Linux

Gestion des répertoires
- `pwd` : Afficher le nom/chemin du répertoire de travail en cours.
- `ls [options] [fichier...]` : Afficher le contenu d'un répertoire (répertoire de travail par défaut).
  - Option `-l` : En plus du nom, afficher le type du fichier, les permissions d'accès, le nombre de liens physiques, le nom du propriétaire et du groupe, la taille en octets, et l'horodatage.
- `cd [repertoire]` : Changer de répertoire (répertoire "maison" par défaut).
- `mkdir [options] repertoire...` : Créer des répertoires.
- `rmdir [options] repertoire...` : Effacer des répertoires vides.

Gestion des fichiers
- `cp [options] fichier chemin` : Copier un fichier vers un autre fichier (`chemin`).
- `cp [options] fichier... repertoire` : Copier un ou plusieurs fichiers vers un répertoire.
- `mv [option...] source cible` : Renommer un fichier/répertoire
- `mv [option...] source... cible` : Déplacer un ensemble de fichiers/répertoires vers un répertoire cible.
- `rm [options] fichier...` : Effacer des fichiers
  - Attention : les fichiers effacés ne sont pas récupérables.

La commande `man`
- `man nom...` : Formater et afficher les pages de manuel en ligne

Exemple : `man pwd` affiche le manuel de la commande `pwd`.

Options du compilateur gcc

La commande `gcc` peut être exécutée avec des nombreuses options :

Option	Action
`-Wall`	afficher tous les avertissements concernant le code compilé
`-o`	spécifier le nom du fichier de sortie (programme exécutable) qui est par défaut `a.out` Exemple : `gcc -Wall -o prog prog.c`
`-O`	optimiser le code produit, en particulier *détecter les variables utilisées et non initialisées*
`-c`	génération du code objet sans édition des liens.
`-g`	ajouter des informations pour le débogage
`-v`	rendre le compilateur « bavard » (verbose) sur ces actions
`-I`, `-L`	spécifier explicitement des répertoires où le compilateur peut trouver des fichiers manquants (fichiers d'entête et bibliothèques)

Directives du préprocesseur

Définition de constantes :

```
#define NOM_CONST constante
```

Attention : la définition se fait sans ' ;' à la fin !

Inclusion de fichiers :

- `#include <nom_fichier>` : inclusion d'un fichier qui se trouve dans les répertoires standard (`/usr/include`, ...)
- `#include "nom_fichier"` : inclusion d'un fichier qui se trouve dans les répertoires définis dans l'environnement et, si le préprocesseur ne le trouve pas, dans les répertoires standard.

Expressions, Instructions et Blocs

Expressions :
- Une *expression* est composée d'un ou plusieurs *opérandes* (constantes, variables, expressions entre parenthèses) reliés par zéro ou plusieurs *opérateurs*.
- A chaque expression est associé un *type*.

Instructions : Une *instruction* est une expression qui se termine par un point-virgule ' ;'.

Blocs : Un *bloc* est une instruction composée
- d'une accolade ouvrante {,
- d'une liste de déclarations de variables internes (éventuellement vide),
- d'une liste d'instructions (éventuellement vide),
- d'une accolade fermante }.

Types C

Types arithmétiques :

Type de donnée	Signification	Taille (octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	$-32\,768$ à $32\,767$
unsigned short int	Entier court non signé	2	0 à 65 535
(signed) int	Entier	2 (proc. 16 bits) 4 (proc. 32 et 64 bits)	$-32\,768$ à $32\,767$ $-2\,147\,483\,648$ à $2\,147\,483\,647$
unsigned int	Entier non signé	2 (proc. 16 bits) 4 (proc. 32 et 64 bits)	0 à 65 535 0 à $4\,294\,967\,295$
long int	Entier long	4 8 (proc. 64 bits)	$-2\,147\,483\,648$ à $2\,147\,483\,647$ $-9\,223\,372\,036\,854\,775\,808$ à $9\,223\,372\,036\,854\,775\,807$
unsigned long int	Entier long non signé	4	0 à $4\,294\,967\,295$
float	flottant (réel)	4	$1.2 * 10^{-38}$ à $3.4 * 10^{38}$
double	flottant double	8	$2.2 * 10^{-308}$ à $1.8 * 10^{308}$
long double	flottant double long	12	$3.4 * 10^{-4\,932}$ à $1.2 * 10^{4\,932}$

Type `void` :
- ensemble vide : `void`

Types dérivés :
- tableaux : `[]`
- enregistrements : `struct`

Opérations arithmétiques binaires :

Symbole	Signification arithmétique
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste de la division

Opérations arithmétiques unaires :

Symbole	Signification arithmétique
`++`	incrémentation (types entiers)
`--`	décrémentation (types entiers)

Remarques :
- Si les deux arguments sont de types entiers, `/` correspond à la division euclidienne entière.
- Le résultat de la division est indéfini, si le second opérande vaut 0.
- Les instructions `i++;` et `i--;` sont équivalentes aux instructions d'affectation `i=i+1;` et `i=i-1;`.

La fonction `printf()`

Séquences d'échappement (fonction `printf`)

caractère	signification
\n	nouvelle ligne
\r	retour chariot
\t	tabulation horizontale
\f	saut de page
\v	tabulation verticale
\a	signal d'alerte
\b	retour arrière

Spécifications de conversion

	Type	Explication
`%d,%i`	`int`	notation décimale signée
`%u`	`int`	notation décimale non signée
`%c`	`int`	un seul caractère après conversion en `unsigned char`
`%s`	`char *`	chaîne de caractères jusqu'au premier `\0`
`%f, %2.7f`	`double`	notation décimale de la forme *`[-]mmm.dddddd`* (précision 6 par défaut)
`%e,E`	`double`	notation scientifique de la forme *`[-]m.dddddd[+-]xx`*
`%g,G`	`double`	notation décimale si l'exposant est compris entre -4 et la précision, scientifique sinon

L'instruction de sélection : `if`

Syntaxe de `if` avec `else` :

```
if (condition) {
  bloc_instructions_then;
}
else {
  bloc_instructions_else;
}
```

Syntaxe de `if` sans `else` :

```
if (condition) {
  bloc_instructions;
}
```


Opérateurs de comparaison :

	Signification arithmétique
`==`	égalité
`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal
`!=`	différent

Opérations booléennes :

	Signification logique
`a&&b`	et (conjonction)
`a||b`	ou (disjonction)
`!a`	non (négation)

Les boucles

Syntaxe de `while`

```
while (condition) {
     déclarations_de_variables_internes
     instructions_while
}
```

Syntaxe de `do-while`

```
do {
     déclarations_de_variables_internes
     instructions_do
} while (condition);
```

Syntaxe de `for`

```
for (init_compteur; condition_compteur; avance_compteur) {
     déclarations_de_variables_internes
     instructions_for
}
```

Les fonctions

Syntaxe de la définition d'une fonction

```
type_res ma_fonction ( type_arg1 arg1,
                       type_arg2 arg2,
                       ...,
                       type_argn argn ) {
  déclarations_de_variables_internes
  instructions_fonction
  return expr;
}
```

L'instruction **`return`** `expr;` permet de quitter la fonction :
- Elle peut apparaître partout dans le corps de la fonction où une instruction est acceptée.
- L'expression `expr` est de type `type_res` et sa valeur sera le résultat de la fonction.
- Si `type_res` est égal à `void`, l'expression `expr` est vide et la fonction est quittée par l'instruction **`return;`** ou à la fin des instructions.

Les tableaux

Déclaration d'un tableau à une dimension

```
type_element nom_du_tableau[nombre_elements];
```

Déclaration d'un tableau à une dimension avec initialisation

```
type nom_du_tableau[N] = {const_1,const_2,...,const_N};
```

Expression d'accès à un élément d'un tableau à une dimension

```
nom_du_tableau[indice]
```

Déclaration d'un tableau à n dimensions

```
type_champ nom_var[N_dim1][N_dim2]...[N_dimn];
```

Déclaration d'un tableau à 2 dimensions avec initialisation

```
type nom_du_tableau[M][N] = {
  {c_0_0, c_0_1, ..., c_0_(N-1)},
  ...,
  {c_(M-1)_0, c_(M-1)_1, ..., c_(M-1)_(N-1)}
};
```

Expression d'accès à un élément d'un tableau à n dimensions

```
nom_du_tableau[i1][i2]...[in]
```

Les pointeurs

Opérateurs

– `&x` retourne un *pointeur* de type `type_var *` vers la zone mémoire *identifiée par la variable* `x` de type `type_var` :

```
type_var x;
type_var *p=&x;
```

– `*p` retourne la valeur de la zone mémoire *référencée par le pointeur* `p` :

```
type_var y=*p;
```

Fonctions

– `sizeof(nom_var)` retourne la taille de la zone mémoire identifiée par `nom_var` en nombre d'octets.
– `malloc(taille)` retourne un pointeur sur une zone mémoire "libre" de `taille` octets.

Tableaux et pointeurs

Une variable de type tableau est un pointeur *constant* sur une zone mémoire continue. Ainsi, l'expression
– `tab` est équivalente à `&(tab[0])`
– `tab+i` est équivalente à `&(tab[i])`
– `tab[i]` est équivalente à `*(tab+i)`

Les entrées/sorties

Gestion des flux :

Fonction	Signification
`FILE *fopen(char *nom_fichier, char *mode)`	création (si nécessaire) et ouverture d'un fichier textuel ou binaire ; le résultat est un flux de type `FILE *`
`int fclose(FILE *flux)`	fermeture d'un flux
`int fseek(FILE *flux, long offset, int origin)`	positionne le pointeur de fichier *binaire* à la position `offset` à partir de `origin` qui peut avoir comme valeur `SEEK_SET` (début), `SEEK_CUR` (position courante) ou `SEEK_END` (fin) ; pour un pointeur de fichier textuel, `offset` doit valoir zéro ou une valeur retournée par `ftell()` (`origin=SEEK_SET`)
`int ftell(FILE *flux)`	retourne la position courante dans `flux`
`void rewind(FILE *flux)`	repositionnement : équivaut `fseek(flux, 0L, SEEK_SET); clearerr(flux);`

Modes de la fonction `fopen` :

Mode	Signification
`r, rb`	ouverture d'un flux (binaire) en lecture
`r+, rb+`	ouverture d'un flux (binaire) en lecture et écriture
`w, wb`	ouverture d'un flux en écriture
`w+, wb+`	ouverture d'un flux (binaire) en lecture et écriture (le contenu du fichier est effacé s'il existe déjà)
`a+, ab+`	ouverture d'un flux (binaire) en écriture pour l'écriture à la fin (append)

Entrées/sorties standard :

Pointeur de fichier	Signification
stdin	entrée standard
stdout	sortie standard
stderr	sortie erreur

Lecture et écriture sur les entrées/sorties standard :

Fonction	Signification
`int printf(char *format, ...)`	écriture vers la sortie standard
`int putchar(int c)`	écriture d'un caractère vers `stdout` (`putc(c,stdout)`)
`int puts(char *s)`	écriture d'une chaîne de caractères vers `stdout` en ajoutant le caractère '`\n`'
`int getchar(void)`	lecture d'un caractère de l'entrée standard (`getc(stdin)`)
`int scanf(char *format,...)`	lecture de valeurs de l'entrée standard en accord avec un format

Entrées/sorties textuelles :

Fonction	Signification
`int fgetc(FILE *flux)`	lecture d'un caractère
`int getc(FILE *flux)`	définition macro de `fgetc()`
`char *fgets(char *s, int n, FILE* flux)`	lecture d'une chaîne de caractères (au plus $n-1$ caractères)
`int fscanf(FILE *flux, char *format, ...)`	lecture de valeurs en accord avec un format (retourne le nombre d'objets convertis et affectés ou `EOF` si la fin du fichier a été atteinte ou une erreur s'est produite)
`int ungetc(int c, FILE *flux)`	remet le dernier caractère `c` lu dans le flux
`fprintf(FILE *flux, char *format,...)`	écriture de valeurs en accord avec un format (`printf` vers un flux spécifié explicitement)
`int fputc(int c, FILE *flux)`	écriture d'un caractère `c`
`int putc(int c, FILE *flux)`	définition macro de `fputc()`
`int fputs(char *s, FILE *flux)`	écriture d'une chaîne de caractères `s`

Entrées/sorties binaires :

Fonction	Signification
`size_t fread(void *ptr, size_t size, size_t nobj, FILE *flux)`	lit sur le flux binaire `flux` au plus `nobj` objets de taille `size` et les place dans le tableau `ptr` (retourne le nombre d'objets lus)
`size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *flux)`	écrit sur le flux binaire `flux` au `nobj` objets de taille `size` et les place dans le tableau `ptr` (retourne le nombre d'objets écrits)

Gestion de fichiers :

Fonction	Signification
`int remove(char *nom_fichier)`	effacement d'un fichier
`int rename(char *nom_avant, char *nom_apres)`	renommage d'un fichier
`FILE *tmpfile()`	création d'un flux binaire temporaire

Gestion d'erreurs :

Fonction	Signification
`int feof(FILE *flux)`	retourne une valeur non nulle si la fin de `flux` a été atteinte

Gestion de tampons :

Fonction	Signification
`int fflush(FILE *flux)`	vider le tampon associé à `flux`

Les enregistrements

Définition d'un type enregistrement :

```
struct id_enreg {
  declaration_var1;
  declaration_var2;
  ...
  declaration_varn;
};
```

– `declaration_vari` est une déclaration d'une ou de plusieurs variables (membres).
– Tous les identifiants de variables doivent être *uniques* dans l'enregistrement.

Déclaration avec définition du type :

```
struct id_enreg {
  ...
} id_var;
```

Déclaration avec définition du type séparée :

```
struct id_enreg {
  ...
};
struct id_enreg id_var;
```

Déclaration avec initialisation :

```
struct id_enreg id_var = { expr_1, ... expr_k };
```

– k est le nombre de membres définis dans `id_enreg`.
– On peut définir une structure, déclarer (et initialiser) des variables dans une seule instruction

Accès aux membres d'un enregistrement :

```
struct id_enreg{ ...; type_i id_i, ...} var, *pvar;
```

– `var.id_i` correspond au membre `id_i` de type `type_i`.
– `pvar->id_i` correspond au membre `id_i` de type `type_i`.
– `pvar->id_i` est équivalent à `(*pvar).id_i`.