowx/X11 latin1 character set.
- FL_EXPORT const char * [fl_local_to_mac_roman](#) (const char *t, int n=-1)
convert text from local encoding to Mac Roman character set.
- FL_EXPORT void [fl_loop](#) (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
Outlines a 4-sided polygon with lines.
- FL_EXPORT void [fl_loop](#) (int x, int y, int x1, int y1, int x2, int y2)
Outlines a 3-sided polygon with lines.

- FL_EXPORT const char * [fl_mac_roman_to_local](#) (const char *t, int n=-1)
convert text from Mac Roman character set to local encoding.
- FL_EXPORT void [fl_measure](#) (const char *str, int &x, int &y, int draw_symbols=1)
Measure how wide and tall the string will be when printed by the [fl_draw\(\)](#) function with align parameter.
- FL_EXPORT int [fl_measure_pixmap](#) (const char *const *cdata, int &w, int &h)
Get the dimensions of a pixmap.
- FL_EXPORT int [fl_measure_pixmap](#) (char *const *data, int &w, int &h)
Get the dimensions of a pixmap.
- FL_EXPORT void [fl_mult_matrix](#) (double a, double b, double c, double d, double x, double y)
Concatenates another transformation onto the current one.
- FL_EXPORT int [fl_not_clipped](#) (int x, int y, int w, int h)
Does the rectangle intersect the current clip region?
- FL_EXPORT void [fl_overlay_clear](#) ()
Erase a selection rectangle without drawing a new one.
- FL_EXPORT void [fl_overlay_rect](#) (int x, int y, int w, int h)
Draws a selection rectangle, erasing a previous one by XOR'ing it first.
- FL_EXPORT void [fl_pie](#) (int x, int y, int w, int h, double a1, double a2)
Draw filled ellipse sections using integer coordinates.
- FL_EXPORT void [fl_point](#) (int x, int y)
Draws a single pixel at the given coordinates.
- FL_EXPORT void [fl_polygon](#) (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)
Fills a 4-sided polygon.
- FL_EXPORT void [fl_polygon](#) (int x, int y, int x1, int y1, int x2, int y2)
Fills a 3-sided polygon.
- FL_EXPORT void [fl_pop_clip](#) ()
Restores the previous clip region.
- FL_EXPORT void [fl_pop_matrix](#) ()
Restores the current transformation matrix from the stack.
- FL_EXPORT void [fl_push_clip](#) (int x, int y, int w, int h)
Intersects the current clip region with a rectangle and pushes this new region onto the stack.
- FL_EXPORT void [fl_push_matrix](#) ()
Saves the current transformation matrix on the stack.
- FL_EXPORT void [fl_push_no_clip](#) ()
Pushes an empty clip region onto the stack so nothing will be clipped.

- FL_EXPORT `uchar * fl_read_image (uchar *p, int X, int Y, int W, int H, int alpha=0)`
Read an RGB(A) image from the current window or off-screen buffer.
- void `fl_rect (int x, int y, int w, int h, FL_Color c)`
Draws a 1-pixel border inside the given bounding box.
- FL_EXPORT void `fl_rect (int x, int y, int w, int h)`
Draws a 1-pixel border inside the given bounding box.
- FL_EXPORT void `fl_rectf (int x, int y, int w, int h, uchar r, uchar g, uchar b)`
Color a rectangle with "exactly" the passed r, g, b color.
- void `fl_rectf (int x, int y, int w, int h, FL_Color c)`
Colors a rectangle that exactly fills the given bounding box.
- FL_EXPORT void `fl_rectf (int x, int y, int w, int h)`
Colors a rectangle that exactly fills the given bounding box.
- FL_EXPORT void `fl_reset_spot (void)`
- FL_EXPORT void `fl_rotate (double d)`
Concatenates rotation transformation onto the current one.
- FL_EXPORT void `fl_rtl_draw (const char *, int n, int x, int y)`
Draws an array of n characters right to left starting at given location.
- FL_EXPORT void `fl_scale (double x)`
Concatenates scaling transformation onto the current one.
- FL_EXPORT void `fl_scale (double x, double y)`
Concatenates scaling transformation onto the current one.
- FL_EXPORT void `fl_scroll (int X, int Y, int W, int H, int dx, int dy, void(*draw_area)(void *, int, int, int), void *data)`
Scroll a rectangle and draw the newly exposed portions.
- FL_EXPORT void `fl_set_spot (int font, int size, int X, int Y, int W, int H, FL_Window *win=0)`
- FL_EXPORT void `fl_set_status (int X, int Y, int W, int H)`
- FL_EXPORT const char * `fl_shortcut_label (int shortcut)`
Get a human-readable string from a shortcut value.
- FL_Fontsize `fl_size ()`
Returns the size set by the most recent call to fl_font().
- FL_EXPORT void `fl_text_extents (const char *, int n, int &dx, int &dy, int &w, int &h)`
Determine the minimum pixel dimensions of a sequence of n characters.
- FL_EXPORT void `fl_text_extents (const char *, int &dx, int &dy, int &w, int &h)`
Determine the minimum pixel dimensions of a nul-terminated string.

- FL_EXPORT double [fl_transform_dx](#) (double x, double y)
Transforms distance using current transformation matrix.
- FL_EXPORT double [fl_transform_dy](#) (double x, double y)
Transforms distance using current transformation matrix.
- FL_EXPORT double [fl_transform_x](#) (double x, double y)
Transforms coordinate using the current transformation matrix.
- FL_EXPORT double [fl_transform_y](#) (double x, double y)
Transform coordinate using the current transformation matrix.
- FL_EXPORT void [fl_transformed_vertex](#) (double x, double y)
Adds coordinate pair to the vertex list without further transformations.
- FL_EXPORT void [fl_translate](#) (double x, double y)
Concatenates translation transformation onto the current one.
- FL_EXPORT void [fl_vertex](#) (double x, double y)
Adds a single vertex to the current path.
- FL_EXPORT double [fl_width](#) (FL_Unichar)
Return the typographical width of a single character .:
- FL_EXPORT double [fl_width](#) (const char *txt, int n)
Return the typographical width of a sequence of n characters.
- FL_EXPORT double [fl_width](#) (const char *txt)
Return the typographical width of a nul-terminated string.
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1, int y2, int x3)
Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2).
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1, int y2)
Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2).
- FL_EXPORT void [fl_xyline](#) (int x, int y, int x1)
Draws a horizontal line from (x,y) to (x1,y).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1, int x2, int y3)
Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1, int x2)
Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1).
- FL_EXPORT void [fl_yxline](#) (int x, int y, int y1)
Draws a vertical line from (x,y) to (x,y1).

Variables

- FL_EXPORT [Fl_Color](#) `fl_color_`
Current color for drawing operations.
- FL_EXPORT char `fl_draw_shortcut`
- FL_EXPORT [Fl_Font](#) `fl_font_`
current font index
- FL_EXPORT [Fl_Fontsize](#) `fl_size_`
current font size

31.8.1 Detailed Description

utility header to pull drawing functions together

31.9 fl_line_style.cxx File Reference

Line style drawing utility hiding different platforms.

```
#include <FL/Fl.H>
#include <FL/fl_draw.H>
#include <FL/x.H>
#include "flstring.h"
#include <stdio.h>
```

Functions

- void [fl_line_style](#) (int style, int width, char *dashes)
Sets how to draw lines (the "pen").

31.9.1 Detailed Description

Line style drawing utility hiding different platforms.

31.10 fl_rect.cxx File Reference

Drawing and clipping routines for rectangles.

```
#include <config.h>
#include <FL/Fl.H>
#include <FL/Fl_Widget.H>
#include <FL/fl_draw.H>
#include <FL/x.H>
```

Defines

- `#define STACK_MAX (STACK_SIZE - 1)`
- `#define STACK_SIZE 10`

Functions

- `int fl_clip_box (int x, int y, int w, int h, int &X, int &Y, int &W, int &H)`
Intersects the rectangle with the current clip region and returns the bounding box of the result.
- `Fl_Region fl_clip_region ()`
- `void fl_clip_region (Fl_Region r)`
Replaces the top of the clipping stack with a clipping region of any shape.
- `void fl_line (int x, int y, int x1, int y1, int x2, int y2)`
Draws a line from (x,y) to (x1,y1) and another from (x1,y1) to (x2,y2).
- `void fl_line (int x, int y, int x1, int y1)`
Draws a line from (x,y) to (x1,y1).
- `void fl_loop (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)`
Outlines a 4-sided polygon with lines.
- `void fl_loop (int x, int y, int x1, int y1, int x2, int y2)`
Outlines a 3-sided polygon with lines.
- `int fl_not_clipped (int x, int y, int w, int h)`
Does the rectangle intersect the current clip region?
- `void fl_point (int x, int y)`
Draws a single pixel at the given coordinates.
- `void fl_polygon (int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)`
Fills a 4-sided polygon.
- `void fl_polygon (int x, int y, int x1, int y1, int x2, int y2)`
Fills a 3-sided polygon.

- void `fl_pop_clip` ()
Restores the previous clip region.
- void `fl_push_clip` (int x, int y, int w, int h)
Intersects the current clip region with a rectangle and pushes this new region onto the stack.
- void `fl_push_no_clip` ()
Pushes an empty clip region onto the stack so nothing will be clipped.
- void `fl_rect` (int x, int y, int w, int h)
Draws a 1-pixel border inside the given bounding box.
- void `fl_rectf` (int x, int y, int w, int h)
Colors a rectangle that exactly fills the given bounding box.
- void `fl_restore_clip` ()
Undoes any clobbering of clip done by your program.
- void `fl_xyline` (int x, int y, int x1, int y2, int x3)
Draws a horizontal line from (x,y) to (x1,y), then a vertical from (x1,y) to (x1,y2) and then another horizontal from (x1,y2) to (x3,y2).
- void `fl_xyline` (int x, int y, int x1, int y2)
Draws a horizontal line from (x,y) to (x1,y), then vertical from (x1,y) to (x1,y2).
- void `fl_xyline` (int x, int y, int x1)
Draws a horizontal line from (x,y) to (x1,y).
- void `fl_yxline` (int x, int y, int y1, int x2, int y3)
Draws a vertical line from (x,y) to (x,y1) then a horizontal from (x,y1) to (x2,y1), then another vertical from (x2,y1) to (x2,y3).
- void `fl_yxline` (int x, int y, int y1, int x2)
Draws a vertical line from (x,y) to (x,y1), then a horizontal from (x,y1) to (x2,y1).
- void `fl_yxline` (int x, int y, int y1)
Draws a vertical line from (x,y) to (x,y1).
- `Fl_Region XRectangleRegion` (int x, int y, int w, int h)

Variables

- int `fl_clip_state_number` = 0

31.10.1 Detailed Description

Drawing and clipping routines for rectangles.

31.10.2 Function Documentation

31.10.2.1 `Fl_Region fl_clip_region ()`

Returns:

the current clipping region.

31.10.2.2 `void fl_clip_region (Fl_Region r)`

Replaces the top of the clipping stack with a clipping region of any shape.

`Fl_Region` is an operating system specific type.

Parameters:

$\leftarrow r$ clipping region

31.11 fl_types.h File Reference

This file contains simple "C"-style type definitions.

Typedefs

Miscellaneous

- typedef const char * [FL_CString](#)
Flexible length utf8 Unicode read-only string.
- typedef char * [FL_String](#)
Flexible length utf8 Unicode text.
- typedef unsigned int [FL_Unichar](#)
24-bit Unicode character + 8-bit indicator for keyboard flags
- typedef unsigned char [uchar](#)
unsigned char
- typedef unsigned long [ulong](#)
unsigned long

31.11.1 Detailed Description

This file contains simple "C"-style type definitions.

31.11.2 Typedef Documentation

31.11.2.1 typedef const char* FL_CString

Flexible length utf8 Unicode read-only string.

See also:

[FL_String](#)

31.11.2.2 typedef char* FL_String

Flexible length utf8 Unicode text.

Todo

FIXME: temporary (?) typedef to mark UTF8 and Unicode conversions

31.12 fl_vertex.cxx File Reference

Portable drawing code for drawing arbitrary shapes with simple 2D transformations.

```
#include <config.h>
#include <FL/fl_draw.H>
#include <FL/x.H>
#include <FL/Fl.H>
#include <FL/math.h>
#include <stdlib.h>
```

Defines

- `#define XPOINT XPoint`

Typedefs

- `typedef short COORD_T`

Enumerations

- `enum { LINE, LOOP, POLYGON, POINT_ }`

Functions

- `void fl_begin_complex_polygon ()`
Starts drawing a complex filled polygon.
- `void fl_begin_line ()`
Starts drawing a list of lines.
- `void fl_begin_loop ()`
Starts drawing a closed sequence of lines.
- `void fl_begin_points ()`
Starts drawing a list of points.
- `void fl_begin_polygon ()`
Starts drawing a convex filled polygon.
- `void fl_circle (double x, double y, double r)`
fl_circle() is equivalent to fl_arc(x,y,r,0,360), but may be faster.
- `void fl_end_complex_polygon ()`
Ends complex filled polygon, and draws.
- `void fl_end_line ()`

Ends list of lines, and draws.

- void [fl_end_loop](#) ()
Ends closed sequence of lines, and draws.
- void [fl_end_points](#) ()
Ends list of points, and draws.
- void [fl_end_polygon](#) ()
Ends convex filled polygon, and draws.
- void [fl_gap](#) ()
Call [fl_gap\(\)](#) to separate loops of the path.
- void [fl_mult_matrix](#) (double a, double b, double c, double d, double x, double y)
Concatenates another transformation onto the current one.
- void [fl_pop_matrix](#) ()
Restores the current transformation matrix from the stack.
- void [fl_push_matrix](#) ()
Saves the current transformation matrix on the stack.
- void [fl_rotate](#) (double d)
Concatenates rotation transformation onto the current one.
- void [fl_scale](#) (double x)
Concatenates scaling transformation onto the current one.
- void [fl_scale](#) (double x, double y)
Concatenates scaling transformation onto the current one.
- double [fl_transform_dx](#) (double x, double y)
Transforms distance using current transformation matrix.
- double [fl_transform_dy](#) (double x, double y)
Transforms distance using current transformation matrix.
- double [fl_transform_x](#) (double x, double y)
Transforms coordinate using the current transformation matrix.
- double [fl_transform_y](#) (double x, double y)
Transform coordinate using the current transformation matrix.
- void [fl_transformed_vertex](#) (double xf, double yf)
Adds coordinate pair to the vertex list without further transformations.
- void [fl_translate](#) (double x, double y)
Concatenates translation transformation onto the current one.

- void `fl_vertex` (double x, double y)
Adds a single vertex to the current path.

31.12.1 Detailed Description

Portable drawing code for drawing arbitrary shapes with simple 2D transformations.

Index

- ~Fl_Browser
 - Fl_Browser, [318](#)
- ~Fl_Check_Browser
 - Fl_Check_Browser, [358](#)
- ~Fl_Double_Window
 - Fl_Double_Window, [387](#)
- ~Fl_File_Chooser
 - Fl_File_Chooser, [398](#)
- ~Fl_Group
 - Fl_Group, [437](#)
- ~Fl_Help_View
 - Fl_Help_View, [453](#)
- ~Fl_Menu_Window
 - Fl_Menu_Window, [513](#)
- ~Fl_Preferences
 - Fl_Preferences, [539](#)
- ~Fl_RGB_Image
 - Fl_RGB_Image, [558](#)
- ~Fl_Scrollbar
 - Fl_Scrollbar, [573](#)
- ~Fl_Shared_Image
 - Fl_Shared_Image, [580](#)
- ~Fl_Text_Display
 - Fl_Text_Display, [623](#)
- ~Fl_Widget
 - Fl_Widget, [685](#)
- ~Fl_Window
 - Fl_Window, [716](#)
- _FL_DIAMOND_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_DIAMOND_UP_BOX
 - Enumerations.H, [734](#)
- _FL_EMBOSSED_LABEL
 - Enumerations.H, [737](#)
- _FL_ENGRAVED_LABEL
 - Enumerations.H, [737](#)
- _FL_GTK_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_DOWN_FRAME
 - Enumerations.H, [734](#)
- _FL_GTK_ROUND_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_ROUND_UP_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_THIN_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_THIN_DOWN_FRAME
 - Enumerations.H, [734](#)
- _FL_GTK_THIN_UP_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_THIN_UP_FRAME
 - Enumerations.H, [734](#)
- _FL_GTK_UP_BOX
 - Enumerations.H, [734](#)
- _FL_GTK_UP_FRAME
 - Enumerations.H, [734](#)
- _FL_ICON_LABEL
 - Enumerations.H, [738](#)
- _FL_IMAGE_LABEL
 - Enumerations.H, [738](#)
- _FL_MULTI_LABEL
 - Enumerations.H, [738](#)
- _FL_OFLAT_BOX
 - Enumerations.H, [734](#)
- _FL_OSHADOW_BOX
 - Enumerations.H, [734](#)
- _FL_OVAL_BOX
 - Enumerations.H, [734](#)
- _FL_OVAL_FRAME
 - Enumerations.H, [734](#)
- _FL_PLASTIC_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_DOWN_FRAME
 - Enumerations.H, [734](#)
- _FL_PLASTIC_ROUND_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_ROUND_UP_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_THIN_DOWN_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_THIN_UP_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_UP_BOX
 - Enumerations.H, [734](#)
- _FL_PLASTIC_UP_FRAME
 - Enumerations.H, [734](#)
- _FL_RFLAT_BOX
 - Enumerations.H, [734](#)
- _FL_ROUNDED_BOX
 - Enumerations.H, [734](#)

- [_FL_ROUNDED_FRAME](#)
 - [Enumerations.H, 734](#)
 - [_FL_ROUND_DOWN_BOX](#)
 - [Enumerations.H, 734](#)
 - [_FL_ROUND_UP_BOX](#)
 - [Enumerations.H, 734](#)
 - [_FL_RSHADOW_BOX](#)
 - [Enumerations.H, 734](#)
 - [_FL_SHADOW_BOX](#)
 - [Enumerations.H, 733](#)
 - [_FL_SHADOW_FRAME](#)
 - [Enumerations.H, 734](#)
 - [_FL_SHADOW_LABEL](#)
 - [Enumerations.H, 737](#)
- [_remove](#)
 - [Fl_Browser, 319](#)
- [activate](#)
 - [Fl_Menu_Item, 505](#)
 - [Fl_Widget, 686](#)
- [active](#)
 - [Fl_Menu_Item, 505](#)
 - [Fl_Widget, 686](#)
- [active_r](#)
 - [Fl_Widget, 686](#)
- [activevisible](#)
 - [Fl_Menu_Item, 506](#)
- [add](#)
 - [Fl_Browser, 319](#)
 - [Fl_Chart, 352](#)
 - [Fl_Check_Browser, 359](#)
 - [Fl_File_Icon, 404](#)
 - [Fl_Input_Choice, 477](#)
 - [Fl_Menu_, 489](#)
 - [Fl_Menu_Item, 506](#)
- [add_awake_handler_](#)
 - [Fl, 293](#)
- [add_check](#)
 - [Fl, 293](#)
- [add_color](#)
 - [Fl_File_Icon, 404](#)
- [add_default_key_bindings](#)
 - [Fl_Text_Editor, 637](#)
- [add_extra](#)
 - [Fl_File_Chooser, 398](#)
- [add_fd](#)
 - [Fl, 293](#)
- [add_handler](#)
 - [fl_events, 224](#)
- [add_idle](#)
 - [Fl, 294](#)
- [add_modify_callback](#)
 - [Fl_Text_Buffer, 607](#)
- [add_predelete_callback](#)
 - [Fl_Text_Buffer, 607](#)
- [add_timeout](#)
 - [Fl, 294](#)
- [add_vertex](#)
 - [Fl_File_Icon, 404](#)
- [align](#)
 - [Fl_Widget, 686, 687](#)
- [angle1](#)
 - [Fl_Dial, 384](#)
- [append](#)
 - [Fl_Text_Buffer, 607](#)
- [appendfile](#)
 - [Fl_Text_Buffer, 607](#)
- [arg](#)
 - [Fl, 294](#)
- [args](#)
 - [Fl, 295](#)
- [argument](#)
 - [Fl_Menu_Item, 506](#)
 - [Fl_Widget, 687](#)
- [array](#)
 - [Fl_Group, 437](#)
- [atclose](#)
 - [fl_windows, 221](#)
- [autosize](#)
 - [Fl_Chart, 352](#)
- [awake](#)
 - [fl_multithread, 265](#)
- [b](#)
 - [Fl_Color_Chooser, 376](#)
- [background](#)
 - [Fl, 296](#)
- [background2](#)
 - [Fl, 296](#)
- [bbox](#)
 - [Fl_Scroll, 569](#)
- [begin](#)
 - [Fl_Group, 437](#)
- [belowmouse](#)
 - [fl_events, 225](#)
- [bitmap](#)
 - [Fl_FormsBitmap, 415](#)
- [border](#)
 - [Fl_Window, 716](#)
- [bottomline](#)
 - [Fl_Browser, 319](#)
- [bound_key_function](#)
 - [Fl_Text_Editor, 637](#)
- [bounds](#)
 - [Fl_Chart, 353](#)
 - [Fl_Slider, 588](#)
 - [Fl_Valuator, 660](#)
- [box](#)

- Fl_Widget, 687
- box_dh
 - Fl, 296
- box_dw
 - Fl, 296
- box_dx
 - Fl, 297
- box_dy
 - Fl, 297
- buffer
 - Fl_Text_Display, 623
- Cairo support functions and classes, 270
- cairo_autolink_context
 - group_cairo, 270
- cairo_cc
 - group_cairo, 271
- calc_line_starts
 - Fl_Text_Display, 623
- callback
 - Fl_Menu_Item, 506
 - Fl_Widget, 688
- can_do
 - Fl_Gl_Window, 426
- can_do_overlay
 - Fl_Gl_Window, 426
- CHANGED
 - Fl_Widget, 685
- changed
 - Fl_Input_Choice, 477
 - Fl_Widget, 689
- character
 - Fl_Text_Buffer, 607
- character_width
 - Fl_Text_Buffer, 607
- check
 - Fl, 297
 - Fl_Menu_Item, 506
- check_all
 - Fl_Check_Browser, 359
- check_none
 - Fl_Check_Browser, 359
- checkbox
 - Fl_Menu_Item, 506
- checked
 - Fl_Check_Browser, 359
 - Fl_Menu_Item, 506
- child
 - Fl_Group, 437
- clamp
 - Fl_Valuator, 660
- clear
 - Fl_Button, 344
 - Fl_Check_Browser, 359
 - Fl_File_Icon, 404
 - Fl_Group, 438
 - Fl_Input_Choice, 477
 - Fl_Menu_, 490
 - Fl_Menu_Item, 507
 - Fl_Scroll, 569
- clear_border
 - Fl_Window, 716
- clear_changed
 - Fl_Input_Choice, 477
 - Fl_Widget, 689
- clear_damage
 - Fl_Widget, 689
- clear_output
 - Fl_Widget, 689
- clear_overlay
 - Fl_Menu_Window, 513
- clear_rectangular
 - Fl_Text_Buffer, 608
- clear_selection
 - Fl_Help_View, 453
- clear_visible
 - Fl_Widget, 690
- clear_visible_focus
 - Fl_Widget, 690
- clear_widget_pointer
 - fl_del_widget, 268
- clip_children
 - Fl_Group, 438
- col
 - FL_CHART_ENTRY, 356
- color
 - Fl_File_Chooser, 398, 399
 - Fl_Tooltip, 655
 - Fl_Widget, 690
- Color & Font functions, 237
- color2
 - Fl_Widget, 691
- color_average
 - Fl_Image, 459
 - Fl_Pixmap, 527
 - Fl_RGB_Image, 558
 - Fl_Shared_Image, 580
 - Fl_Tiled_Image, 648
- column_char
 - Fl_Browser, 319
- column_widths
 - Fl_Browser, 319, 320
- Common Dialogs classes and functions, 272
- compose
 - fl_events, 225
- compose_reset
 - fl_events, 225
- contains

- Fl_Widget, 691
- context
 - Fl_Gl_Window, 426
- context_valid
 - Fl_Gl_Window, 426
- COPIED_LABEL
 - Fl_Widget, 685
- copy
 - Fl_Bitmap, 309
 - fl_clipboard, 232
 - Fl_Image, 459
 - Fl_Input_, 469
 - Fl_Menu_, 490
 - Fl_Pixmap, 527
 - Fl_RGB_Image, 558
 - Fl_Shared_Image, 580, 581
 - Fl_Tiled_Image, 648
- copy_cuts
 - Fl_Input_, 469
- copy_label
 - Fl_Widget, 691
 - Fl_Window, 716
- count
 - Fl_File_Chooser, 399
 - Fl_Image, 459
- count_displayed_characters
 - Fl_Text_Buffer, 608
- count_lines
 - Fl_Text_Buffer, 608
 - Fl_Text_Display, 623
- current
 - Fl_Group, 438
 - Fl_Tooltip, 655
 - Fl_Window, 717
- current_
 - Fl_Window, 722
- cursor
 - Fl_Window, 717
- cursor_color
 - Fl_Input_, 470
 - Fl_Text_Display, 623, 624
 - Fl_Value_Input, 666
- cursor_style
 - Fl_Text_Display, 624
- cut
 - Fl_Input_, 470
- d
 - Fl_Image, 459, 460
- damage
 - Fl, 297
 - Fl_Widget, 692
- damage_resize
 - Fl_Widget, 692
- data
 - Fl_Browser, 320
 - Fl_Image, 460
- deactivate
 - Fl_Menu_Item, 507
 - Fl_Repeat_Button, 552
 - Fl_Widget, 692
- default_atclose
 - fl_windows, 220
- default_callback
 - Fl_Widget, 693
- default_cursor
 - Fl_Window, 717
- default_key_function
 - Fl_Text_Editor, 637
- deimage
 - Fl_Widget, 693
- delay
 - Fl_Tooltip, 655, 656
- delete_widget
 - fl_del_widget, 268
- deleted
 - Fl_Widget_Tracker, 711
- deleteEntry
 - Fl_Preferences, 539
- deleteGroup
 - Fl_Preferences, 539
- deleting
 - Fl_Browser_, 336
- desaturate
 - Fl_Image, 460
 - Fl_Pixmap, 527
 - Fl_RGB_Image, 559
 - Fl_Shared_Image, 581
 - Fl_Tiled_Image, 648
- deselect
 - Fl_Browser_, 336
- direction
 - Fl_Timer, 651
- directory
 - Fl_File_Chooser, 399
 - Fl_Help_View, 453
- disable
 - Fl_Tooltip, 656
- display
 - Fl, 297
 - Fl_Browser, 320
 - Fl_Browser_, 337
- display_insert
 - Fl_Text_Display, 624
- dnd
 - fl_clipboard, 232
- dnd_text_ops
 - Fl, 298

- do_callback
 - Fl_Menu_Item, 507
 - Fl_Widget, 694
- do_widget_deletion
 - fl_del_widget, 268
- down_box
 - Fl_Button, 344
 - Fl_File_Input, 409
 - Fl_Menu_, 490
- draw
 - Fl_Adjuster, 306
 - Fl_Bitmap, 309, 310
 - Fl_Box, 313
 - Fl_Browser_, 337
 - Fl_Button, 344
 - Fl_Chart, 353
 - Fl_Choice, 365
 - Fl_Clock_Output, 371
 - Fl_Counter, 381
 - Fl_Dial, 384
 - Fl_File_Icon, 404
 - Fl_FormsBitmap, 415
 - Fl_FormsPixmap, 418
 - Fl_Free, 420
 - Fl_Gl_Window, 426
 - Fl_Glut_Window, 432
 - Fl_Group, 438
 - Fl_Image, 460
 - Fl_Input, 464
 - Fl_Label, 482
 - Fl_Light_Button, 484
 - Fl_Menu_Bar, 495
 - Fl_Menu_Button, 498
 - Fl_Menu_Item, 507
 - Fl_Pack, 524
 - Fl_Pixmap, 527
 - Fl_Positioner, 533
 - Fl_Progress, 551
 - Fl_Return_Button, 555
 - Fl_RGB_Image, 559
 - Fl_Roller, 562
 - Fl_Scroll, 569
 - Fl_Scrollbar, 573
 - Fl_Shared_Image, 581
 - Fl_Slider, 588
 - Fl_Tabs, 597
 - Fl_Text_Display, 624
 - Fl_Tiled_Image, 648, 649
 - Fl_Timer, 651
 - Fl_Value_Input, 666
 - Fl_Value_Output, 672
 - Fl_Value_Slider, 675
 - Fl_Widget, 694
 - Fl_Window, 717
- draw_box
 - Fl_Widget, 695
- draw_box_active
 - Fl, 298
- draw_child
 - Fl_Group, 439
- draw_children
 - Fl_Group, 439
- draw_empty
 - Fl_Image, 460
- draw_label
 - Fl_Widget, 695
- draw_line_numbers
 - Fl_Text_Display, 624
- draw_outside_label
 - Fl_Group, 439
- draw_overlay
 - Fl_Glut_Window, 432
- draw_range
 - Fl_Text_Display, 624
- draw_string
 - Fl_Text_Display, 625
- draw_text
 - Fl_Text_Display, 625
- Drawing functions, 246
- drawtext
 - Fl_Input_, 470
- enable
 - Fl_Tooltip, 656
- enabled
 - Fl_Tooltip, 656
- end
 - Fl_Group, 439
- enter_area
 - Fl_Tooltip, 656
- entries
 - Fl_Preferences, 539
- entry
 - Fl_Preferences, 539
- entryExists
 - Fl_Preferences, 540
- Enumerations.H, 727
 - _FL_DIAMOND_DOWN_BOX, 734
 - _FL_DIAMOND_UP_BOX, 734
 - _FL_EMBOSSSED_LABEL, 737
 - _FL_ENGRAVED_LABEL, 737
 - _FL_GTK_DOWN_BOX, 734
 - _FL_GTK_DOWN_FRAME, 734
 - _FL_GTK_ROUND_DOWN_BOX, 734
 - _FL_GTK_ROUND_UP_BOX, 734
 - _FL_GTK_THIN_DOWN_BOX, 734
 - _FL_GTK_THIN_DOWN_FRAME, 734
 - _FL_GTK_THIN_UP_BOX, 734

- [_FL_GTK_THIN_UP_FRAME, 734](#)
- [_FL_GTK_UP_BOX, 734](#)
- [_FL_GTK_UP_FRAME, 734](#)
- [_FL_ICON_LABEL, 738](#)
- [_FL_IMAGE_LABEL, 738](#)
- [_FL_MULTI_LABEL, 738](#)
- [_FL_OFLAT_BOX, 734](#)
- [_FL_OSHADOW_BOX, 734](#)
- [_FL_OVAL_BOX, 734](#)
- [_FL_OVAL_FRAME, 734](#)
- [_FL_PLASTIC_DOWN_BOX, 734](#)
- [_FL_PLASTIC_DOWN_FRAME, 734](#)
- [_FL_PLASTIC_ROUND_DOWN_BOX, 734](#)
- [_FL_PLASTIC_ROUND_UP_BOX, 734](#)
- [_FL_PLASTIC_THIN_DOWN_BOX, 734](#)
- [_FL_PLASTIC_THIN_UP_BOX, 734](#)
- [_FL_PLASTIC_UP_BOX, 734](#)
- [_FL_PLASTIC_UP_FRAME, 734](#)
- [_FL_RFLAT_BOX, 734](#)
- [_FL_ROUNDED_BOX, 734](#)
- [_FL_ROUNDED_FRAME, 734](#)
- [_FL_ROUND_DOWN_BOX, 734](#)
- [_FL_ROUND_UP_BOX, 734](#)
- [_FL_RSHADOW_BOX, 734](#)
- [_FL_SHADOW_BOX, 733](#)
- [_FL_SHADOW_FRAME, 734](#)
- [_FL_SHADOW_LABEL, 737](#)
- [FL_ACTIVATE, 736](#)
- [FL_BACKGROUND2_COLOR, 735](#)
- [FL_BORDER_BOX, 733](#)
- [FL_BORDER_FRAME, 733](#)
- [FL_CLOSE, 736](#)
- [FL_DEACTIVATE, 736](#)
- [FL_DND_DRAG, 737](#)
- [FL_DND_ENTER, 737](#)
- [FL_DND_LEAVE, 737](#)
- [FL_DND_RELEASE, 737](#)
- [FL_DOWN_BOX, 733](#)
- [FL_DOWN_FRAME, 733](#)
- [FL_DRAG, 735](#)
- [FL_EMBOSSED_BOX, 733](#)
- [FL_EMBOSSED_FRAME, 733](#)
- [FL_ENGRAVED_BOX, 733](#)
- [FL_ENGRAVED_FRAME, 733](#)
- [FL_ENTER, 735](#)
- [FL_FLAT_BOX, 733](#)
- [FL_FOCUS, 735](#)
- [FL_FOREGROUND_COLOR, 735](#)
- [FL_FREE_BOXTYPE, 734](#)
- [FL_FREE_LABELTYPE, 738](#)
- [FL_HIDE, 736](#)
- [FL_INACTIVE_COLOR, 735](#)
- [FL_KEYBOARD, 736](#)
- [FL_KEYDOWN, 736](#)
- [FL_KEYUP, 736](#)
- [FL_LEAVE, 735](#)
- [FL_MOUSEWHEEL, 737](#)
- [FL_MOVE, 736](#)
- [FL_NO_BOX, 733](#)
- [FL_NO_EVENT, 735](#)
- [FL_NO_LABEL, 737](#)
- [FL_NORMAL_LABEL, 737](#)
- [FL_PASTE, 737](#)
- [FL_PUSH, 735](#)
- [FL_RELEASE, 735](#)
- [FL_SELECTION_COLOR, 735](#)
- [FL_SELECTIONCLEAR, 737](#)
- [FL_SHORTCUT, 736](#)
- [FL_SHOW, 737](#)
- [FL_THIN_DOWN_BOX, 733](#)
- [FL_THIN_DOWN_FRAME, 733](#)
- [FL_THIN_UP_BOX, 733](#)
- [FL_THIN_UP_FRAME, 733](#)
- [FL_UNFOCUS, 736](#)
- [FL_UP_BOX, 733](#)
- [FL_UP_FRAME, 733](#)
- [FL_WHEN_CHANGED, 738](#)
- [FL_WHEN_ENTER_KEY, 738](#)
- [FL_WHEN_ENTER_KEY_ALWAYS, 738](#)
- [FL_WHEN_ENTER_KEY_CHANGED, 738](#)
- [FL_WHEN_NEVER, 738](#)
- [FL_WHEN_NOT_CHANGED, 738](#)
- [FL_WHEN_RELEASE, 738](#)
- [FL_WHEN_RELEASE_ALWAYS, 738](#)
- [Fl_Align, 733](#)
- [FL_ALIGN_BOTTOM, 739](#)
- [FL_ALIGN_CENTER, 739](#)
- [FL_ALIGN_CLIP, 739](#)
- [FL_ALIGN_IMAGE_OVER_TEXT, 739](#)
- [FL_ALIGN_INSIDE, 739](#)
- [FL_ALIGN_LEFT, 739](#)
- [FL_ALIGN_RIGHT, 739](#)
- [FL_ALIGN_TEXT_OVER_IMAGE, 739](#)
- [FL_ALIGN_TOP, 739](#)
- [FL_ALIGN_WRAP, 739](#)
- [fl_box, 738](#)
- [Fl_Boxtype, 733](#)
- [Fl_Color, 734](#)
- [fl_down, 738](#)
- [Fl_Event, 735](#)
- [Fl_Font, 733](#)
- [Fl_Fontsize, 733](#)
- [fl_frame, 738](#)
- [Fl_Labeltype, 737](#)
- [FL_MAJOR_VERSION, 732](#)
- [FL_MINOR_VERSION, 732](#)
- [FL_PATCH_VERSION, 732](#)
- [FL_VERSION, 732](#)

- Fl_When, 738
- error
 - group_comdlg, 277
- errorcolor
 - Fl_File_Input, 409
- event
 - fl_events, 225
- event_alt
 - fl_events, 226
- event_button
 - fl_events, 226
- event_button1
 - fl_events, 226
- event_button2
 - fl_events, 226
- event_button3
 - fl_events, 226
- event_buttons
 - fl_events, 226
- event_clicks
 - fl_events, 227
- event_ctrl
 - fl_events, 227
- event_dx
 - fl_events, 227
- event_dy
 - fl_events, 227
- event_inside
 - fl_events, 227
- event_is_click
 - fl_events, 228
- event_key
 - fl_events, 228
- event_length
 - fl_events, 229
- event_original_key
 - fl_events, 229
- event_shift
 - fl_events, 229
- event_state
 - fl_events, 229
- event_text
 - fl_events, 229
- event_x_root
 - fl_events, 230
- event_y_root
 - fl_events, 230
- Events handling functions, 222
- exists
 - Fl_Widget_Tracker, 711
- expand_character
 - Fl_Text_Buffer, 608
- fatal
 - group_comdlg, 278
- File names and URI utility functions, 279
- filename
 - Fl_Help_View, 453
- filenames
 - Fl_File_Sort_F, 279
 - fl_filename_absolute, 280
 - fl_filename_expand, 280
 - fl_filename_ext, 280
 - fl_filename_isdir, 280
 - fl_filename_match, 280
 - fl_filename_name, 281
 - fl_filename_relative, 281
 - fl_filename_setext, 281
- filetype
 - Fl_File_Browser, 391
- filter
 - Fl_File_Browser, 391
 - Fl_File_Chooser, 399
- filter_value
 - Fl_File_Chooser, 399
- find
 - Fl_File_Icon, 405
 - Fl_Group, 439
- find_item
 - Fl_Browser_, 337
 - Fl_Menu_, 490
- find_line
 - Fl_Browser, 321
- find_line_end
 - Fl_Text_Display, 625
- find_shortcut
 - Fl_Menu_Item, 507
- find_wrap_range
 - Fl_Text_Display, 625
- findchar_backward
 - Fl_Text_Buffer, 608
- findchar_forward
 - Fl_Text_Buffer, 609
- findchars_backward
 - Fl_Text_Buffer, 609
- findchars_forward
 - Fl_Text_Buffer, 609
- first
 - Fl_File_Icon, 405
 - Fl_Menu_Item, 507, 508
- first_window
 - fl_windows, 220
- Fl, 283
 - add_awake_handler_, 293
 - add_check, 293
 - add_fd, 293
 - add_idle, 294
 - add_timeout, 294

- arg, [294](#)
- args, [295](#)
- background, [296](#)
- background2, [296](#)
- box_dh, [296](#)
- box_dw, [296](#)
- box_dx, [297](#)
- box_dy, [297](#)
- check, [297](#)
- damage, [297](#)
- display, [297](#)
- dnd_text_ops, [298](#)
- draw_box_active, [298](#)
- flush, [298](#)
- foreground, [298](#)
- get_awake_handler_, [298](#)
- get_boxtype, [298](#)
- get_system_colors, [298](#)
- gl_visual, [299](#)
- help, [303](#)
- idle, [303](#)
- own_colormap, [299](#)
- ready, [299](#)
- release, [299](#)
- reload_scheme, [299](#)
- remove_check, [300](#)
- remove_fd, [300](#)
- remove_timeout, [300](#)
- repeat_timeout, [300](#)
- run, [300](#)
- scheme, [300](#)
- set_boxtype, [301](#)
- set_idle, [301](#)
- set_labeltype, [301](#)
- version, [301](#)
- visible_focus, [301](#)
- visual, [302](#)
- wait, [302](#)
- FL_ACTIVATE
 - Enumerations.H, [736](#)
- FL_BACKGROUND2_COLOR
 - Enumerations.H, [735](#)
- FL_BORDER_BOX
 - Enumerations.H, [733](#)
- FL_BORDER_FRAME
 - Enumerations.H, [733](#)
- FL_CAP_FLAT
 - fl_drawings, [252](#)
- FL_CAP_ROUND
 - fl_drawings, [252](#)
- FL_CAP_SQUARE
 - fl_drawings, [252](#)
- FL_CLOSE
 - Enumerations.H, [736](#)
- FL_DASH
 - fl_drawings, [252](#)
- FL_DASHDOT
 - fl_drawings, [252](#)
- FL_DASHDOTDOT
 - fl_drawings, [252](#)
- FL_DEACTIVATE
 - Enumerations.H, [736](#)
- FL_DND_DRAG
 - Enumerations.H, [737](#)
- FL_DND_ENTER
 - Enumerations.H, [737](#)
- FL_DND_LEAVE
 - Enumerations.H, [737](#)
- FL_DND_RELEASE
 - Enumerations.H, [737](#)
- FL_DOT
 - fl_drawings, [252](#)
- FL_DOWN_BOX
 - Enumerations.H, [733](#)
- FL_DOWN_FRAME
 - Enumerations.H, [733](#)
- FL_DRAG
 - Enumerations.H, [735](#)
- fl_drawings
 - FL_CAP_FLAT, [252](#)
 - FL_CAP_ROUND, [252](#)
 - FL_CAP_SQUARE, [252](#)
 - FL_DASH, [252](#)
 - FL_DASHDOT, [252](#)
 - FL_DASHDOTDOT, [252](#)
 - FL_DOT, [252](#)
 - FL_JOIN_BEVEL, [252](#)
 - FL_JOIN_MITER, [252](#)
 - FL_JOIN_ROUND, [252](#)
 - FL_SOLID, [252](#)
- FL_EMBOSSSED_BOX
 - Enumerations.H, [733](#)
- FL_EMBOSSSED_FRAME
 - Enumerations.H, [733](#)
- FL_ENGRAVED_BOX
 - Enumerations.H, [733](#)
- FL_ENGRAVED_FRAME
 - Enumerations.H, [733](#)
- FL_ENTER
 - Enumerations.H, [735](#)
- FL_FLAT_BOX
 - Enumerations.H, [733](#)
- FL_FOCUS
 - Enumerations.H, [735](#)
- FL_FOREGROUND_COLOR
 - Enumerations.H, [735](#)
- FL_FREE_BOXTYPE
 - Enumerations.H, [734](#)

- FL_FREE_LABELTYPE
 - Enumerations.H, [738](#)
- FL_HIDE
 - Enumerations.H, [736](#)
- FL_INACTIVE_COLOR
 - Enumerations.H, [735](#)
- FL_JOIN_BEVEL
 - fl_drawings, [252](#)
- FL_JOIN_MITER
 - fl_drawings, [252](#)
- FL_JOIN_ROUND
 - fl_drawings, [252](#)
- FL_KEYBOARD
 - Enumerations.H, [736](#)
- FL_KEYDOWN
 - Enumerations.H, [736](#)
- FL_KEYUP
 - Enumerations.H, [736](#)
- FL_LEAVE
 - Enumerations.H, [735](#)
- FL_MOUSEWHEEL
 - Enumerations.H, [737](#)
- FL_MOVE
 - Enumerations.H, [736](#)
- FL_NO_BOX
 - Enumerations.H, [733](#)
- FL_NO_EVENT
 - Enumerations.H, [735](#)
- FL_NO_LABEL
 - Enumerations.H, [737](#)
- FL_NORMAL_LABEL
 - Enumerations.H, [737](#)
- FL_PASTE
 - Enumerations.H, [737](#)
- Fl_Preferences
 - SYSTEM, [538](#)
 - USER, [538](#)
- FL_PUSH
 - Enumerations.H, [735](#)
- FL_RELEASE
 - Enumerations.H, [735](#)
- FL_SELECTION_COLOR
 - Enumerations.H, [735](#)
- FL_SELECTIONCLEAR
 - Enumerations.H, [737](#)
- FL_SHORTCUT
 - Enumerations.H, [736](#)
- FL_SHOW
 - Enumerations.H, [737](#)
- FL_SOLID
 - fl_drawings, [252](#)
- FL_THIN_DOWN_BOX
 - Enumerations.H, [733](#)
- FL_THIN_DOWN_FRAME
 - Enumerations.H, [733](#)
- FL_THIN_UP_BOX
 - Enumerations.H, [733](#)
- FL_THIN_UP_FRAME
 - Enumerations.H, [733](#)
- FL_UNFOCUS
 - Enumerations.H, [736](#)
- FL_UP_BOX
 - Enumerations.H, [733](#)
- FL_UP_FRAME
 - Enumerations.H, [733](#)
- FL_WHEN_CHANGED
 - Enumerations.H, [738](#)
- FL_WHEN_ENTER_KEY
 - Enumerations.H, [738](#)
- FL_WHEN_ENTER_KEY_ALWAYS
 - Enumerations.H, [738](#)
- FL_WHEN_ENTER_KEY_CHANGED
 - Enumerations.H, [738](#)
- FL_WHEN_NEVER
 - Enumerations.H, [738](#)
- FL_WHEN_NOT_CHANGED
 - Enumerations.H, [738](#)
- FL_WHEN_RELEASE
 - Enumerations.H, [738](#)
- FL_WHEN_RELEASE_ALWAYS
 - Enumerations.H, [738](#)
- Fl_Widget
 - CHANGED, [685](#)
 - COPIED_LABEL, [685](#)
 - INACTIVE, [685](#)
 - INVISIBLE, [685](#)
 - OUTPUT, [685](#)
 - SHORTCUT_LABEL, [685](#)
 - VISIBLE_FOCUS, [685](#)
- fl_add_symbol
 - fl_drawings, [252](#)
- Fl_Adjuster, [305](#)
 - draw, [306](#)
 - Fl_Adjuster, [306](#)
 - Fl_Adjuster, [306](#)
 - handle, [306](#)
 - soft, [307](#)
- fl_alert
 - group_comdlg, [273](#)
- Fl_Align
 - Enumerations.H, [733](#)
- FL_ALIGN_BOTTOM
 - Enumerations.H, [739](#)
- FL_ALIGN_CENTER
 - Enumerations.H, [739](#)
- FL_ALIGN_CLIP
 - Enumerations.H, [739](#)
- FL_ALIGN_IMAGE_OVER_TEXT

- Enumerations.H, 739
- FL_ALIGN_INSIDE
 - Enumerations.H, 739
- FL_ALIGN_LEFT
 - Enumerations.H, 739
- FL_ALIGN_RIGHT
 - Enumerations.H, 739
- FL_ALIGN_TEXT_OVER_IMAGE
 - Enumerations.H, 739
- FL_ALIGN_TOP
 - Enumerations.H, 739
- FL_ALIGN_WRAP
 - Enumerations.H, 739
- fl_arc
 - fl_drawings, 252
- fl_arc.cxx, 741
- fl_arci.cxx, 742
- fl_ask
 - group_comdlg, 273
- fl_attributes
 - fl_color, 239, 240
 - fl_color_average, 240
 - fl_contrast, 240
 - fl_font, 240
 - fl_height, 241
 - fl_latin1_to_local, 241
 - fl_local_to_latin1, 241
 - fl_local_to_mac_roman, 241
 - fl_mac_roman_to_local, 242
 - fl_size, 242
 - fl_text_extents, 242
 - fl_width, 242
 - fl_xpixel, 243
 - free_color, 243
 - get_color, 243
 - get_font, 244
 - get_font_name, 244
 - get_font_sizes, 244
 - set_color, 244
 - set_font, 245
 - set_fonts, 245
- fl_begin_complex_polygon
 - fl_drawings, 253
- fl_begin_points
 - fl_drawings, 253
- Fl_Bitmap, 308
 - copy, 309
 - draw, 309, 310
 - Fl_Bitmap, 309
 - Fl_Bitmap, 309
 - label, 310
 - uncache, 310
- Fl_BMP_Image, 311
 - Fl_BMP_Image, 311
- Fl_BMP_Image, 311
- Fl_Box, 312
 - draw, 313
 - Fl_Box, 312
 - Fl_Box, 312
 - handle, 313
- fl_box
 - Enumerations.H, 738
- Fl_Boxtype
 - Enumerations.H, 733
- fl_boxtype.cxx, 743
 - fl_internal_boxtype, 744
 - fl_rectbound, 744
- Fl_Browser, 314
 - ~Fl_Browser, 318
 - _remove, 319
 - add, 319
 - bottomline, 319
 - column_char, 319
 - column_widths, 319, 320
 - data, 320
 - display, 320
 - find_line, 321
 - Fl_Browser, 318
 - Fl_Browser, 318
 - format_char, 321
 - full_height, 322
 - hide, 322
 - incr_height, 322
 - insert, 323
 - item_at, 323
 - item_draw, 323
 - item_first, 324
 - item_height, 324
 - item_last, 324
 - item_next, 325
 - item_prev, 325
 - item_select, 325
 - item_selected, 325
 - item_swap, 326
 - item_text, 326
 - item_width, 326
 - lineno, 326
 - lineposition, 327
 - load, 327
 - make_visible, 327
 - middleline, 327
 - move, 327
 - remove, 328
 - replace, 328
 - select, 328
 - selected, 328
 - show, 328, 329
 - size, 329

- swap, 329
- text, 330
- topline, 330
- value, 330
- visible, 331
- Fl_Browser_, 332
 - deleting, 336
 - deselect, 336
 - display, 337
 - draw, 337
 - find_item, 337
 - Fl_Browser_, 336
 - Fl_Browser_, 336
 - full_height, 337
 - full_width, 337
 - handle, 337
 - has_scrollbar, 337
 - incr_height, 338
 - inserting, 338
 - item_first, 338
 - item_height, 338
 - item_last, 338
 - item_next, 338
 - item_prev, 338
 - item_quick_height, 339
 - item_select, 339
 - item_selected, 339
 - item_width, 339
 - new_list, 339
 - redraw_lines, 339
 - replacing, 339
 - resize, 339
 - scrollbar, 341
 - scrollbar_left, 340
 - scrollbar_right, 340
 - select, 340
 - select_only, 340
 - selection, 340
 - sort, 340
 - top, 340
- Fl_Button, 342
 - clear, 344
 - down_box, 344
 - draw, 344
 - Fl_Button, 344
 - Fl_Button, 344
 - handle, 345
 - set, 345
 - shortcut, 345, 346
 - value, 346
- Fl_Cairo_State, 347
- Fl_Cairo_Window, 348
 - set_draw_cb, 349
- fl_can_do_alpha_blending
 - fl_drawings, 253
- Fl_Chart, 350
 - add, 352
 - autosize, 352
 - bounds, 353
 - draw, 353
 - Fl_Chart, 352
 - Fl_Chart, 352
 - insert, 353
 - maxsize, 353
 - replace, 354
 - size, 354
 - textcolor, 354
 - textfont, 354
 - textsize, 354
- FL_CHART_ENTRY, 356
 - col, 356
 - str, 356
 - val, 356
- Fl_Check_Browser, 357
 - ~Fl_Check_Browser, 358
 - add, 359
 - check_all, 359
 - check_none, 359
 - checked, 359
 - clear, 359
 - Fl_Check_Browser, 358
 - Fl_Check_Browser, 358
 - handle, 359
 - nchecked, 359
 - nitems, 359
 - remove, 359
 - set_checked, 360
 - text, 360
 - value, 360
- Fl_Check_Button, 361
 - Fl_Check_Button, 362
 - Fl_Check_Button, 362
- Fl_Choice, 363
 - draw, 365
 - Fl_Choice, 364
 - Fl_Choice, 364
 - handle, 365
 - value, 365, 366
- fl_choice
 - group_comdlg, 273
- fl_circle
 - fl_drawings, 253
- fl_clip_box
 - fl_drawings, 254
- fl_clip_region
 - fl_rect.cxx, 760
- fl_clipboard
 - copy, 232

- dnd, 232
- paste, 233
- selection, 233
- selection_owner, 233
- Fl_Clock, 367
 - Fl_Clock, 368
 - Fl_Clock, 368
 - handle, 369
 - update, 369
- Fl_Clock_Output, 370
 - draw, 371
 - Fl_Clock_Output, 371
 - Fl_Clock_Output, 371
 - hour, 372
 - minute, 372
 - second, 372
 - value, 372
- Fl_Color
 - Enumerations.H, 734
- fl_color
 - fl_attributes, 239, 240
- fl_color.cxx, 745
- fl_color_average
 - fl_attributes, 240
- Fl_Color_Chooser, 374
 - b, 376
 - Fl_Color_Chooser, 376
 - Fl_Color_Chooser, 376
 - g, 376
 - hsv, 376
 - hsv2rgb, 376
 - hue, 377
 - r, 377
 - rgb, 377
 - rgb2hsv, 377
 - saturation, 377
 - value, 377
- fl_color_chooser
 - group_comdlg, 274
- Fl_Color_Chooser.H, 747
- fl_contrast
 - fl_attributes, 240
- Fl_Counter, 379
 - draw, 381
 - Fl_Counter, 381
 - Fl_Counter, 381
 - handle, 381
 - lstep, 382
 - step, 382
- Fl_CString
 - fl_types.h, 761
- fl_cursor
 - fl_drawings, 254
- fl_curve
 - fl_drawings, 254
- fl_curve.cxx, 748
- fl_del_widget
 - clear_widget_pointer, 268
 - delete_widget, 268
 - do_widget_deletion, 268
 - release_widget_pointer, 268
 - watch_widget_pointer, 269
- Fl_Dial, 383
 - angle1, 384
 - draw, 384
 - Fl_Dial, 384
 - Fl_Dial, 384
 - handle, 385
- fl_dir_chooser
 - group_comdlg, 275
- Fl_Double_Window, 386
 - ~Fl_Double_Window, 387
 - flush, 387
 - hide, 387
 - resize, 388
 - show, 388
- fl_down
 - Enumerations.H, 738
- fl_draw
 - fl_drawings, 254, 255
- fl_draw.H, 749
- fl_draw_box
 - fl_drawings, 255
- fl_draw_image
 - fl_drawings, 255, 256
- Fl_Draw_Image_Cb
 - fl_drawings, 251
- fl_draw_image_mono
 - fl_drawings, 256
- fl_draw_pixmap
 - fl_drawings, 257
- fl_draw_symbol
 - fl_drawings, 257
- fl_drawings
 - fl_add_symbol, 252
 - fl_arc, 252
 - fl_begin_complex_polygon, 253
 - fl_begin_points, 253
 - fl_can_do_alpha_blending, 253
 - fl_circle, 253
 - fl_clip_box, 254
 - fl_cursor, 254
 - fl_curve, 254
 - fl_draw, 254, 255
 - fl_draw_box, 255
 - fl_draw_image, 255, 256
 - Fl_Draw_Image_Cb, 251
 - fl_draw_image_mono, 256

- [fl_draw_pixmap, 257](#)
- [fl_draw_symbol, 257](#)
- [fl_expand_text, 257](#)
- [fl_frame, 258](#)
- [fl_frame2, 258](#)
- [fl_gap, 258](#)
- [fl_line_style, 258](#)
- [fl_measure, 259](#)
- [fl_measure_pixmap, 259](#)
- [fl_mult_matrix, 259](#)
- [fl_not_clipped, 260](#)
- [fl_pie, 260](#)
- [fl_polygon, 260](#)
- [fl_pop_clip, 260](#)
- [fl_push_clip, 261](#)
- [fl_push_matrix, 261](#)
- [fl_read_image, 261](#)
- [fl_rectf, 261](#)
- [fl_reset_spot, 261](#)
- [fl_rotate, 262](#)
- [fl_scale, 262](#)
- [fl_scroll, 262](#)
- [fl_set_spot, 262](#)
- [fl_set_status, 262](#)
- [fl_shortcut_label, 263](#)
- [fl_transform_dx, 263](#)
- [fl_transform_dy, 263](#)
- [fl_transform_x, 263](#)
- [fl_transform_y, 263](#)
- [fl_transformed_vertex, 263](#)
- [fl_translate, 264](#)
- [fl_vertex, 264](#)
- [Fl_End, 389](#)
- [Fl_Event](#)
 - [Enumerations.H, 735](#)
- [fl_events](#)
 - [add_handler, 224](#)
 - [belowmouse, 225](#)
 - [compose, 225](#)
 - [compose_reset, 225](#)
 - [event, 225](#)
 - [event_alt, 226](#)
 - [event_button, 226](#)
 - [event_button1, 226](#)
 - [event_button2, 226](#)
 - [event_button3, 226](#)
 - [event_buttons, 226](#)
 - [event_clicks, 227](#)
 - [event_ctrl, 227](#)
 - [event_dx, 227](#)
 - [event_dy, 227](#)
 - [event_inside, 227](#)
 - [event_is_click, 228](#)
 - [event_key, 228](#)
 - [event_length, 229](#)
 - [event_original_key, 229](#)
 - [event_shift, 229](#)
 - [event_state, 229](#)
 - [event_text, 229](#)
 - [event_x_root, 230](#)
 - [event_y_root, 230](#)
 - [focus, 230](#)
 - [get_key, 230](#)
 - [get_mouse, 230](#)
 - [handle, 231](#)
 - [pushed, 231](#)
 - [test_shortcut, 231](#)
- [fl_expand_text](#)
 - [fl_drawings, 257](#)
- [Fl_File_Browser, 390](#)
 - [filetype, 391](#)
 - [filter, 391](#)
 - [Fl_File_Browser, 391](#)
 - [Fl_File_Browser, 391](#)
 - [iconsize, 391](#)
 - [load, 392](#)
- [Fl_File_Chooser, 393](#)
 - [~Fl_File_Chooser, 398](#)
 - [add_extra, 398](#)
 - [color, 398, 399](#)
 - [count, 399](#)
 - [directory, 399](#)
 - [filter, 399](#)
 - [filter_value, 399](#)
 - [Fl_File_Chooser, 398](#)
 - [Fl_File_Chooser, 398](#)
 - [hide, 399](#)
 - [iconsize, 399](#)
 - [label, 400](#)
 - [preview, 400](#)
 - [rescan, 400](#)
 - [show, 400](#)
 - [textcolor, 400](#)
 - [textfont, 400](#)
 - [textsize, 400, 401](#)
 - [type, 401](#)
 - [value, 401](#)
 - [visible, 401](#)
- [fl_file_chooser](#)
 - [group_comdlg, 275](#)
- [fl_file_chooser_callback](#)
 - [group_comdlg, 276](#)
- [fl_file_chooser_ok_label](#)
 - [group_comdlg, 276, 277](#)
- [Fl_File_Icon, 402](#)
 - [add, 404](#)
 - [add_color, 404](#)
 - [add_vertex, 404](#)

- clear, [404](#)
- draw, [404](#)
- find, [405](#)
- first, [405](#)
- Fl_File_Icon, [403](#)
- Fl_File_Icon, [403](#)
- label, [405](#)
- labeltype, [405](#)
- load, [405](#)
- load_fti, [406](#)
- load_image, [406](#)
- load_system_icons, [406](#)
- next, [406](#)
- pattern, [406](#)
- size, [406](#)
- type, [407](#)
- value, [407](#)
- Fl_File_Input, [408](#)
 - down_box, [409](#)
 - errorcolor, [409](#)
 - Fl_File_Input, [409](#)
 - Fl_File_Input, [409](#)
 - handle, [410](#)
 - value, [410](#)
- Fl_File_Sort_F
 - filenames, [279](#)
- fl_filename_absolute
 - filenames, [280](#)
- fl_filename_expand
 - filenames, [280](#)
- fl_filename_ext
 - filenames, [280](#)
- fl_filename_isdir
 - filenames, [280](#)
- fl_filename_match
 - filenames, [280](#)
- fl_filename_name
 - filenames, [281](#)
- fl_filename_relative
 - filenames, [281](#)
- fl_filename_setext
 - filenames, [281](#)
- Fl_Fill_Dial, [411](#)
 - Fl_Fill_Dial, [411](#)
 - Fl_Fill_Dial, [411](#)
- Fl_Fill_Slider, [412](#)
 - Fl_Fill_Slider, [412](#)
 - Fl_Fill_Slider, [412](#)
- Fl_Float_Input, [413](#)
 - Fl_Float_Input, [413](#)
 - Fl_Float_Input, [413](#)
- Fl_Font
 - Enumerations.H, [733](#)
- fl_font
 - fl_attributes, [240](#)
- Fl_Font_Descriptor, [414](#)
- Fl_Fontsize
 - Enumerations.H, [733](#)
- Fl_FormsBitmap, [415](#)
 - bitmap, [415](#)
 - draw, [415](#)
 - set, [416](#)
- Fl_FormsPixmap, [417](#)
 - draw, [418](#)
 - Fl_FormsPixmap, [417](#)
 - Fl_FormsPixmap, [417](#)
 - Pixmap, [418](#)
 - set, [418](#)
- fl_frame
 - Enumerations.H, [738](#)
 - fl_drawings, [258](#)
- fl_frame2
 - fl_drawings, [258](#)
- Fl_Free, [419](#)
 - draw, [420](#)
 - Fl_Free, [420](#)
 - Fl_Free, [420](#)
 - handle, [420](#)
- fl_gap
 - fl_drawings, [258](#)
- Fl_GIF_Image, [422](#)
 - Fl_GIF_Image, [422](#)
 - Fl_GIF_Image, [422](#)
- Fl_Gl_Window, [423](#)
 - can_do, [426](#)
 - can_do_overlay, [426](#)
 - context, [426](#)
 - context_valid, [426](#)
 - draw, [426](#)
 - Fl_Gl_Window, [425](#)
 - Fl_Gl_Window, [425](#)
 - flush, [427](#)
 - hide_overlay, [427](#)
 - make_current, [427](#)
 - make_overlay_current, [427](#)
 - mode, [427](#)
 - ortho, [428](#)
 - redraw_overlay, [428](#)
 - resize, [428](#)
 - show, [428](#)
 - swap_buffers, [429](#)
 - valid, [429](#)
- Fl_Glut_Bitmap_Font, [430](#)
- Fl_Glut_Window, [431](#)
 - draw, [432](#)
 - draw_overlay, [432](#)
 - Fl_Glut_Window, [432](#)
 - Fl_Glut_Window, [432](#)

- handle, 433
- make_current, 433
- Fl_Group, 434
 - ~Fl_Group, 437
 - array, 437
 - begin, 437
 - child, 437
 - clear, 438
 - clip_children, 438
 - current, 438
 - draw, 438
 - draw_child, 439
 - draw_children, 439
 - draw_outside_label, 439
 - end, 439
 - find, 439
 - Fl_Group, 437
 - Fl_Group, 437
 - focus, 439
 - handle, 439
 - init_sizes, 440
 - insert, 440
 - remove, 440, 441
 - resizable, 441
 - resize, 442
 - sizes, 442
 - update_child, 443
- fl_height
 - fl_attributes, 241
- Fl_Help_Dialog, 444
 - Fl_Help_Dialog, 445
 - Fl_Help_Dialog, 445
 - h, 445
 - hide, 445
 - load, 446
 - position, 446
 - resize, 446
 - show, 446
 - textsize, 446
 - value, 446
 - visible, 446
 - w, 447
 - x, 447
 - y, 447
- Fl_Help_Font_Style, 448
- Fl_Help_Link, 449
- Fl_Help_Target, 450
- Fl_Help_View, 451
 - ~Fl_Help_View, 453
 - clear_selection, 453
 - directory, 453
 - filename, 453
 - leftline, 453
 - link, 453
 - load, 454
 - resize, 454
 - select_all, 454
 - size, 454
 - textcolor, 454
 - textfont, 454
 - textsize, 454, 455
 - title, 455
 - topline, 455
 - value, 455
- Fl_Hold_Browser, 456
 - Fl_Hold_Browser, 456
 - Fl_Hold_Browser, 456
- Fl_Image, 457
 - color_average, 459
 - copy, 459
 - count, 459
 - d, 459, 460
 - data, 460
 - desaturate, 460
 - draw, 460
 - draw_empty, 460
 - Fl_Image, 459
 - Fl_Image, 459
 - h, 460
 - inactive, 461
 - label, 461
 - ld, 461
 - uncache, 461
 - w, 461
- Fl_Input, 462
 - draw, 464
 - Fl_Input, 464
 - Fl_Input, 464
 - handle, 464
- fl_input
 - group_comdlg, 277
- Fl_Input_, 466
 - copy, 469
 - copy_cuts, 469
 - cursor_color, 470
 - cut, 470
 - drawtext, 470
 - Fl_Input_, 469
 - Fl_Input_, 469
 - index, 470
 - input_type, 470
 - insert, 470
 - mark, 470, 471
 - maximum_size, 471
 - position, 471
 - readonly, 471
 - replace, 471
 - resize, 472

- shortcut, [472](#)
- size, [472](#)
- static_value, [473](#)
- textcolor, [473](#)
- textfont, [473](#)
- textsize, [473](#)
- undo, [473](#)
- up_down_position, [473](#)
- value, [474](#)
- wrap, [474](#)
- Fl_Input_Choice, [475](#)
 - add, [477](#)
 - changed, [477](#)
 - clear, [477](#)
 - clear_changed, [477](#)
 - Fl_Input_Choice, [477](#)
 - Fl_Input_Choice, [477](#)
 - input, [477](#)
 - menu, [477](#), [478](#)
 - menubutton, [478](#)
 - resize, [478](#)
 - set_changed, [478](#)
 - value, [478](#)
- Fl_Int_Input, [479](#)
 - Fl_Int_Input, [479](#)
 - Fl_Int_Input, [479](#)
- fl_internal_boxttype
 - fl_boxttype.cxx, [744](#)
- Fl_JPEG_Image, [480](#)
 - Fl_JPEG_Image, [480](#)
 - Fl_JPEG_Image, [480](#)
- Fl_Label, [481](#)
 - draw, [482](#)
 - measure, [482](#)
 - type, [482](#)
- Fl_Labeltype
 - Enumerations.H, [737](#)
- fl_latin1_to_local
 - fl_attributes, [241](#)
- Fl_Light_Button, [483](#)
 - draw, [484](#)
 - Fl_Light_Button, [484](#)
 - Fl_Light_Button, [484](#)
 - handle, [484](#)
- fl_line_style
 - fl_drawings, [258](#)
- fl_line_style.cxx, [757](#)
- fl_local_to_latin1
 - fl_attributes, [241](#)
- fl_local_to_mac_roman
 - fl_attributes, [241](#)
- fl_mac_roman_to_local
 - fl_attributes, [242](#)
- FL_MAJOR_VERSION
 - Enumerations.H, [732](#)
- fl_measure
 - fl_drawings, [259](#)
- fl_measure_pixmap
 - fl_drawings, [259](#)
- Fl_Menu_, [486](#)
 - add, [489](#)
 - clear, [490](#)
 - copy, [490](#)
 - down_box, [490](#)
 - find_item, [490](#)
 - Fl_Menu_, [489](#)
 - Fl_Menu_, [489](#)
 - global, [490](#)
 - menu, [490](#)
 - mode, [490](#), [491](#)
 - mvalue, [491](#)
 - picked, [491](#)
 - remove, [491](#)
 - replace, [491](#)
 - shortcut, [491](#)
 - size, [491](#), [492](#)
 - test_shortcut, [492](#)
 - text, [492](#)
 - textcolor, [492](#)
 - textfont, [492](#)
 - textsize, [492](#)
 - value, [493](#)
- Fl_Menu_Bar, [494](#)
 - draw, [495](#)
 - Fl_Menu_Bar, [495](#)
 - Fl_Menu_Bar, [495](#)
 - handle, [495](#)
- Fl_Menu_Button, [497](#)
 - draw, [498](#)
 - Fl_Menu_Button, [498](#)
 - Fl_Menu_Button, [498](#)
 - handle, [499](#)
 - popup, [499](#)
- Fl_Menu_Item, [500](#)
 - activate, [505](#)
 - active, [505](#)
 - activevisible, [506](#)
 - add, [506](#)
 - argument, [506](#)
 - callback, [506](#)
 - check, [506](#)
 - checkbox, [506](#)
 - checked, [506](#)
 - clear, [507](#)
 - deactivate, [507](#)
 - do_callback, [507](#)
 - draw, [507](#)
 - find_shortcut, [507](#)

- first, [507](#), [508](#)
- hide, [508](#)
- label, [508](#)
- labelcolor, [508](#)
- labelfont, [508](#)
- labelsize, [508](#)
- labeltype, [508](#), [509](#)
- measure, [509](#)
- next, [509](#)
- popup, [509](#)
- pulldown, [509](#)
- radio, [510](#)
- set, [510](#)
- setonly, [510](#)
- shortcut, [510](#)
- show, [510](#)
- submenu, [510](#)
- test_shortcut, [511](#)
- uncheck, [511](#)
- value, [511](#)
- visible, [511](#)
- Fl_Menu_Window, [512](#)
- ~Fl_Menu_Window, [513](#)
- clear_overlay, [513](#)
- Fl_Menu_Window, [513](#)
- Fl_Menu_Window, [513](#)
- flush, [513](#)
- hide, [513](#)
- set_overlay, [513](#)
- show, [513](#)
- fl_message
 - group_comdlg, [277](#)
- FL_MINOR_VERSION
 - Enumerations.H, [732](#)
- fl_mult_matrix
 - fl_drawings, [259](#)
- Fl_Multi_Browser, [515](#)
 - Fl_Multi_Browser, [515](#)
 - Fl_Multi_Browser, [515](#)
- Fl_Multiline_Input, [516](#)
 - Fl_Multiline_Input, [516](#)
 - Fl_Multiline_Input, [516](#)
- Fl_Multiline_Output, [517](#)
 - Fl_Multiline_Output, [517](#)
 - Fl_Multiline_Output, [517](#)
- fl_multithread
 - awake, [265](#)
 - lock, [265](#)
 - thread_message, [266](#)
 - unlock, [266](#)
- fl_not_clipped
 - fl_drawings, [260](#)
- Fl_Output, [518](#)
 - Fl_Output, [519](#)
 - Fl_Output, [519](#)
- Fl_Overlay_Window, [520](#)
 - Fl_Overlay_Window, [521](#)
 - Fl_Overlay_Window, [521](#)
 - hide, [521](#)
 - redraw_overlay, [521](#)
 - resize, [521](#)
 - show, [522](#)
- Fl_Pack, [523](#)
 - draw, [524](#)
 - Fl_Pack, [524](#)
 - Fl_Pack, [524](#)
- fl_password
 - group_comdlg, [277](#)
- FL_PATCH_VERSION
 - Enumerations.H, [732](#)
- fl_pie
 - fl_drawings, [260](#)
- Fl_Pixmap, [525](#)
 - color_average, [527](#)
 - copy, [527](#)
 - desaturate, [527](#)
 - draw, [527](#)
 - Fl_Pixmap, [526](#)
 - Fl_Pixmap, [526](#)
 - label, [527](#), [528](#)
 - uncache, [528](#)
- Fl_PNG_Image, [529](#)
 - Fl_PNG_Image, [529](#)
 - Fl_PNG_Image, [529](#)
- Fl_PNM_Image, [530](#)
 - Fl_PNM_Image, [530](#)
 - Fl_PNM_Image, [530](#)
- fl_polygon
 - fl_drawings, [260](#)
- fl_pop_clip
 - fl_drawings, [260](#)
- Fl_Positioner, [531](#)
 - draw, [533](#)
 - Fl_Positioner, [533](#)
 - Fl_Positioner, [533](#)
 - handle, [533](#)
 - value, [533](#)
 - xbounds, [534](#)
 - xstep, [534](#)
 - xvalue, [534](#)
 - ybounds, [534](#)
 - ystep, [534](#)
 - yvalue, [534](#)
- Fl_Preferences, [535](#)
 - ~Fl_Preferences, [539](#)
 - deleteEntry, [539](#)
 - deleteGroup, [539](#)
 - entries, [539](#)

- entry, 539
- entryExists, 540
- Fl_Preferences, 538
- Fl_Preferences, 538
- flush, 540
- get, 540–542
- getUserdataPath, 542
- group, 543
- groupExists, 543
- groups, 543
- Root, 537
- set, 544, 545
- size, 546
- Fl_Preferences::Entry, 547
- Fl_Preferences::Name, 548
 - Name, 548
- Fl_Progress, 550
 - draw, 551
 - Fl_Progress, 551
 - Fl_Progress, 551
 - maximum, 551
 - minimum, 551
 - value, 551
- fl_push_clip
 - fl_drawings, 261
- fl_push_matrix
 - fl_drawings, 261
- fl_read_image
 - fl_drawings, 261
- fl_rect.cxx, 758
 - fl_clip_region, 760
- fl_rectbound
 - fl_boxtype.cxx, 744
- fl_rectf
 - fl_drawings, 261
- Fl_Repeat_Button, 552
 - deactivate, 552
 - Fl_Repeat_Button, 552
 - Fl_Repeat_Button, 552
 - handle, 553
- fl_reset_spot
 - fl_drawings, 261
- Fl_Return_Button, 554
 - draw, 555
 - Fl_Return_Button, 555
 - Fl_Return_Button, 555
 - handle, 555
- Fl_RGB_Image, 557
 - ~Fl_RGB_Image, 558
 - color_average, 558
 - copy, 558
 - desaturate, 559
 - draw, 559
 - Fl_RGB_Image, 558
 - Fl_RGB_Image, 558
 - label, 559
 - uncache, 559
- Fl_Roller, 561
 - draw, 562
 - Fl_Roller, 562
 - Fl_Roller, 562
 - handle, 562
- fl_rotate
 - fl_drawings, 262
- Fl_Round_Button, 564
- Fl_Round_Clock, 566
 - Fl_Round_Clock, 566
 - Fl_Round_Clock, 566
- fl_scale
 - fl_drawings, 262
- fl_screen
 - h, 235
 - screen_xywh, 235, 236
 - w, 236
 - x, 236
 - y, 236
- Fl_Scroll, 567
 - bbox, 569
 - clear, 569
 - draw, 569
 - Fl_Scroll, 569
 - Fl_Scroll, 569
 - handle, 570
 - resize, 570
 - scroll_to, 570
 - xposition, 571
 - yposition, 571
- fl_scroll
 - fl_drawings, 262
- Fl_Scrollbar, 572
 - ~Fl_Scrollbar, 573
 - draw, 573
 - Fl_Scrollbar, 573
 - Fl_Scrollbar, 573
 - handle, 574
 - linesize, 574
 - value, 574, 575
- Fl_Secret_Input, 576
 - Fl_Secret_Input, 576
 - Fl_Secret_Input, 576
- Fl_Select_Browser, 577
 - Fl_Select_Browser, 577
 - Fl_Select_Browser, 577
- fl_set_spot
 - fl_drawings, 262
- fl_set_status
 - fl_drawings, 262
- Fl_Shared_Image, 578

- ~Fl_Shared_Image, 580
- color_average, 580
- copy, 580, 581
- desaturate, 581
- draw, 581
- Fl_Shared_Image, 580
- Fl_Shared_Image, 580
- get, 581
- num_images, 581
- refcount, 582
- release, 582
- uncache, 582
- fl_shortcut_label
 - fl_drawings, 263
- Fl_Simple_Counter, 583
- Fl_Single_Window, 584
 - flush, 585
 - make_current, 585
 - show, 585
- fl_size
 - fl_attributes, 242
- Fl_Slider, 586
 - bounds, 588
 - draw, 588
 - Fl_Slider, 587
 - Fl_Slider, 587
 - handle, 588
 - scrollvalue, 588
 - slider, 589
 - slider_size, 589
- Fl_Spinner, 590
 - Fl_Spinner, 592
 - Fl_Spinner, 592
 - format, 592
 - handle, 592
 - maximum, 592
 - maximum, 592
 - minimum, 593
 - minimum, 593
 - range, 593
 - resize, 593
 - step, 593
 - textcolor, 593, 594
 - textfont, 594
 - textsize, 594
 - type, 594
 - value, 594
- Fl_String
 - fl_types.h, 761
- Fl_Tabs, 595
 - draw, 597
 - Fl_Tabs, 596
 - Fl_Tabs, 596
 - handle, 597
 - value, 597
- Fl_Text_Buffer, 599
 - add_modify_callback, 607
 - add_predelete_callback, 607
 - append, 607
 - appendfile, 607
 - character, 607
 - character_width, 607
 - clear_rectangular, 608
 - count_displayed_characters, 608
 - count_lines, 608
 - expand_character, 608
 - findchar_backward, 608
 - findchar_forward, 609
 - findchars_backward, 609
 - findchars_forward, 609
 - Fl_Text_Buffer, 607
 - Fl_Text_Buffer, 607
 - highlight, 609
 - highlight_position, 609
 - highlight_selection, 609
 - highlight_text, 610
 - insert, 610
 - insert_, 610
 - insert_column, 610
 - insert_column_, 610
 - insertfile, 610
 - length, 610
 - line_start, 610
 - line_text, 611
 - mNodifyProcs, 615
 - mNullSubsChar, 615
 - mPredeleteProcs, 615
 - mTabDist, 615
 - null_substitution_character, 611
 - outputfile, 611
 - overlay_rectangular, 611
 - overlay_rectangular_, 611
 - primary_selection, 611
 - rectangular_selection_boundaries, 611
 - remove, 612
 - remove_, 612
 - remove_modify_callback, 612
 - remove_predelete_callback, 612
 - remove_rectangular_, 612
 - remove_secondary_selection, 612
 - remove_selection, 612
 - remove_selection_, 612
 - replace_rectangular, 612
 - replace_secondary_selection, 613
 - replace_selection, 613
 - replace_selection_, 613
 - rewind_lines, 613
 - secondary_select, 613

- secondary_selection, 613
- secondary_selection_position, 613
- secondary_selection_text, 613
- secondary_unselect, 614
- select, 614
- selection_text, 614
- substitute_null_characters, 614
- tab_distance, 614
- text, 614
- text_in_rectangle, 614
- text_range, 614
- unhighlight, 615
- unsubstitute_null_characters, 615
- word_end, 615
- Fl_Text_Display, 616
 - ~Fl_Text_Display, 623
 - buffer, 623
 - calc_line_starts, 623
 - count_lines, 623
 - cursor_color, 623, 624
 - cursor_style, 624
 - display_insert, 624
 - draw, 624
 - draw_line_numbers, 624
 - draw_range, 624
 - draw_string, 625
 - draw_text, 625
 - find_line_end, 625
 - find_wrap_range, 625
 - Fl_Text_Display, 623
 - Fl_Text_Display, 623
 - get_absolute_top_line_number, 625
 - handle, 625
 - highlight_data, 626
 - insert, 626
 - line_end, 626
 - maintain_absolute_top_line_number, 627
 - measure_deleted_lines, 627
 - measure_proportional_character, 627
 - move_down, 627
 - move_left, 627
 - move_right, 627
 - move_up, 628
 - next_word, 628
 - offset_line_starts, 628
 - overstrike, 628
 - position_style, 628
 - position_to_line, 628
 - position_to_linecol, 628
 - position_to_xy, 629
 - previous_word, 629
 - redisplay_range, 629
 - reset_absolute_top_line_number, 629
 - scroll, 629
 - scrollbar_width, 629
 - shortcut, 629
 - show_insert_position, 630
 - skip_lines, 630
 - textcolor, 630
 - textfont, 630
 - textsize, 630
 - word_end, 630
 - word_start, 630
 - wrap_mode, 630
 - wrap_uses_character, 630
 - wrapped_column, 631
 - wrapped_line_counter, 631
 - wrapped_row, 631
 - xy_to_position, 631
 - xy_to_rowcol, 632
- Fl_Text_Display::Style_Table_Entry, 633
- Fl_Text_Editor, 634
 - add_default_key_bindings, 637
 - bound_key_function, 637
 - default_key_function, 637
 - Fl_Text_Editor, 637
 - Fl_Text_Editor, 637
 - handle, 637
 - insert_mode, 638
 - kf_backspace, 638
 - kf_c_s_move, 638
 - kf_copy, 638
 - kf_cut, 638
 - kf_delete, 638
 - kf_down, 638
 - kf_end, 639
 - kf_home, 639
 - kf_insert, 639
 - kf_left, 639
 - kf_move, 639
 - kf_page_down, 639
 - kf_page_up, 639
 - kf_paste, 639
 - kf_right, 639
 - kf_select_all, 639
 - kf_shift_move, 639
 - kf_undo, 640
 - kf_up, 640
 - remove_all_key_bindings, 640
 - remove_key_binding, 640
- Fl_Text_Editor::Key_Binding, 641
- fl_text_extents
 - fl_attributes, 242
- Fl_Text_Selection, 642
- Fl_Tile, 644
 - Fl_Tile, 645
 - Fl_Tile, 645
 - handle, 645

- position, 646
- resize, 646
- Fl_Tiled_Image, 647
 - color_average, 648
 - copy, 648
 - desaturate, 648
 - draw, 648, 649
 - Fl_Tiled_Image, 648
 - Fl_Tiled_Image, 648
- Fl_Timer, 650
 - direction, 651
 - draw, 651
 - Fl_Timer, 651
 - Fl_Timer, 651
 - handle, 651
 - suspended, 652
- Fl_Toggle_Button, 653
 - Fl_Toggle_Button, 653
 - Fl_Toggle_Button, 653
- Fl_Tooltip, 654
 - color, 655
 - current, 655
 - delay, 655, 656
 - disable, 656
 - enable, 656
 - enabled, 656
 - enter_area, 656
 - font, 656
 - hoverdelay, 656
 - size, 657
 - textcolor, 657
- fl_transform_dx
 - fl_drawings, 263
- fl_transform_dy
 - fl_drawings, 263
- fl_transform_x
 - fl_drawings, 263
- fl_transform_y
 - fl_drawings, 263
- fl_transformed_vertex
 - fl_drawings, 263
- fl_translate
 - fl_drawings, 264
- fl_types.h, 761
 - Fl_CString, 761
 - Fl_String, 761
- Fl_Valuator, 658
 - bounds, 660
 - clamp, 660
 - Fl_Valuator, 660
 - Fl_Valuator, 660
 - format, 661
 - handle_drag, 661
 - handle_release, 661
 - increment, 661
 - maximum, 661
 - minimum, 661
 - precision, 661
 - range, 662
 - round, 662
 - set_value, 662
 - step, 662
 - value, 662
- Fl_Value_Input, 664
 - cursor_color, 666
 - draw, 666
 - Fl_Value_Input, 666
 - Fl_Value_Input, 666
 - handle, 667
 - resize, 667
 - shortcut, 668
 - soft, 668
 - textcolor, 668
 - textfont, 668
 - textsize, 668
- Fl_Value_Output, 670
 - draw, 672
 - Fl_Value_Output, 671
 - Fl_Value_Output, 671
 - handle, 672
 - soft, 672
 - textcolor, 673
 - textfont, 673
 - textsize, 673
- Fl_Value_Slider, 674
 - draw, 675
 - Fl_Value_Slider, 675
 - Fl_Value_Slider, 675
 - handle, 675
 - textcolor, 676
 - textfont, 676
 - textsize, 676
- FL_VERSION
 - Enumerations.H, 732
- fl_vertex
 - fl_drawings, 264
- fl_vertex.cxx, 762
- Fl_When
 - Enumerations.H, 738
- Fl_Widget, 677
 - ~Fl_Widget, 685
 - activate, 686
 - active, 686
 - active_r, 686
 - align, 686, 687
 - argument, 687
 - box, 687
 - callback, 688

- changed, 689
- clear_changed, 689
- clear_damage, 689
- clear_output, 689
- clear_visible, 690
- clear_visible_focus, 690
- color, 690
- color2, 691
- contains, 691
- copy_label, 691
- damage, 692
- damage_resize, 692
- deactivate, 692
- default_callback, 693
- deimage, 693
- do_callback, 694
- draw, 694
- draw_box, 695
- draw_label, 695
- Fl_Widget, 685
- Fl_Widget, 685
- h, 695, 696
- handle, 696
- hide, 696
- image, 696, 697
- inside, 697
- label, 697, 698
- label_shortcut, 698
- labelcolor, 698
- labelfont, 698, 699
- labelsize, 699
- labeltype, 699, 700
- measure_label, 700
- output, 700
- parent, 700, 701
- position, 701
- redraw, 701
- redraw_label, 701
- resize, 701
- selection_color, 702
- set_changed, 702
- set_output, 703
- set_visible, 703
- set_visible_focus, 703
- show, 703
- size, 703
- take_focus, 704
- takesevents, 704
- test_shortcut, 704
- tooltip, 704, 705
- type, 705
- user_data, 705
- visible, 706
- visible_focus, 706
- visible_r, 706
- w, 706, 707
- when, 707
- window, 708
- x, 708
- y, 708
- Fl_Widget_Tracker, 710
 - deleted, 711
 - exists, 711
 - widget, 711
- fl_width
 - fl_attributes, 242
- Fl_Window, 712
 - ~Fl_Window, 716
 - border, 716
 - clear_border, 716
 - copy_label, 716
 - current, 717
 - current_, 722
 - cursor, 717
 - default_cursor, 717
 - draw, 717
 - Fl_Window, 715, 716
 - Fl_Window, 715, 716
 - flush, 718
 - fullscreen, 718
 - handle, 718
 - hide, 718
 - hotspot, 718
 - iconize, 719
 - iconlabel, 719
 - label, 719
 - make_current, 719
 - modal, 719
 - non_modal, 720
 - override, 720
 - resize, 720
 - set_modal, 720
 - set_non_modal, 720
 - show, 720
 - shown, 721
 - size_range, 721
 - xclass, 721
- fl_windows
 - atclose, 221
 - default_atclose, 220
 - first_window, 220
 - grab, 220
 - modal, 220
 - next_window, 221
 - set_atclose, 221
- Fl_Wizard, 723
 - Fl_Wizard, 723
 - Fl_Wizard, 723

- next, [724](#)
- prev, [724](#)
- value, [724](#)
- Fl_XBM_Image, [725](#)
 - Fl_XBM_Image, [725](#)
 - Fl_XBM_Image, [725](#)
- fl_xpixel
 - fl_attributes, [243](#)
- Fl_XPM_Image, [726](#)
 - Fl_XPM_Image, [726](#)
 - Fl_XPM_Image, [726](#)
- flush
 - Fl, [298](#)
 - Fl_Double_Window, [387](#)
 - Fl_Gl_Window, [427](#)
 - Fl_Menu_Window, [513](#)
 - Fl_Preferences, [540](#)
 - Fl_Single_Window, [585](#)
 - Fl_Window, [718](#)
- focus
 - fl_events, [230](#)
 - Fl_Group, [439](#)
- font
 - Fl_Tooltip, [656](#)
- foreground
 - Fl, [298](#)
- format
 - Fl_Spinner, [592](#)
 - Fl_Valuator, [661](#)
- format_char
 - Fl_Browser, [321](#)
- free_color
 - fl_attributes, [243](#)
- full_height
 - Fl_Browser, [322](#)
 - Fl_Browser_, [337](#)
- full_width
 - Fl_Browser_, [337](#)
- fullscreen
 - Fl_Window, [718](#)
- g
 - Fl_Color_Chooser, [376](#)
- get
 - Fl_Preferences, [540–542](#)
 - Fl_Shared_Image, [581](#)
- get_absolute_top_line_number
 - Fl_Text_Display, [625](#)
- get_awake_handler_
 - Fl, [298](#)
- get_boxtype
 - Fl, [298](#)
- get_color
 - fl_attributes, [243](#)
- get_font
 - fl_attributes, [244](#)
- get_font_name
 - fl_attributes, [244](#)
- get_font_sizes
 - fl_attributes, [244](#)
- get_key
 - fl_events, [230](#)
- get_mouse
 - fl_events, [230](#)
- get_system_colors
 - Fl, [298](#)
- getUserdataPath
 - Fl_Preferences, [542](#)
- gl_visual
 - Fl, [299](#)
- global
 - Fl_Menu_, [490](#)
- grab
 - fl_windows, [220](#)
- group
 - Fl_Preferences, [543](#)
- group_cairo
 - cairo_autolink_context, [270](#)
 - cairo_cc, [271](#)
- group_comdlg
 - error, [277](#)
 - fatal, [278](#)
 - fl_alert, [273](#)
 - fl_ask, [273](#)
 - fl_choice, [273](#)
 - fl_color_chooser, [274](#)
 - fl_dir_chooser, [275](#)
 - fl_file_chooser, [275](#)
 - fl_file_chooser_callback, [276](#)
 - fl_file_chooser_ok_label, [276, 277](#)
 - fl_input, [277](#)
 - fl_message, [277](#)
 - fl_password, [277](#)
 - warning, [278](#)
- groupExists
 - Fl_Preferences, [543](#)
- groups
 - Fl_Preferences, [543](#)
- h
 - Fl_Help_Dialog, [445](#)
 - Fl_Image, [460](#)
 - fl_screen, [235](#)
 - Fl_Widget, [695, 696](#)
- handle
 - Fl_Adjuster, [306](#)
 - Fl_Box, [313](#)
 - Fl_Browser_, [337](#)

- Fl_Button, 345
- Fl_Check_Browser, 359
- Fl_Choice, 365
- Fl_Clock, 369
- Fl_Counter, 381
- Fl_Dial, 385
- fl_events, 231
- Fl_File_Input, 410
- Fl_Free, 420
- Fl_Glut_Window, 433
- Fl_Group, 439
- Fl_Input, 464
- Fl_Light_Button, 484
- Fl_Menu_Bar, 495
- Fl_Menu_Button, 499
- Fl_Positioner, 533
- Fl_Repeat_Button, 553
- Fl_Return_Button, 555
- Fl_Roller, 562
- Fl_Scroll, 570
- Fl_Scrollbar, 574
- Fl_Slider, 588
- Fl_Spinner, 592
- Fl_Tabs, 597
- Fl_Text_Display, 625
- Fl_Text_Editor, 637
- Fl_Tile, 645
- Fl_Timer, 651
- Fl_Value_Input, 667
- Fl_Value_Output, 672
- Fl_Value_Slider, 675
- Fl_Widget, 696
- Fl_Window, 718
- handle_drag
 - Fl_Valuator, 661
- handle_release
 - Fl_Valuator, 661
- has_scrollbar
 - Fl_Browser_, 337
- help
 - Fl, 303
- hide
 - Fl_Browser, 322
 - Fl_Double_Window, 387
 - Fl_File_Chooser, 399
 - Fl_Help_Dialog, 445
 - Fl_Menu_Item, 508
 - Fl_Menu_Window, 513
 - Fl_Overlay_Window, 521
 - Fl_Widget, 696
 - Fl_Window, 718
- hide_overlay
 - Fl_Gl_Window, 427
- highlight
 - Fl_Text_Buffer, 609
- highlight_data
 - Fl_Text_Display, 626
- highlight_position
 - Fl_Text_Buffer, 609
- highlight_selection
 - Fl_Text_Buffer, 609
- highlight_text
 - Fl_Text_Buffer, 610
- hotspot
 - Fl_Window, 718
- hour
 - Fl_Clock_Output, 372
- hoverdelay
 - Fl_Tooltip, 656
- hsv
 - Fl_Color_Chooser, 376
- hsv2rgb
 - Fl_Color_Chooser, 376
- hue
 - Fl_Color_Chooser, 377
- iconize
 - Fl_Window, 719
- iconlabel
 - Fl_Window, 719
- iconsize
 - Fl_File_Browser, 391
 - Fl_File_Chooser, 399
- idle
 - Fl, 303
- image
 - Fl_Widget, 696, 697
- INACTIVE
 - Fl_Widget, 685
- inactive
 - Fl_Image, 461
- incr_height
 - Fl_Browser, 322
 - Fl_Browser_, 338
- increment
 - Fl_Valuator, 661
- index
 - Fl_Input_, 470
- init_sizes
 - Fl_Group, 440
- input
 - Fl_Input_Choice, 477
- input_type
 - Fl_Input_, 470
- insert
 - Fl_Browser, 323
 - Fl_Chart, 353
 - Fl_Group, 440

- FL_Input_, 470
- FL_Text_Buffer, 610
- FL_Text_Display, 626
- insert_
 - FL_Text_Buffer, 610
- insert_column
 - FL_Text_Buffer, 610
- insert_column_
 - FL_Text_Buffer, 610
- insert_mode
 - FL_Text_Editor, 638
- insertfile
 - FL_Text_Buffer, 610
- inserting
 - FL_Browser_, 338
- inside
 - FL_Widget, 697
- INVISIBLE
 - FL_Widget, 685
- item_at
 - FL_Browser, 323
- item_draw
 - FL_Browser, 323
- item_first
 - FL_Browser, 324
 - FL_Browser_, 338
- item_height
 - FL_Browser, 324
 - FL_Browser_, 338
- item_last
 - FL_Browser, 324
 - FL_Browser_, 338
- item_next
 - FL_Browser, 325
 - FL_Browser_, 338
- item_prev
 - FL_Browser, 325
 - FL_Browser_, 338
- item_quick_height
 - FL_Browser_, 339
- item_select
 - FL_Browser, 325
 - FL_Browser_, 339
- item_selected
 - FL_Browser, 325
 - FL_Browser_, 339
- item_swap
 - FL_Browser, 326
- item_text
 - FL_Browser, 326
- item_width
 - FL_Browser, 326
 - FL_Browser_, 339
- kf_backspace
 - FL_Text_Editor, 638
- kf_c_s_move
 - FL_Text_Editor, 638
- kf_copy
 - FL_Text_Editor, 638
- kf_cut
 - FL_Text_Editor, 638
- kf_delete
 - FL_Text_Editor, 638
- kf_down
 - FL_Text_Editor, 638
- kf_end
 - FL_Text_Editor, 639
- kf_home
 - FL_Text_Editor, 639
- kf_insert
 - FL_Text_Editor, 639
- kf_left
 - FL_Text_Editor, 639
- kf_move
 - FL_Text_Editor, 639
- kf_page_down
 - FL_Text_Editor, 639
- kf_page_up
 - FL_Text_Editor, 639
- kf_paste
 - FL_Text_Editor, 639
- kf_right
 - FL_Text_Editor, 639
- kf_select_all
 - FL_Text_Editor, 639
- kf_shift_move
 - FL_Text_Editor, 639
- kf_undo
 - FL_Text_Editor, 640
- kf_up
 - FL_Text_Editor, 640
- label
 - FL_Bitmap, 310
 - FL_File_Chooser, 400
 - FL_File_Icon, 405
 - FL_Image, 461
 - FL_Menu_Item, 508
 - FL_Pixmap, 527, 528
 - FL_RGB_Image, 559
 - FL_Widget, 697, 698
 - FL_Window, 719
- label_shortcut
 - FL_Widget, 698
- labelcolor
 - FL_Menu_Item, 508
 - FL_Widget, 698

- labelfont
 - Fl_Menu_Item, 508
 - Fl_Widget, 698, 699
- labelsize
 - Fl_Menu_Item, 508
 - Fl_Widget, 699
- labeltype
 - Fl_File_Icon, 405
 - Fl_Menu_Item, 508, 509
 - Fl_Widget, 699, 700
- ld
 - Fl_Image, 461
- leftline
 - Fl_Help_View, 453
- length
 - Fl_Text_Buffer, 610
- line_end
 - Fl_Text_Display, 626
- line_start
 - Fl_Text_Buffer, 610
- line_text
 - Fl_Text_Buffer, 611
- lineno
 - Fl_Browser, 326
- lineposition
 - Fl_Browser, 327
- linesize
 - Fl_Scrollbar, 574
- link
 - Fl_Help_View, 453
- load
 - Fl_Browser, 327
 - Fl_File_Browser, 392
 - Fl_File_Icon, 405
 - Fl_Help_Dialog, 446
 - Fl_Help_View, 454
- load_fti
 - Fl_File_Icon, 406
- load_image
 - Fl_File_Icon, 406
- load_system_icons
 - Fl_File_Icon, 406
- lock
 - fl_multithread, 265
- lstep
 - Fl_Counter, 382
- maintain_absolute_top_line_number
 - Fl_Text_Display, 627
- make_current
 - Fl_Gl_Window, 427
 - Fl_Glut_Window, 433
 - Fl_Single_Window, 585
 - Fl_Window, 719
- make_overlay_current
 - Fl_Gl_Window, 427
- make_visible
 - Fl_Browser, 327
- mark
 - Fl_Input_, 470, 471
- maximum
 - Fl_Progress, 551
 - Fl_Spinner, 592
 - Fl_Valuator, 661
- maximum_size
 - Fl_Input_, 471
- maxinum
 - Fl_Spinner, 592
- maxsize
 - Fl_Chart, 353
- measure
 - Fl_Label, 482
 - Fl_Menu_Item, 509
- measure_deleted_lines
 - Fl_Text_Display, 627
- measure_label
 - Fl_Widget, 700
- measure_proportional_character
 - Fl_Text_Display, 627
- menu
 - Fl_Input_Choice, 477, 478
 - Fl_Menu_, 490
- menubutton
 - Fl_Input_Choice, 478
- middleline
 - Fl_Browser, 327
- minimum
 - Fl_Progress, 551
 - Fl_Spinner, 593
 - Fl_Valuator, 661
- mininum
 - Fl_Spinner, 593
- minute
 - Fl_Clock_Output, 372
- mNodifyProcs
 - Fl_Text_Buffer, 615
- mNullSubsChar
 - Fl_Text_Buffer, 615
- modal
 - Fl_Window, 719
 - fl_windows, 220
- mode
 - Fl_Gl_Window, 427
 - Fl_Menu_, 490, 491
- move
 - Fl_Browser, 327
- move_down
 - Fl_Text_Display, 627

- move_left
 - Fl_Text_Display, 627
- move_right
 - Fl_Text_Display, 627
- move_up
 - Fl_Text_Display, 628
- mPredeleteProcs
 - Fl_Text_Buffer, 615
- mTabDist
 - Fl_Text_Buffer, 615
- Multithreading support functions, 265
- mvalue
 - Fl_Menu_, 491
- Name
 - Fl_Preferences::Name, 548
- nchecked
 - Fl_Check_Browser, 359
- new_list
 - Fl_Browser_, 339
- next
 - Fl_File_Icon, 406
 - Fl_Menu_Item, 509
 - Fl_Wizard, 724
- next_window
 - fl_windows, 221
- next_word
 - Fl_Text_Display, 628
- nitems
 - Fl_Check_Browser, 359
- non_modal
 - Fl_Window, 720
- null_substitution_character
 - Fl_Text_Buffer, 611
- num_images
 - Fl_Shared_Image, 581
- offset_line_starts
 - Fl_Text_Display, 628
- ortho
 - Fl_Gl_Window, 428
- OUTPUT
 - Fl_Widget, 685
- output
 - Fl_Widget, 700
- outputfile
 - Fl_Text_Buffer, 611
- overlay_rectangular
 - Fl_Text_Buffer, 611
- overlay_rectangular_
 - Fl_Text_Buffer, 611
- override
 - Fl_Window, 720
- overstrike
 - Fl_Text_Display, 628
- own_colormap
 - Fl, 299
- parent
 - Fl_Widget, 700, 701
- paste
 - fl_clipboard, 233
- pattern
 - Fl_File_Icon, 406
- picked
 - Fl_Menu_, 491
- Pixmap
 - Fl_FormsPixmap, 418
- popup
 - Fl_Menu_Button, 499
 - Fl_Menu_Item, 509
- position
 - Fl_Help_Dialog, 446
 - Fl_Input_, 471
 - Fl_Tile, 646
 - Fl_Widget, 701
- position_style
 - Fl_Text_Display, 628
- position_to_line
 - Fl_Text_Display, 628
- position_to_linecol
 - Fl_Text_Display, 628
- position_to_xy
 - Fl_Text_Display, 629
- precision
 - Fl_Valuator, 661
- prev
 - Fl_Wizard, 724
- preview
 - Fl_File_Chooser, 400
- previous_word
 - Fl_Text_Display, 629
- primary_selection
 - Fl_Text_Buffer, 611
- pulldown
 - Fl_Menu_Item, 509
- pushed
 - fl_events, 231
- r
 - Fl_Color_Chooser, 377
- radio
 - Fl_Menu_Item, 510
- range
 - Fl_Spinner, 593
 - Fl_Valuator, 662
- readonly
 - Fl_Input_, 471

- ready
 - Fl, [299](#)
- rectangular_selection_boundaries
 - Fl_Text_Buffer, [611](#)
- redisplay_range
 - Fl_Text_Display, [629](#)
- redraw
 - Fl_Widget, [701](#)
- redraw_label
 - Fl_Widget, [701](#)
- redraw_lines
 - Fl_Browser_, [339](#)
- redraw_overlay
 - Fl_Gl_Window, [428](#)
 - Fl_Overlay_Window, [521](#)
- refcount
 - Fl_Shared_Image, [582](#)
- release
 - Fl, [299](#)
 - Fl_Shared_Image, [582](#)
- release_widget_pointer
 - fl_del_widget, [268](#)
- reload_scheme
 - Fl, [299](#)
- remove
 - Fl_Browser, [328](#)
 - Fl_Check_Browser, [359](#)
 - Fl_Group, [440](#), [441](#)
 - Fl_Menu_, [491](#)
 - Fl_Text_Buffer, [612](#)
- remove_
 - Fl_Text_Buffer, [612](#)
- remove_all_key_bindings
 - Fl_Text_Editor, [640](#)
- remove_check
 - Fl, [300](#)
- remove_fd
 - Fl, [300](#)
- remove_key_binding
 - Fl_Text_Editor, [640](#)
- remove_modify_callback
 - Fl_Text_Buffer, [612](#)
- remove_predelete_callback
 - Fl_Text_Buffer, [612](#)
- remove_rectangular_
 - Fl_Text_Buffer, [612](#)
- remove_secondary_selection
 - Fl_Text_Buffer, [612](#)
- remove_selection
 - Fl_Text_Buffer, [612](#)
- remove_selection_
 - Fl_Text_Buffer, [612](#)
- remove_timeout
 - Fl, [300](#)
- repeat_timeout
 - Fl, [300](#)
- replace
 - Fl_Browser, [328](#)
 - Fl_Chart, [354](#)
 - Fl_Input_, [471](#)
 - Fl_Menu_, [491](#)
- replace_rectangular
 - Fl_Text_Buffer, [612](#)
- replace_secondary_selection
 - Fl_Text_Buffer, [613](#)
- replace_selection
 - Fl_Text_Buffer, [613](#)
- replace_selection_
 - Fl_Text_Buffer, [613](#)
- replacing
 - Fl_Browser_, [339](#)
- rescan
 - Fl_File_Chooser, [400](#)
- reset_absolute_top_line_number
 - Fl_Text_Display, [629](#)
- resizable
 - Fl_Group, [441](#)
- resize
 - Fl_Browser_, [339](#)
 - Fl_Double_Window, [388](#)
 - Fl_Gl_Window, [428](#)
 - Fl_Group, [442](#)
 - Fl_Help_Dialog, [446](#)
 - Fl_Help_View, [454](#)
 - Fl_Input_, [472](#)
 - Fl_Input_Choice, [478](#)
 - Fl_Overlay_Window, [521](#)
 - Fl_Scroll, [570](#)
 - Fl_Spinner, [593](#)
 - Fl_Tile, [646](#)
 - Fl_Value_Input, [667](#)
 - Fl_Widget, [701](#)
 - Fl_Window, [720](#)
- rewind_lines
 - Fl_Text_Buffer, [613](#)
- rgb
 - Fl_Color_Chooser, [377](#)
- rgb2hsv
 - Fl_Color_Chooser, [377](#)
- Root
 - Fl_Preferences, [537](#)
- round
 - Fl_Valuator, [662](#)
- run
 - Fl, [300](#)
- Safe widget deletion support functions, [267](#)
- saturation

- FL_Color_Chooser, 377
- scheme
 - FL, 300
- Screen functions, 235
- screen_xywh
 - fl_screen, 235, 236
- scroll
 - FL_Text_Display, 629
- scroll_to
 - FL_Scroll, 570
- scrollbar
 - FL_Browser_, 341
- scrollbar_left
 - FL_Browser_, 340
- scrollbar_right
 - FL_Browser_, 340
- scrollbar_width
 - FL_Text_Display, 629
- scrollvalue
 - FL_Slider, 588
- second
 - FL_Clock_Output, 372
- secondary_select
 - FL_Text_Buffer, 613
- secondary_selection
 - FL_Text_Buffer, 613
- secondary_selection_position
 - FL_Text_Buffer, 613
- secondary_selection_text
 - FL_Text_Buffer, 613
- secondary_unselect
 - FL_Text_Buffer, 614
- select
 - FL_Browser, 328
 - FL_Browser_, 340
 - FL_Text_Buffer, 614
- select_all
 - FL_Help_View, 454
- select_only
 - FL_Browser_, 340
- selected
 - FL_Browser, 328
- selection
 - FL_Browser_, 340
 - fl_clipboard, 233
- Selection & Clipboard functions, 232
- selection_color
 - FL_Widget, 702
- selection_owner
 - fl_clipboard, 233
- selection_text
 - FL_Text_Buffer, 614
- set
 - FL_Button, 345
 - FL_FormsBitmap, 416
 - FL_FormsPixmap, 418
 - FL_Menu_Item, 510
 - FL_Preferences, 544, 545
- set_atclose
 - fl_windows, 221
- set_boxtype
 - FL, 301
- set_changed
 - FL_Input_Choice, 478
 - FL_Widget, 702
- set_checked
 - FL_Check_Browser, 360
- set_color
 - fl_attributes, 244
- set_draw_cb
 - FL_Cairo_Window, 349
- set_font
 - fl_attributes, 245
- set_fonts
 - fl_attributes, 245
- set_idle
 - FL, 301
- set_labeltype
 - FL, 301
- set_modal
 - FL_Window, 720
- set_non_modal
 - FL_Window, 720
- set_output
 - FL_Widget, 703
- set_overlay
 - FL_Menu_Window, 513
- set_value
 - FL_Valuator, 662
- set_visible
 - FL_Widget, 703
- set_visible_focus
 - FL_Widget, 703
- setonly
 - FL_Menu_Item, 510
- shortcut
 - FL_Button, 345, 346
 - FL_Input_, 472
 - FL_Menu_, 491
 - FL_Menu_Item, 510
 - FL_Text_Display, 629
 - FL_Value_Input, 668
- SHORTCUT_LABEL
 - FL_Widget, 685
- show
 - FL_Browser, 328, 329
 - FL_Double_Window, 388
 - FL_File_Chooser, 400

- FL_Gl_Window, 428
- FL_Help_Dialog, 446
- FL_Menu_Item, 510
- FL_Menu_Window, 513
- FL_Overlay_Window, 522
- FL_Single_Window, 585
- FL_Widget, 703
- FL_Window, 720
- show_insert_position
 - FL_Text_Display, 630
- shown
 - FL_Window, 721
- size
 - FL_Browser, 329
 - FL_Chart, 354
 - FL_File_Icon, 406
 - FL_Help_View, 454
 - FL_Input_, 472
 - FL_Menu_, 491, 492
 - FL_Preferences, 546
 - FL_Tooltip, 657
 - FL_Widget, 703
- size_range
 - FL_Window, 721
- sizes
 - FL_Group, 442
- skip_lines
 - FL_Text_Display, 630
- slider
 - FL_Slider, 589
- slider_size
 - FL_Slider, 589
- soft
 - FL_Adjuster, 307
 - FL_Value_Input, 668
 - FL_Value_Output, 672
- sort
 - FL_Browser_, 340
- static_value
 - FL_Input_, 473
- step
 - FL_Counter, 382
 - FL_Spinner, 593
 - FL_Valuator, 662
- str
 - FL_CHART_ENTRY, 356
- submenu
 - FL_Menu_Item, 510
- substitute_null_characters
 - FL_Text_Buffer, 614
- suspended
 - FL_Timer, 652
- swap
 - FL_Browser, 329
- swap_buffers
 - FL_Gl_Window, 429
- SYSTEM
 - FL_Preferences, 538
- tab_distance
 - FL_Text_Buffer, 614
- take_focus
 - FL_Widget, 704
- takesevents
 - FL_Widget, 704
- test_shortcut
 - fl_events, 231
 - FL_Menu_, 492
 - FL_Menu_Item, 511
 - FL_Widget, 704
- text
 - FL_Browser, 330
 - FL_Check_Browser, 360
 - FL_Menu_, 492
 - FL_Text_Buffer, 614
- text_in_rectangle
 - FL_Text_Buffer, 614
- text_range
 - FL_Text_Buffer, 614
- textcolor
 - FL_Chart, 354
 - FL_File_Chooser, 400
 - FL_Help_View, 454
 - FL_Input_, 473
 - FL_Menu_, 492
 - FL_Spinner, 593, 594
 - FL_Text_Display, 630
 - FL_Tooltip, 657
 - FL_Value_Input, 668
 - FL_Value_Output, 673
 - FL_Value_Slider, 676
- textfont
 - FL_Chart, 354
 - FL_File_Chooser, 400
 - FL_Help_View, 454
 - FL_Input_, 473
 - FL_Menu_, 492
 - FL_Spinner, 594
 - FL_Text_Display, 630
 - FL_Value_Input, 668
 - FL_Value_Output, 673
 - FL_Value_Slider, 676
- textsize
 - FL_Chart, 354
 - FL_File_Chooser, 400, 401
 - FL_Help_Dialog, 446
 - FL_Help_View, 454, 455
 - FL_Input_, 473

- Fl_Menu_, 492
- Fl_Spinner, 594
- Fl_Text_Display, 630
- Fl_Value_Input, 668
- Fl_Value_Output, 673
- Fl_Value_Slider, 676
- thread_message
 - fl_multithread, 266
- title
 - Fl_Help_View, 455
- tooltip
 - Fl_Widget, 704, 705
- top
 - Fl_Browser_, 340
- topline
 - Fl_Browser, 330
 - Fl_Help_View, 455
- type
 - Fl_File_Chooser, 401
 - Fl_File_Icon, 407
 - Fl_Label, 482
 - Fl_Spinner, 594
 - Fl_Widget, 705
- uncache
 - Fl_Bitmap, 310
 - Fl_Image, 461
 - Fl_Pixmap, 528
 - Fl_RGB_Image, 559
 - Fl_Shared_Image, 582
- uncheck
 - Fl_Menu_Item, 511
- undo
 - Fl_Input_, 473
- unhighlight
 - Fl_Text_Buffer, 615
- unlock
 - fl_multithread, 266
- unsubstitute_null_characters
 - Fl_Text_Buffer, 615
- up_down_position
 - Fl_Input_, 473
- update
 - Fl_Clock, 369
- update_child
 - Fl_Group, 443
- USER
 - Fl_Preferences, 538
- user_data
 - Fl_Widget, 705
- val
 - FL_CHART_ENTRY, 356
- valid
 - Fl_Gl_Window, 429
- value
 - Fl_Browser, 330
 - Fl_Button, 346
 - Fl_Check_Browser, 360
 - Fl_Choice, 365, 366
 - Fl_Clock_Output, 372
 - Fl_Color_Chooser, 377
 - Fl_File_Chooser, 401
 - Fl_File_Icon, 407
 - Fl_File_Input, 410
 - Fl_Help_Dialog, 446
 - Fl_Help_View, 455
 - Fl_Input_, 474
 - Fl_Input_Choice, 478
 - Fl_Menu_, 493
 - Fl_Menu_Item, 511
 - Fl_Positioner, 533
 - Fl_Progress, 551
 - Fl_Scrollbar, 574, 575
 - Fl_Spinner, 594
 - Fl_Tabs, 597
 - Fl_Valuator, 662
 - Fl_Wizard, 724
- version
 - Fl, 301
- visible
 - Fl_Browser, 331
 - Fl_File_Chooser, 401
 - Fl_Help_Dialog, 446
 - Fl_Menu_Item, 511
 - Fl_Widget, 706
- VISIBLE_FOCUS
 - Fl_Widget, 685
- visible_focus
 - Fl, 301
 - Fl_Widget, 706
- visible_r
 - Fl_Widget, 706
- visual
 - Fl, 302
- w
 - Fl_Help_Dialog, 447
 - Fl_Image, 461
 - fl_screen, 236
 - Fl_Widget, 706, 707
- wait
 - Fl, 302
- warning
 - group_comdlg, 278
- watch_widget_pointer
 - fl_del_widget, 269
- when

- Fl_Widget, [707](#)
- widget
 - Fl_Widget_Tracker, [711](#)
- window
 - Fl_Widget, [708](#)
- Windows handling functions, [219](#)
- word_end
 - Fl_Text_Buffer, [615](#)
 - Fl_Text_Display, [630](#)
- word_start
 - Fl_Text_Display, [630](#)
- wrap
 - Fl_Input_, [474](#)
- wrap_mode
 - Fl_Text_Display, [630](#)
- wrap_uses_character
 - Fl_Text_Display, [630](#)
- wrapped_column
 - Fl_Text_Display, [631](#)
- wrapped_line_counter
 - Fl_Text_Display, [631](#)
- wrapped_row
 - Fl_Text_Display, [631](#)
- x
 - Fl_Help_Dialog, [447](#)
 - fl_screen, [236](#)
 - Fl_Widget, [708](#)
- xbounds
 - Fl_Positioner, [534](#)
- xclass
 - Fl_Window, [721](#)
- xposition
 - Fl_Scroll, [571](#)
- xstep
 - Fl_Positioner, [534](#)
- xvalue
 - Fl_Positioner, [534](#)
- xy_to_position
 - Fl_Text_Display, [631](#)
- xy_to_rowcol
 - Fl_Text_Display, [632](#)
- y
 - Fl_Help_Dialog, [447](#)
 - fl_screen, [236](#)
 - Fl_Widget, [708](#)
- ybounds
 - Fl_Positioner, [534](#)
- yposition
 - Fl_Scroll, [571](#)
- ystep
 - Fl_Positioner, [534](#)
- yvalue