Introduction à la programmation impérative en C

li205 – CINI

Document de cours pour les étudiants en formation à distance

Année 2009-2010

Table des matières

1 Introduction **3**
- 1.0.1 Ouvrages de référence . . . 3
- 1.1 Le langage C . . . 3
  - 1.1.1 Bref historique . . . 3
  - 1.1.2 Principe général . . . 4
  - 1.1.3 Les différentes phases de la compilation . . . 5

2 Opérateurs de base, Instructions, Expressions **7**
- 2.1 Expressions et types . . . 7
  - 2.1.1 Les types de base . . . 7
  - 2.1.2 Les opérateurs sur les types de base . . . 9
  - 2.1.3 Le type char* . . . 10
- 2.2 Variables . . . 11
  - 2.2.1 Définition . . . 11
  - 2.2.2 Déclaration de variable . . . 11
  - 2.2.3 Initialisation de variables . . . 11
  - 2.2.4 Utilisation de variables . . . 12
- 2.3 Instructions . . . 12
  - 2.3.1 Séquence et bloc d'instructions . . . 12
  - 2.3.2 Instructions d'affectation . . . 12
  - 2.3.3 Incréments et décréments . . . 13
  - 2.3.4 Instruction conditionnelle . . . 14
- 2.4 Instructions de boucles . . . 16
  - 2.4.1 Boucle **`while`** . . . 16
  - 2.4.2 Boucle do-while . . . 17
  - 2.4.3 Boucle for . . . 18
  - 2.4.4 Remarques sur les boucles . . . 18
  - 2.4.5 Instructions de contrôle . . . 19

3 Fonctions et programme **20**
- 3.1 Fonctions . . . 20
  - 3.1.1 Syntaxe . . . 20
  - 3.1.2 Appel de fonctions . . . 21
  - 3.1.3 Fonctions récursives . . . 23
- 3.2 Programme . . . 24
  - 3.2.1 La fonction main . . . 24
  - 3.2.2 Structure d'un programme . . . 24
- 3.3 Bibliothèques . . . 25
  - 3.3.1 La bibliothèque *stdio* . . . 25
    - 3.3.1.1 La fonction `printf` . . . 25
    - 3.3.1.2 La fonction `scanf` . . . 27
  - 3.3.2 La bibliothèque *stdlib* . . . 27
  - 3.3.3 La bibliothèque *time* . . . 27

3.3.4 La bibliothèque *string* . . . 28
3.3.5 La bibliothèque graphique de CINI . . . 28

4 Pointeurs et tableaux **30**
4.1 Pointeurs . . . 30
4.1.1 Définition . . . 30
4.1.2 Syntaxe . . . 30
4.1.3 Exemples d'utilisation . . . 32
4.2 Les tableaux . . . 34
4.2.1 Définition . . . 34
4.2.2 Syntaxe . . . 34
4.2.3 Tableaux et représentation mémoire . . . 35
4.2.4 Accès aux éléments d'un tableau . . . 37
4.2.5 Les chaînes de caractères . . . 37
4.2.6 Parcours d'un tableau . . . 40
4.2.7 Fonctions et tableaux . . . 42
4.2.8 Fonctions récursives et tableaux . . . 43

5 Les enregistrements **45**
5.1 Principe général . . . 45
5.1.1 Déclaration de type . . . 45
5.1.2 Déclaration de variables . . . 45
5.1.3 Accès aux données . . . 46
5.1.4 Initialisation de variables . . . 46
5.1.5 Enregistrements, tableaux et pointeurs . . . 47
5.1.5.1 Enregistrements et tableaux . . . 47
5.1.5.2 Recopie d'enregistrements . . . 48
5.2 Un exemple d'utilisation . . . 48
5.3 Quelques remarques pour finir . . . 49

6 Algorithmes classiques **50**
6.1 Produit de matrices . . . 50
6.2 Recherche dichotomique . . . 52
6.3 Tris . . . 54
6.4 Un exemple très complet : le pivot de Gauß . . . 56
6.4.1 Systèmes linéaires . . . 56
6.4.2 Algorithme du pivot de Gauß . . . 57
6.4.3 Méthode du pivot de Gauß . . . 57
6.4.4 Résolution d'un système triangulaire supérieur par l'algorithme de remontée . . . 58
6.4.5 Pivot de Gauß en C . . . 58

Chapitre 1

Introduction

Ce premier chapitre présente le contenu de l'unité d'enseignement (UE) ≪ Initiation à la programmation impérative en C ≫ (CINI) et définit les principes généraux du langage C tels que nous les utilisons dans cette UE. Les notions présentées dans ce chapitre sont abordées dans la première semaine de TP.

1.0.1 Ouvrages de référence

Si vous souhaitez aller plus loin dans l'apprentissage du langage C, ou simplement si vous souhaitez consulter d'autres ouvrages pédagogiques, nous vous recommandons particulièrement la lecture des livres suivants.

Le langage C : norme ANSI, par Brian W. Kernighan & Denis M. Ritchie, chez Dunod. Appelé familièrement ≪ le Kernighan & Ritchie ≫, c'est LA référence du langage. Cependant s'il est relativement pédagogique, il s'adresse à des lecteurs ayant déjà l'habitude de manipuler des concepts en informatique ; il propose de nombreux exercices dont les corrigés sont donnés par

Exercices corrigés sur le langage C : Solutions des exercices du Kernighan et Ritchie, par Clovis-L Tondo & Scott-E Gimpel, chez Dunod.

Si vous voulez un introduction qui se rapproche plus d'un cours introductif pour néophyte, alors le livre *Programmer en langage C*, par Claude Delannoy, chez Eyrolles, vous conviendra sans doute mieux.

Langage C par Samuel P. Harbison & Guy L. Steele, chez Dunod, est à la fois un bon ouvrage de référence et une bonne description des principales bibliothèques disponibles en standard.

Il existe de nombreux livres sur l'écriture de code C élégant, idiomatique, etc. Citons ici l'ouvrage en anglais *C : a software engineering approach*, par Peter A. Darnell & Philip E. Margolis, chez Springer-Verlag, qui constitue une excellente réflexion sur la conception et la qualité d'un code C d'envergure.

Enfin un excellent ouvrage, malheureusement disponible seulement en anglais à ma connaissance, *C programming : a modern approach* par K. N. King publié chez Norton & Compagny, fait actuellement autorité en matière de clarté, de pédagogie et de complétude dans les universités américaines du débutant au programmeur chevronné.

1.1 Le langage C

Après une présentation rapide du rôle du langage C dans l'histoire de l'informatique, nous présentons dans cette section les principes généraux du langage C et les commandes que vous devrez utiliser pour travailler en TP.

1.1.1 Bref historique

Le langage C a été inventé en 1972 par Dennis Ritchie et Ken Thompson des laboratoires d'AT&T. L'objectif du langage était double. Il s'agissait tout d'abord de pouvoir écrire facilement les systèmes Unix dont

l'architecture venait d'être proposée, mais aussi de développer des programmes de plus haut niveau fonctionnant sous Unix. En 1978, Brian Kernighan et Dennis Ritchie publient la première définition du langage C dans le livre « *The C Programming language* », considéré comme la référence du langage. En 1989, l'American National Standards Institute (ANSI) propose une normalisation du langage C (on parle du C-ANSI). En 1990, l'Organisation de Standardisation Internationale (ISO) en fait un standard. Depuis cette date, le langage n'a pas (ou peu) connu d'évolution.

Le langage C fait partie de la famille des *langages impératifs*, tout comme VBA ou PHP, par opposition aux langages fonctionnels (Scheme, CAML...) ou aux langages à objets (C++, Java, Eiffel). La programmation impérative consiste à décrire les opérations en termes d'états du programme, c'est-à-dire de valeurs affectées à des *variables* et de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Une recette de cuisine est aussi un exemple de programmation impérative : chaque étape de la recette correspond à une instruction et le monde physique constitue l'état du programme.

La quasi-totalité des micro-processeurs qui équipent les ordinateurs est de nature impérative : ils sont faits pour exécuter du code écrit en *assembleur*, c'est-à-dire le langage propre au processeur, utilisant un jeu d'instruction minimal. Les langages de haut-niveau (comme C) utilisent des instructions plus complexes que les langages d'assembleur, ce qui permet une programmation plus aisée, avec des structures de boucle et des fonctions. Ces instructions sont ensuites traduites en assembleur au moyen d'un *compilateur*, différent pour chaque machine.

La spécificité du langage C est qu'il est à la fois un langage de haut niveau et qu'il permet, en particulier grâce aux *pointeurs*, de travailler à un niveau proche de la machine, offrant ainsi un accès complet aux primitives « bas niveau » du système. Cela lui assura dès son origine un succès qui ne s'est jamais démenti depuis.

Le langage C a été utilisé non seulement dans la définition de tous les systèmes UNIX, mais il est à la base de la syntaxe de la plupart des langages de programmation impératifs et objets apparus dans les années 90, comme C++, Java, VBA (tous ces langages ont un « cœur » programmé en C). L'apprentissage du langage C constitue donc une excellente base pour aborder les notions avancées que l'on trouve dans les langages modernes. De plus, le langage C est omniprésent dans tous les systèmes d'exploitation grand public, comme Microsoft-Windows, Mac-OS ou Linux. C'est aussi pour cette raison que vous trouverez des compilateurs et des environnements de programmation pour le C sous n'importe lequel de ces systèmes.

1.1.2 Principe général

Le langage C n'est pas un langage interprété (contrairement à Scheme, par exemple), c'est-à-dire que l'ordinateur n'exécute pas les lignes du programme au fur et à mesure qu'il les lit. C'est un langage compilé, c'est-à-dire qu'un programme spécifique, le *compilateur*, transforme votre programme C en une série d'instructions en binaire qui constituent un programme exécutable (interprétable directement par l'ordinateur). On distingue donc en C le programme *source* (celui que vous avez écrit) du programme exécutable (celui que vous produisez lorsque vous *compilez*). Si le programme source est globalement indépendant du système d'exploitation que vous utilisez, le programme exécutable est, lui, dépendant de votre système d'exploitation, voire de votre ordinateur.

Pour écrire un programme C, vous utiliserez un *éditeur de texte* (l'éditeur recommandé dans l'environnement de TP est `gedit`) dans lequel vous taperez les instructions de votre programme source. Vous enregistrerez vos programmes dans des *fichiers sources* qui auront systématiquement l'extention `.c` (par exemple : `td1-exercice1.c`).

Pour compiler un programme, vous utiliserez la commande `gcc` dans un terminal, avec toutes les options d'avertissement (ces options sont présentées dans le TP de la première semaine) :

gcc `-Wall -Werror -O` *`fichier_source.c`* `-o` *`programme_executable`*

où *`fichier_source.c`* est le nom de votre fichier source et *`programme_executable`* celui que vous souhaitez donner à votre programme exécutable. Par exemple :

```
gcc -Wall -Werror -O td1-exercice1.c -o mon_premier_programme.bin
```

Les noms des fichiers peuvent éventuellement être précédés de chemins dans l'arborescence de fichiers :

```
gcc -Wall -Werror -O ~/cini/semaine1/exercice1.c -o ~/cini/bin/exo1
```

Dans cet exemple, le fichier source `exercice1.c` du répertoire `cini/semaine1` de votre dossier personnel est compilé en un fichier exécutable `exo1` dans le répertoire `cini/bin`. La manipulation de cette arborescence est présentée dans le TP de la première semaine.

Pour exécuter votre programme, il suffit de donner son nom précédé de son chemin d'accès (`./` si le programme est dans le répertoire courant). Par exemple :

```
./mon_premier_programme.bin
~/cini/bin/exo1
```

Un programme exécutable n'est pas la traduction littérale de son programme source. En particulier, il peut regrouper plusieurs fichiers sources en un seul programme et il intègre généralement le code de tout un ensemble de fonctions prédéfinies, que l'on trouve dans des *bibliothèques*, et dont certaines dépendent fortement du système d'exploitation. Dans l'UE CINI, nous ne verrons pas de fonctions dépendantes du système et nous n'aborderons pas la compilation multi-fichiers, mais nous utiliserons certaines bibliothèques standards.

1.1.3 Les différentes phases de la compilation

Du programme source au programme exécutable, il y a plusieurs étapes toutes réalisées par un compilateur moderne complet tel que `gcc` :

- bien sûr la *compilation* proprement dite
- mais auparavant le code source est traité par un *préprocesseur* qui retire les commentaires et effectue un certain nombre de substitutions textuelles indiquées par le programmeur par des instructions spéciales appelées *directives* pour le préprocesseur,
- et après la compilation des différents fichiers sources, l'*édition de liens* qui fabrique un exécutable à partir du résultat de leur compilation.

Nous allons revenir dans ce qui suit sur les commentaires puis sur les directives pour le préprocesseur.

Les commentaires

Il s'agit de texte libre inséré dans le programme pour faciliter sa compréhension à la relecture. Vous ***devez*** utiliser des commentaires dans vos programmes :

- Avant toute fonction pour expliquer ce que fait la fonction, ce que représentent les paramètres et quelles sont les hypothèses que vous faites dessus (par exemple, on suppose que n est plus petit que m) ;
- Avant chaque boucle (ou presque), pour expliquer ce qu'elle fait ;
- À chaque étape importante dans votre programme. En particulier, vous n'aurez jamais plus de 5 lignes de codes successives sans commentaire...

En C, les commentaires peuvent être insérés à n'importe quel endroit, sauf au milieu d'un nom de variable ou de fonction. Ils sont définis par les marqueurs `/*` (début) et `*/` (fin). Tout ce qui figure entre ces deux marqueurs est purement et simplement supprimé par le préprocesseur. Vous verrez à l'occasion des fins de ligne transformées en commentaire en ajoutant le marqueur `//` au début de la partie à commenter. Il s'agit là du début de commentaire standard de C++ toléré par certains compilateurs C, c'est cependant à éviter car cela donne du code non portable.

Voici un exemple simple de fonction C commentée :

```
/* La fonction abs_diff prend en entrée deux entiers i et j
   et retourne la valeur absolue de leur différence */
int abs_diff(int i, int j) {
  /* Si i>j, renvoyer i-j */
  if (i>j)
    return i-j;
  /* Sinon, renvoyer j-i */
  return j-i;
}
```

Les directives pour le préprocesseur

Dans ce cours, nous n'aborderons pas toutes les spécifités des directives pour le préprocesseur, qui peuvent être très riches, allant jusqu'à constituer un programme dans le programme. Nous nous intéresserons ici à deux directives :

`#include` pour les inclusions de bibliothèques : Dans un programme C, on utilise généralement un ensemble de fonctions prédéfinies, regroupées dans des bibliothèques. Par exemple, il y a une bibliothèque pour toutes les fonctions d'entrée-sortie (écrire dans un fichier, lire une saisie de l'utilisateur, etc). Pour indiquer au compilateur où se situent les fonctions prédéfinies que vous utilisez (c'est-à-dire quelle(s) bibliothèque(s) il doit inclure), vous utilisez en tout début de votre fichier source la directive **`#include`** dont la syntaxe est la suivante :

```
#include <nom_de_bibliotheque.h>
```

Par exemple, dans l'UE CINI, nous utiliserons souvent les bibliothèques suivantes :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Le fichier *nom_de_bibliotheque.h* contient une information sommaire composée essentiellement des définitions et déclarations de types et des prototypes des fonctions nécessaires à la compilation de fichiers faisant appel à la bibliothèque. Après la compilation, lors de l'édition de liens, le résultat de la compilation des différents fichiers sources et les bibliothèques complètes sont réunies en un exécutable.

`#define` pour les directives de réécriture : Il est possible en C de définir des opérations de réécriture (ou *macros*) qui consistent à remplacer toutes les occurences d'un mot par un texte donné. Pour définir des constantes en C, on utilise la primitive **`#define`** dont la syntaxe est la suivante :

```
#define mot texte
```

Par exemple :

```
#define N 12
```

signifie que le symbole N sera remplacé par la valeur 12 dans tout le programme par le préprocesseur.

Dans le cadre de l'UE CINI, nous utiliserons les directives de réécriture pour définir des constantes, lorsqu'une valeur caractéristique est utilisée de manière systématique tout au long du programme (par exemple, la taille d'un tableau). L'utilisation de constantes permet d'éviter les erreurs lorsque cette valeur doit être changée (il suffit de modifier la constante, tout le programme est alors affecté). Pour définir des constantes en C, on utilisera donc la primitive :

```
#define |NOM_CONSTANTE|  |valeur|
```

Attention ! *La définition de constantes en C n'est pas une opération anodine. Elle remplace toutes les occurrences du nom par sa valeur. En particulier, si vous définissez une variable qui porte le même nom que la constante, votre programme ne compilera plus car votre variable aura été « remplacée » par la valeur. Pour éviter ce type d'erreur, on prendra soin de définir systématiquement les constantes en MAJUSCULES, alors que les variables et les noms de fonctions seront en minuscules.*

De plus, faites bien attention à la différence de syntaxe entre la déclaration de variable et la directive **`#define`** pour définir une constante :

- Il n'y a pas de symbole « = » entre le nom et la valeur ;
- Il n'y a pas de symbole « ; » à la fin de la ligne.

Chapitre 2

Opérateurs de base, Instructions, Expressions

Note : les notions de ce chapitre sont étudiées dans les semaines de TD et de TP N°1 à 3.

Le langage C, comme tous les langages impératifs, repose sur la définition de séquences d'*instructions*. On distingue donc deux notions fondamentales en C : les *expressions*, qui peuvent être évaluées pour calculer un valeur, et les instructions, qui sont des expressions particulières se terminant par un point-virgule.

2.1 Expressions et types

À chaque expression peuvent être associés un *type* et une *valeur*. Par exemple, l'expression $(1 + 3)$ est d'un type « entier » (nous allons en voir un certain nombre dans la suite) et a pour valeur « 4 ». Nous allons considérer ici les types et les valeurs de base. Nous verrons par la suite des types plus élaborés.

2.1.1 Les types de base

Le langage C définit les types de base suivants :

`int` C'est le type pour les « entiers signés » de base. Les valeurs de type *int* sont représentées en notation binaire en mémoire sur 16 bits pour les architectures 16 bits (ordinateurs anciens ou processeurs embarqués), ou 32 bits sur les architectures 32 ou 64 bits (c'est-à-dire les ordinateurs actuels), le premier bit représentant le *signe* du nombre. Leur valeur va donc de -2^{15} à $2^{15} - 1$ sur une architecture 16 bits (c'est-à-dire de $-32\,768$ à $32\,767$) et de -2^{31} à $2^{31} - 1$ (c'est-à-dire de $-2\,147\,483\,648$ à $2\,147\,483\,647$) sur les architectures plus récentes.

`unsigned int` C'est le type pour les « entiers non signés ». Leur valeur va donc de 0 à $2^{16} - 1$ ($65\,535$) sur les architectures 16 bits et de 0 à $2^{32} - 1$ ($4\,294\,967\,295$) sur les architectures récentes.

`short int` Aussi noté **`short`**, c'est le type pour les « entiers courts », représenté sur 16 bits quelle que soit l'architecture machine. Leur valeur va donc de -2^{15} à $2^{15} - 1$.

`unsigned short int` Aussi noté **`unsigned short`**, c'est le type pour les « entiers courts non signés », dont la valeur va de 0 à $2^{16} - 1$.

`long int` Aussi noté **`long`**, c'est le type « entiers longs », représenté sur 32 bits pour les architectures 16 et 32 bits, et sur 64 bits pour les architectures 64 bits. Leur valeur sur ces architectures va donc de -2^{63} ($-9\,223\,372\,036\,854\,775\,808$) à $2^{63} - 1$.

`unsigned long int` Aussi noté **`unsigned long`**, c'est le type pour les « entiers longs non signés », dont la valeur va de 0 à $2^{64} - 1$ sur les architectures 64 bits.

`char` C'est le type pour les « caractères ». Aussi étonnant que cela puisse paraître, c'est avant tout un type numérique, pour les nombres représentés sur 8 bits. Leur valeur va donc de -2^7 (-128) à $2^7 - 1$ (127). Mais c'est aussi le type qui est utilisé pour représenter les caractères sur 7 bits en C, conformément à la table ASCII qui associe un code numérique pour chaque caractère. Par exemple, le caractère A (en majuscule) correspond au nombre 65 (de type **`char`**).

`unsigned char` C'est le type pour les « caractères non signés », dont la valeur va de 0 à $2^8 - 1$ (255).

`float` C'est le type pour les « nombres à virgule flottante » qui permet de représenter les nombres réels sur 32 bits. Le type *float* permet de représenter des nombres entre -3.4×10^{38} et 3.4×10^{38}. La représentation des nombres flottants en mémoire n'étant pas l'objet de ce cours. Le lecteur intéressé peut se référer aux ouvrages proposés en référence.

`double` Comme son nom l'indique, le type *double* permet de représenter des nombres à virgule flottante sur le double d'octets que celui utilisé par le type *float* (donc 64 bits), à la fois pour une meilleure précision et pour un intervalle de valeur plus grand (jusqu'à 1.8×10^{308}).

`long double` Le type *long double* permet de représenter des nombres à virgule flottante sur encore plus de bits : cela varie de 80 à 128 bits selon les ordinateurs.

Remarques importantes :

- Les types permettant de représenter des nombres réels ne peuvent pas être « non signés ».
- S'il est généralement inutile de retenir exactement la plage de valeur de chaque type, il est important de comprendre cette limite de la représentation des nombres en C. En effet, lors de l'évaluation des valeurs, seul le type de l'expression est pris en compte mais les calculs en binaire ne tiennent pas compte d'éventuels dépassements de capacité. Ainsi, dans le type **`int`**, l'expression `(125+10)` prend la valeur `135`. Mais dans le type **`char`** limité à 8 bits, la valeur `127+1` n'existe pas et sa représentation binaire correspond à la valeur `-128`. Ainsi, `(125+10)` prend la valeur `-121`, et ce sans aucun avertissement pour le programmeur...
  Les dépassements de capacité sont l'une des sources d'erreur les plus difficiles à détecter dans un programme C.
- Soulignons enfin qu'il n'y a pas de type pour les « booléens » (de même qu'il n'y a pas réellement de type pour les caractères, en dehors de leur représentation ASCII). En C, la valeur « faux » est représentée par le nombre entier `0` (quel que soit le type) et la valeur « vrai » par toute autre valeur entière.

Valeurs de base

Pour chaque type, il existe un ensemble d'expressions « atomiques » permettant de définir directement une valeur :

- Pour les types entiers (**`char`**, **`short`**, **`int`**, **`long`**) : toute séquence de chiffres sans espace, sans point, sans virgule, éventuellement précédée du signe « - » pour les valeurs signées. Par exemple :

  ```
  78676   -217676
  ```

  Par défaut, toute expression qui ne contient que des chiffres est de type **`int`**, si ce type permet de représenter cet entier, sinon elle est de type **`long`**. On peut forcer un autre type en ajoutant des indications si nécessaires (on parle alors de *transtypage explicite*). Nous présentons ce mécanisme au paragraphe 2.1.2.
- Pour les types flottants (**`float`**, **`double`**, **`long double`**) : une séquence éventuellement précédée du signe « - », contenant des chiffres et un point ou un exposant (c'est-à-dire le symbole « `e` » ou « `E` » suivi d'une séquence de chiffres éventuellement précédée du signe « - » ou du signe « + »). Par exemple voici quelques représentations flottantes de `57` :

  ```
  57.0   57.   57.0e0   57E0   5.7e1   5.7e+1   .57e2   570e-1
  ```

  Par défaut, toute expression qui contient un point ou un exposant est de type **`double`**.
- Pour le type *char*, un caractère entre guillemets simples :

  ```
  'z'   '3'   '('
  ```

  De plus, certains caractères de contrôle définis dans la table ASCII mais qui ne peuvent pas être saisis directement au clavier (par exemple le retour à la ligne ou la tabulation) sont représentés par un code d'échapement précédé du caractère *backslash* `\` :
  - `'\n'` : retour à la ligne
  - `'\r'` : retour chariot

- `'\t'` : tabulation horizontale
- `'\f'` : saut de page
- `'\v'` : tabulation verticale
- `'\a'` : signal d'alerte (bip sonore)
- `'\b'` : retour en arrière

Rôle des types

Le type associé à une valeur a tout d'abord un rôle de lisibilité pour le programmeur et permet une vérification de cohérence minimale : si l'on passe un type incompatible en argument à une opération, alors le compilateur détecte l'erreur et permet au programmeur de la corriger.

De plus, à un niveau plus proche de la machine, une même représentation en mémoire correspond à différentes valeurs. Le type permet de choisir l'interprétation qui convient.

Par exemple le nombre de 32 bits `m=1100 1111 1010 1001 1000 0100 0100 0110` admet les diverses interprétations suivantes selon le type associé :

- **`int`** : `-810974138`
- **`unsigned int`** : `3483993158`
- **`short int`** : `-31674`
- **`unsigned short int`** : `33862`
- **`char`** : `'F'`
- **`float`** : `-810974144.000000`

Le programme `representations_nombres.c` du polycopié de cours en présentiel illustre ce problème.

Transtypage implicite

Certaines valeurs d'un type donné peuvent être acceptées comme des expressions d'un autre type : on parle alors de *transtypage implicite*. Ainsi :

`char` < `short` < `int` < `long` < `float` < `double` < `long double`

C'est-à-dire :

- Une valeur de type **`char`** peut toujours être acceptée comme expression pour les types **`short`**.
- Une valeur de type **`short`** peut toujours être acceptée comme expression pour les types **`int`** ou **`long`**.
- ...

Par exemple, il est possible d'écrire l' instruction suivante en C :

```
short x = 's';
```

Si vous utilisez *explicitement* une valeur « plus grande » dans un type « plus petit », le compilateur vous préviendra. La seule exception est l'utilisation d'une valeur signée comme expression non-signée (par exemple `-123` dans un type **`unsigned char`**), pour lequel le dépassement de capacité ne sera pas toujours signalé.

En revanche, vous devez être bien conscients que le dépassement de capacité *au cours de l'exécution d'un programme* n'est *jamais* signalé [1].

Il existe aussi une forme explicite de transtypage dont nous allons parler à la fin de la sous-section suivante.

2.1.2 Les opérateurs sur les types de base

À partir des valeurs de base, il est possible de construire des expressions en utilisant toute une batterie d'opérateurs :

- Opérateurs sur les nombres : addition (+), soustraction (−), multiplication (⋆) et division (/) et les symboles de parenthèse. Par exemple :

  ```
  234*(53.24/-2.8)
  ```

  La division a cependant un sens particulier selon qu'il s'agit d'une expression entière ou à virgule :

1. En effet, le compilateur part du principe que vous savez que chaque type est un anneau fini et donc que vous ferez attention

Si les deux membres de la division sont des expressions de type entier, alors le symbole `/` représente la division euclidienne et l'expression a une valeur entière.

Par exemple, l'expression `(3/2)` est de type entier et vaut `1`, alors que l'expression `(3/2.0)` est de type flottant et vaut `1.5`. Ce double usage de la division est l'une des plus grandes sources d'erreur dans les programmes des étudiants en CINI.

– Reste de la division entière (ou *modulo*), avec le symbole `%`. Par exemple, `128%5` vaut `3`.
– Comparaison de valeurs : égal (==), différent (!=), supérieur (>), supérieur ou égal (>=), inférieur (<), inférieur ou égal (<=). Par exemple :

```
123.5 <= 998      12 == -3.8
```

Note : l'expression prendra la valeur 0 si elle est fausse et 1 sinon.
Attention au *double* symbole égal == pour l'opérateur d'égalité ! Une erreur extrêmement commune en C consiste à utiliser un simple = qui représente l'affectation de la valeur à droite du signe = à la variable placée à gauche de ce signe : cette instruction a quasiment toujours un code de retour nul (signifiant qu'elle s'est bien passée) sans liaison avec la comparaison que l'on souhaitait effectuer. Elle ne provoque pas d'erreur sauf si à gauche du signe = ne se trouve pas une variable. Elle provoque en général un *warning* qui s'avère bien utile pour détecter une erreur qui est le plus souvent une erreur d'inattention.
Nous verrons d'autres opérateurs sur les entiers tels qu'incrément et décrément par la suite.

– Opérateurs sur les booléens : conjonction (`&&`), disjonction (`||`), négation (`!`) et les symboles de parenthèse. Par exemple :

```
(1||0) && (!0)
```

Transtypage explicite

Il est possible d'utiliser un transtypage explicite pour convertir une expression d'un type donné en un autre type, en mettant le type souhaité entre parenthèse au début de l'expression. Il s'agit de changer la façon d'interpréter la valeur de l'expression stockée en mémoire. Il faut en user avec discernement car « transformer des choux en carottes » n'a pas nécessairement un sens.

Cependant, lorsqu'un type est converti en un type « inférieur » (cf. paragraphe précédent sur le transtypage implicite), l'opération peut avoir un sens tout à fait clair.

En particulier, cette opération est utilisée pour calculer la partie entière d'un nombre flottant. Par exemple :

```
(int)(3.2+2.1)
```

est une expression entière, dont la valeur est le nombre entier 5.

2.1.3 Le type char*

Le langage C n'utilise pas de type « chaîne de caractères ». Ces chaînes sont représentées par des tableaux de caractères se terminant par le caractère spécial `'\0'` (de valeur ASCII 0), représentant la fin de chaîne. La convention est donc d'utiliser le type pointeur `char*` pour représenter une chaîne de caractères. Les types pointeurs sont présentés plus en détail dans la section 4.1.

La valeur d'une chaîne de caractères est définie par une séquence de caractères entre guillemets doubles. Par exemple :

```
"Ce document contient\tdes notes de cours\npour le cours CINI"
```

Notons qu'une chaîne peut contenir des caractères de contrôle (`\t` et `\n` dans notre exemple ci-dessus).

Attention : puisque les chaînes de caractères sont en fait des pointeurs, on ne peut pas les comparer avec l'opérateur == (cet opérateur se contenterait de regarder si ces deux chaînes de caractères sont au même endroit dans la mémoire sans comparer leur contenu si elles sont à des emplacements différents en mémoire). Ainsi l'expression :

```
"abc" == "abc"
```

prend systématiquement la valeur *faux* (0). Pour comparer des chaînes, on utilise la fonction `strcmp` de la bibliothèque `string` (*cf.* paragraphe 3.3.4), qui prend en argument les deux chaînes et retourne un entier négatif si la première chaîne est plus « petite » (selon l'ordre lexicographique) que la seconde, nul si les deux chaînes sont identiques et positif si la première est plus grande que la seconde (toujours selon l'ordre lexicographique). L'ordre lexicographique est l'ordre du dictionnaire, c'est-à-dire que si le premier caractère des deux chaînes n'est pas identique alors c'est l'ordre sur ce premier caractère, sinon on compare les deuxièmes caractères des chaînes, etc. jusqu'à ce que les deux chaînes différent ou que l'une des deux soit épuisée.

Par exemple :

```
strcmp("abcd","aa")
```

retourne une valeur positive.

2.2 Variables

2.2.1 Définition

Une variable est caractérisée par son nom, son type et sa valeur.

Le nom d'une variable doit commencer par un caractère alphabétique ou par le symbole « _ » et peut contenir des caractères alphabétiques, des chiffres et le symbole « _ ». Par convention, les noms de variables sont généralement en minuscule. Par exemple :

```
i       toto2        ma_variable
```

Nous avons vu le triple rôle des types dans la section précédente.

2.2.2 Déclaration de variable

En langage C, une variable est définie à l'aide d'une instruction de la forme :

```
<type> <nom>;
```

Par exemple :

```
int i;
```

définit une variable i de type **`int`**. Il est possible de déclarer plusieurs variables d'un même type en les séparant par des virgules. Par exemple :

```
float a, b, c;
```

définit 3 variables `a`, `b` et `c` de type **`float`**.

2.2.3 Initialisation de variables

En C, lorsqu'une variable est créée, une zone mémoire de la taille adéquate lui est allouée mais aucune valeur par défaut n'y est enregistrée. On dit que la variable n'est pas **initialisée**. En fait, la valeur de la variable est définie par les bits préexistants dans la zone mémoire allouée, c'est-à-dire très probablement n'importe quoi. C'est pourquoi il faut systématiquement initialiser les variables en C, c'est-à-dire leur avoir affecté une (première) valeur, avant toute utilisation dans une expression.

Nous reviendrons sur l'affectation à la section suivante.

Lorsque l'on connaît la valeur d'initialisation d'une variable dès sa déclaration, on peut combiner cette (première) affectation avec la déclaration de la variable de la façon suivante :

```
<type> <nom> = <valeur>;
```

Par exemple :

```
char c1 = 'Z', c2 = 'k', c3;
```

déclare trois variables de type **`char`** : `c1` qui prend la valeur `'Z'`, `c2` qui prend la valeur `'k'` et `c3` qui n'est pas initialisée. Enfin, la « valeur » d'initialisation peut être une expression. Par exemple :

```
int trois = 1 + 2;
```

initialise la variable `trois` par la valeur `3`.

L'absence d'initialisation d'une variable est une cause d'erreur très courante en C. C'est une erreur pourtant aisée à détecter avec le compilateur `gcc` puisque l'option `-O` (qui est une option d'optimisation du code exécutable produit) détecte les variables non initialisées, mais l'option `-Wall` ne suffit pas.

2.2.4 Utilisation de variables

Une variable peut être utilisée comme une expression. Sa valeur est alors la valeur de la variable. Par exemple :

```
int a = 2;
```

déclare une variable `a` initialisée avec la valeur `2` et ensuite

```
int b = a + 1;
```

déclare une variable `b` initialisée avec la valeur de `a+1` c'est-à-dire `3`.

Ces deux déclarations auraient pu être faites simultanément par

```
int a = 2, b = a + 1;
```

On aurait ainsi déclaré une variable `a` de valeur `2` et une variable `b` de valeur `3` comme précédemment.

2.3 Instructions

Dans la section précédente, nous avons vu comment écrire des expressions de différents types et nous avons vu une première instruction : la déclaration de variables. Dans cette section, nous présentons les principales instructions du langage C.

2.3.1 Séquence et bloc d'instructions

Nous reviendrons par la suite sur les autres structures de contrôles du langage, mais nous avons d'ores et déjà besoin du mode le plus simple de combinaison d'instructions : la séquence, c'est-à-dire l'enchaînement des instructions, leur exécution dans l'ordre où elles apparaissent. Les instructions sont simplement écrites les unes à la suite des autres. Par convention, on se limite à une instruction par ligne. Par exemple, la séquence d'instructions suivante :

```
int a = 2;
float b = a+1.2;
```

déclare tout d'abord la variable `a` de valeur `2`. Puis, ensuite, elle déclare une variable `b` de valeur `2.2`.

Par ailleurs les structures de contrôle ainsi que la définition de fonction que nous verrons par la suite nécessitent que l'on délimite une séquence d'instructions, pour séparer le corps de la fonction du reste, pour délimiter ce qui doit être itéré ou exécuté seulement si une condition est vérifiée, etc. On appelle cela un *bloc* et on le délimite en C par des accolades.

2.3.2 Instructions d'affectation

L'instruction d'affectation permet de modifier la valeur d'une variable. Sa syntaxe est la suivante :

```
<nom> = <valeur>;
```

où *nom* est le nom de la variable et *valeur* est une expression dont la valeur sera enregistrée dans la variable (comme pour l'instruction d'initialisation). La nouvelle valeur écrase la précédente.

À titre d'exemple, considérons la séquence d'instructions suivante :

```
1       int a, b;
2       a = 3;
3       b = 5;
4       a = 12;
5       a = b + 1;
6       a = a + 1;
```

La première instruction déclare les variables `a` et `b` (sans les initialiser). Les deux instructions suivantes sont des instructions d'affectation qui mettent la valeur `3` dans `a` et la `5` dans `b`, ce qui les initialise. La quatrième instruction affecte à la variable `a` la valeur `12`. Dans la cinquième instruction l'expression `b+1` est évaluée et sa valeur `6` est affectée à la variable `6`. Enfin, dans la dernière instruction l'expression `a+1` est évaluée et sa valeur `7` est affectée à la variable `a`. À la fin de la séquence, la variable `a` a pour valeur `7`.

Dans cette dernière instruction, à gauche du signe = se trouve la variable, c'est-à-dire l'indication de l'emplacement mémoire où trouver le contenu de la variable, alors qu'à droite il s'agit de sa valeur c'est-à-dire du contenu lui-même.

Dans une instruction d'affectation, on évalue l'expression placée à droite du signe = puis on met dans la variable placée à gauche la valeur ainsi obtenue, que l'expression à évaluer fasse ou non référence à celle à affecter. L'utilisation du signe = synonyme à la fois de test d'égalité et de relation symétrique en mathématiques pour une opération fondamentalement dissymétrique (`a=b;` n'a rien à voir avec `b=a;` et encore moins `a=b+1;` avec `b+1=a;`) sont une difficulté de compréhension et une source d'erreur. Rappelons ici que le test d'égalité en C de ce fait s'écrit avec == pour éviter la confusion, mais cela reste une source d'erreur importante.

En C, une instruction d'affectation est aussi une expression, dont la valeur est la valeur affectée à la variable. Ainsi, on peut écrire :

```
int a, b;
a = (b=3);
```

L'affectation `b=3` a comme valeur `3` tout en affectant à la variable `b` la valeur `3` en sorte que cette instruction met dans *a* et *b* la valeur *3*. A priori, vous n'aurez jamais besoin d'utiliser une instruction comme expression dans le module CINI.

2.3.3 Incréments et décréments

Il est possible en C de combiner une instruction d'affectation et une opération arithmétique (addition, soustration, multiplication, division, modulo) en une seule instruction lorsqu'on souhaite modifier une variable à partir de sa propre valeur. On utilise pour cela les opérateurs `+=`, `-=`, `*=`, `/=` et `%=`. La définition générale de ces opérateurs peut être résumée de la manière suivante :

variable = variable <op> valeur ; ⇔ *variable <op>= valeur;*

Par exemple :

```
float a = 2;
a *= 2.5;
```

Cette instruction met dans `a` la valeur `5.0` résultat de l'évaluation de `2*2.5`.

De plus, il existe quatre opérateurs pour incrémenter ou décrémenter une variable entière :

– L'opérateur `++` en préfixe, qui est strictement équivalent à l'instruction `+=1` :

```
int a = 2;
++a;
```

À la fin de cette séquence d'instructions, `a` contient la valeur `3`. Cette instruction est aussi une expression. Elle a logiquement pour valeur la valeur résultat de l'incrément. Par exemple :

```
int a = 2, b;
b = ++a;
```

À la fin de cette séquence d'instructions, `a` et `b` prennent tous les deux la valeur `3`.

– L'opérateur ++ en postfixe, qui incrémente la variable mais qui, vue comme une expression, prend la valeur de la variable **avant l'incrément** :

```
int a = 2, b;
b = a++;
```

À la fin de cette séquence d'instructions, `a` vaut `3` mais `b` vaut `2`. Autrement dit, l'opérateur ++ en postfixe permet d'utiliser la valeur d'une variable dans une expression puis d'incrémenter la variable (alors que l'opérateur ++ en préfixe incrémente la variable avant de l'utiliser).

– L'opérateur −− en préfixe, qui est strictement équivalent à l'instruction `-=1` :

```
int a = 2, b;
b = --a;
```

À la fin de cette séquence d'instructions, `a` et `b` prennent tous les deux la valeur `1`.

– L'opérateur −− en postfixe, qui décrémente la variable et retourne la valeur **avant modification** :

```
int a = 2, b;
b = a--;
```

À la fin de cette séquence d'instructions, `a` vaut `1` et `b` vaut `2`.

Comme nous le verrons par la suite, ces opérateurs sont très souvent utilisés dans leur version postfixe, en particulier dans les boucles pour incrémenter le compteur de la boucle.

2.3.4 Instruction conditionnelle

Note : l'instruction conditionnelle est l'objet de la semaine de TD N°2.

L'instruction conditionnelle permet d'effectuer un ensemble d'instructions seulement lorsqu'une condition est vérifiée. Elle est définie à l'aide du mot-clef **`if`** de la manière suivante :

```
if (condition) {
  instruction;
  instruction;
  ...
}
```

Par exemple :

```
if (a<2) {
  a++;
}
```

est une instruction qui incrémente la valeur de `a` seulement si elle est plus petite que `2`.

Il est aussi possible de définir un ensemble d'instructions à effectuer lorsque la condition n'est pas vérifiée, à l'aide du mot-clef **`else`** :

```
if (condition) {
  instruction;
  ...
} else {
  instruction;
  ...
}
```

Par exemple :

```
if (a<2) {
  a++;
} else {
  a--;
}
```

Cette instruction incrémente la valeur de `a` lorsqu'elle est inférieure à 2 et la décrémente dans tous les autres cas.

Remarque

Lorsqu'il n'y a qu'une seule instruction à effectuer dans le bloc entre accolades, il est possible d'omettre les accolades. Par exemple :

```
if (a<2)
  a++;
```

Dans la mesure où c'est souvent une source d'erreurs (on rajoute une instruction à exécuter si la condition est vérifiée mais on oublie d'ajouter des accolades et la portée du **`if`** n'est pas modifiée, la seconde instruction est exécutée quelle que soit la valeur de la condition), nous nous efforçons dans ce cours d'utiliser systématiquement des blocs même lorsque ce n'est pas absolument nécessaire.

L'opérateur `switch`

L'opérateur **`switch`** permet de définir un ensemble d'instructions conditionnées par la valeur d'une variable **entière** *`variable`*. Sa syntaxe est la suivante :

```
switch (variable) {
  case valeur1:
    instruction;
    instruction;
    ...
  case valeur2:
    instruction;
    ...
  ...
  default:
    instruction;
    ...
}
```

Les $valeur_1$, $valeur_2$, ..., doivent être des constantes (pas de variable, ni d'appel de fonction dedans). Cependant cela peut être le résultat d'un calcul tel que `3+1`.

L'exécution d'une telle instruction se déroule de la façon suivante :

- soit *`variable`* fait partie des valeurs ainsi énumérées, par exemple $valeur_i$ et alors les instructions qui suivent la ligne **`case`** $valeur_i$**`:`** sont exécutées
- soit *`variable`* ne fait pas partie des valeurs ainsi énumérées, et alors les instructions qui suivent **`default:`** sont exécutées.

En pratique, la dernière instruction correspondant à un cas *doit* être **`break`** pour clore la liste des instructions à exécuter pour chacune des valeurs.

Attention : l'instruction **`break`** n'est pas obligatoire, le compilateur ne signalera pas son absence, mais l'exécution des instructions qui suivent un **`case`** $valeur_i$**`:`** ne s'arrête que si l'on rencontre un **`break`** ou l'accolade fermante qui clôt le bloc. En particulier elle ne s'arrête pas sur le **`case`** $valeur_{i+1}$**`:`** suivant. Ajouter systématiquement l'instruction **`break`** à la fin de chaque cas est donc nécessaire pour des raisons de clarté car diagnostiquer l'oubli d'un **`break`** est souvent pénible (aucun message du compilateur, comportement inattendu du programme qu'il faut décortiquer pour arriver à comprendre où il a cessé d'être normal, etc.)

On rencontrera à nouveau l'instruction **`break`** dans la section 2.4.5.

Considérons l'exemple suivant qui attribue une valeur à la variable `b` selon celle de la variable `a` :

```
switch (a) {
  case 1:
    b=1;
    break;
```

```
  case -2:
    b=2;
  default:
    b=3;
}
```

L'exécution de cette instruction se déroule de la façon suivante :

- si `a` vaut `1`, alors l'instruction `b=1` est exécutée puis l'instruction **`break`** arrête l'exécution du **`switch`** ce qui fait que `b` a pour valeur `1` à la sortie du **`switch`** ;
- si `a` vaut `2`, alors l'instruction `b=2` est exécutée, il n'y a pas d'instruction **`break`** donc l'exécution des instructions suivantes du **`switch`** continue et l'instruction `b=3` est exécutée puis l'accolade fermante arrête l'exécution de l'instruction, ce qui fait que `b` a pour valeur `3` à la sortie du **`switch`** ;
- si `a` ne vaut ni `1`, ni `2`, alors on est dans le cas par défaut et on exécute les instructions qui suivent le mot-clef **`default`**, soit `b=3`, puis l'accolade fermante arrête l'exécution de l'instruction, ce qui fait que `b` a pour valeur `3` à la sortie du **`switch`**

Remarque

Soulignons que la suite d'instructions conditionnelles :

```
if (variable==valeur_1)
  instruction_1;
else if (variable==valeur_2)
  instruction_2;
else ...
else
  instruction_N;
```

est strictement équivalente à :

```
switch (variable) {
  case valeur_1:
    instruction_1;
    break;
  case valeur_2:
    instruction_2;
    break;
  ...
  default:
    instruction_N;
}
```

2.4 Instructions de boucles

Note : Les boucles sont l'objet des semaines de TD N°2 et 3.

Les instructions de boucle permettent de répéter un ensemble d'instructions tant qu'une condition est valide.

2.4.1 Boucle `while`

La syntaxe d'une boucle **`while`** est la suivante :

```
while (condition) {
  instruction;
  ...
}
```

La condition est vérifiée **avant** l'exécution des instructions de la boucle. Tant que cette condition est vraie, l'ensemble des instructions est effectué.

Par exemple, on peut écrire la suite d'instructions suivante :

```
int a = 1, b = 0;

while (a<10) {
  b+=a;
  a++;
}
```

À la fin de cette boucle, `b` contient la somme des entiers de 1 à 9.

2.4.2 Boucle do-while

La boucle *do-while* est définie de la manière suivante :

```
do {
  instruction;
  ...
} while (condition);
```

Contrairement à la boucle **`while`**, la condition est vérifiée **après** l'exécution des instructions. Tant que cette condition est vraie, le système recommence l'exécution des instructions.

Par exemple, on peut écrire la suite d'instructions suivante :

```
int a = 10;
do {
  b+=a;
  a--;
} while (a>0);
```

À la fin de cette boucle, *b* contient la somme des entiers de 1 à 10.

Remarques importantes

– On utilise généralement une boucle *do-while* quand il est nécessaire de faire une première fois les instructions pour initialiser des variables. Par exemple, si l'on souhaite demander à l'utilisateur de saisir une valeur que l'on veut obligatoirement *positive*, on utilisera :

```
int v;

do {
  <demander de saisir la valeur de v>
} while (v<0);
```

On ne peut pas utiliser une boucle *while* ici sans donner une fausse valeur initiale à *v* !

– La condition dans l'instruction *do-while* clôt l'instruction et donc est suivie d'un point-virgule. Dans l'instruction **`while`** la condition ne clôt pas l'instruction puisque les instructions à exécuter dans la boucle n'ont pas encore été décrites. Il n'y a donc pas de point-virgule juste après la condition.
Si l'on met tout de même un point-virgule après la condition, alors la boucle **`while`** ne contient qu'une instruction vide et le bloc d'instructions n'a plus de lien avec la boucle. Soit la condition n'est jamais vérifiée, on n'entre pas dans la boucle mais on exécute le bloc ; soit la condition est vérifiée mais aucune modification n'est apportée à cette condition par l'instruction vide donc on exécute indéfiniment cette boucle vide. Dans l'un ou l'autre cas ce n'est pas le comportement attendu. C'est particulièrement dangereux car ce n'est pas incorrect donc le compilateur ne signale rien et là encore comprendre pourquoi un programme n'a pas le comportement attendu et isoler l'endroit et la nature de l'erreur n'est pas aisé.

2.4.3 Boucle for

La boucle *for* est utilisée le plus souvent pour effectuer un ensemble d'instructions un nombre fini de fois, **connu à l'avance** et **déterminé à partir d'une variable-compteur** (parce que c'est l'usage dans la plupart des langages, mais elle est beaucoup plus puissante que cela en C, comme on le verra par la suite). Sa syntaxe est la suivante :

```
for (<initialisation>;<condition>;<incrément>) {
  instruction;
  ...
}
```

Dans l'usage restreint que nous comptons en faire ici, l'initialisation, la condition et l'incrément porteront sur la variable-compteur. Par exemple :

```
int a, b=0;
for(a=1;a<=512;a=a*2) {
  b += a;
}
```

À la fin de cette boucle, `b` contient la somme des puissances de `2` entre `1` et `512`.

Remarques

- En fait, la boucle for n'est qu'une instruction *while* déguisée :

  ```
  <initialisation>
  while (<condition>) {
    instruction;
    ...
    <incrément>
  }
  ```

  et l'objectif est simplement de faciliter l'écriture de boucles avec incrément. Ainsi, il est possible d'utiliser des boucles **`for`** à la place de boucles **`while`** et réciproquement. Cependant, dans le cadre de l'UE CINI, nous restreindrons l'utilisation des boucles **`for`** au cas où le nombre d'itérations est déterminé à l'avance. Dans tous les autres cas, nous utiliserons une boucle **`while`**.
- Une erreur fréquente est l'ajout d'un `;` à la suite du triplet conditionnant l'exécution de la boucle comme dans l'exemple suivant :

  ```
  /* ATTENTION CETTE BOUCLE EST INCORRECTE */
  for (<initialisation>;<condition>;<incrément>);  {
            instruction;
  }
  ```

  Dans ce cas, la seule instruction de la boucle est l'instruction vide et votre boucle ne sert donc à rien (si ce n'est à modifier l'incrément). Les instructions que vous avez écrites entre les accolades ne sont exécutées qu'une fois, à la sortie de la boucle.

2.4.4 Remarques sur les boucles

- Comme pour l'instruction conditionnelle, lorsque la boucle ne contient qu'une seule instruction, il est possible d'omettre les accolades. Par exemple :

  ```
  while (a<10)
    b+=a;
  ```

- Une instruction dans une boucle peut très bien être une autre boucle. On parle alors de *boucles imbriquées*. Par exemple :

```
int i,j;

for(i=1;i<=10;i++) {
  for(j=1;j<=10;j++)
    printf("%d ",i*j);
  printf("\n");
}
```

permet d'écrire la table de multiplication pour les entiers de 1 à 10. L'instruction *printf* vue ici est présentée dans la section 3.3.1.

– L'une des erreurs les plus courantes dans les boucles *while* et *do-while* est l'oubli de l'incrément (ou de manière générale, d'une modification de valeur qui ferait que la condition pourrait un jour devenir fausse). Le programme boucle alors indéfiniment et il faut l'arrêter à la main (`Control+C`).

2.4.5 Instructions de contrôle

Il est possible d'utiliser certaines instructions spécifiques, appelées « instructions de contrôle » pour sortir d'une boucle ou d'une instruction *switch*. Dans le cadre de l'UE CINI, nous utiliserons les 4 instructions de contrôle suivantes :

– Nous avons déjà rencontré l'instruction **`break`** avec l'instruction **`switch`**. Dans le cas d'une boucle **`break`** permet de sortir de la boucle courante et est généralement utilisée dans une instruction conditionnelle.
Par exemple pour chercher si un entier inférieur à 10 admet un carré qui se termine par 6 en base 10, on pourrait écrire :

```
for(i=1;i<10;i++) {
  if (((i*i)%10)==6)
    break;
}
if (i < 10) {
   printf ("pas trouvé de carré se terminant par 6");
} else {
   printf ("pour i=%d, i*i=%d se termine par 6\n", i, i*i);
}
```

La boucle s'interrompt pour le premier entier qui vérifie la propriété et affiche un message
Attention : dans le cas de boucles imbriquées, l'instruction **`break`** sort seulement de la boucle la plus intérieure (et pas des boucles englobantes).

– L'instruction **`continue`** permet, sans sortir de la boucle, de passer directement au tour de boucle suivant : elle saute les instructions restantes du tour de boucle courant et va directement au test de la condition. Par exemple :

```
for(i=1;i<10;i++) {
  if ((i%7)==0)
    continue;
  b += i;
}
```

À l'issue de cette boucle, `b` contient la somme des entiers entre `1` et `9`, sauf les multiples de `7`.

– L'instruction **`return`** qui permet de sortir de la fonction courante (voir section 3.1).

– L'instruction `exit` qui permet de sortir du programme (voir section 3.2).

Dans le cadre de l'UE CINI, vous n'aurez pas besoin d'utiliser les instructions **`break`** et **`continue`** dans les boucles (les programmes seront suffisemment simples pour qu'il soit toujours possible de les écrires autrement, par exemple en utilisant une boucle **`while`** ou en inversant les conditions du **`if`**).

Chapitre 3

Fonctions et programme

Note : les notions de ce chapitre sont l'objet des semaines de TD et de TP N°4, N°8 et N°9.

3.1 Fonctions

3.1.1 Syntaxe

Une fonction est définie par :
- son nom ;
- la liste des paramètres auxquels elle s'applique, chacun ayant un type défini ; ces paramètres sont appelés *paramètres formels*.
- le type du résultat qu'elle produit. Si la fonction ne produit aucun résultat (par exemple dans le cas d'un affichage), le type associé est **`void`**.

Cet ensemble constitue la *signature de la fonction* (qu'on appelle aussi parfois *spécification* ou *prototype*). La signature de la fonction, accompagnée d'un commentaire approprié, doit permettre au programmeur d'utiliser cette fonction sans connaître la manière dont elle est réalisée. Par exemple :

```
/* renvoie le plus petit commun multiple des entiers a et b */
int ppcm(int a, int b)
```

indique que la fonction de nom *ppcm* opère sur deux paramètres entiers et renvoie un résultat entier. Le commentaire précise l'opération effectuée par la fonction.

Avec ces informations, un programmeur peut utiliser la fonction sans connaître les instructions exécutées pour calculer le résultat.

Le *corps de la fonction* est un bloc, c'est-à-dire une section de code entre accolades, qui va définir les opérations effectuées sur les paramètres pour obtenir le résultat.

Pour calculer ce résultat, la fonction peut avoir besoin d'utiliser des variables supplémentaires. On peut donc trouver dans le corps de la fonction une partie déclarative. Les variables qui y sont définies sont appelées *variables locales*. Elles ne sont pas accessibles en dehors de la fonction et n'existent que le temps que celle-ci s'exécute. On dit que leur *portée* ou leur *visibilité* est limitée à la fonction.

De ce fait, si un programme exécute plusieurs fois la même fonction, chaque instance de la fonction a ses propres variables et aucune valeur n'est conservée d'une exécution sur l'autre.

Instructions spécifiques

L'instruction `return` permet d'affecter un résultat à une fonction.

Elle n'est donc utilisée sans paramètre que si le type de retour de la fonction est **`void`**.

Lorsque le type de la fonction n'est pas **`void`**, alors tous les chemins d'exécution de la fonction (c'est-à-dire toutes les suites d'instruction qui peuvent apparaître lors de l'exécution de la fonction, quelles que soient les branches conditionnelles et boucles utilisées) doivent comporter une instruction **`return`** accompagnée d'une expression dont la valeur est du type du résultat. C'est la valeur de cette expression qui est affectée comme résultat à la fonction.

L'instruction **return** termine l'exécution de la fonction : aucune instruction de la fonction ne peut être exécutée après.

Dans l'exemple suivant, il y a deux chemins d'exécution possibles, suivant que *a* est positif ou non. On trouve dans chaque chemin une instruction **return** accompagnée d'une expression de type **float**. Lorsque *a* est négatif, l'exécution de **return**`(-a)` termine l'exécution de la fonction. **return** `a` ne sera donc pas exécuté à la sortie du **if**.

```
/* renvoie la valeur absolue de a */
float valeur_absolue(float a) {
   if (a < 0) {
      return (-a);
   }
   return a;
}
```

La fonction exit termine l'exécution du programme (et pas seulement de la fonction qui l'appelle à la différence de **return**). Son utilisation est donc généralement limitée au traitement de cas d'erreur.

La signature de cette fonction est **void** `exit(`**int** `status);`. La valeur affectée au paramètre *status* est transmise comme résultat du programme. Par convention, ce paramètre est choisi différent de 0, la valeur 0 étant réservée à la terminaison normale du programme.

Dans le cas particulier de la fonction `main` (voir paragraphe 3.2.1), l'instruction **return** a pour effet de sortir de la fonction principale et donc du programme. C'est pourquoi elle doit retourner un *status* entier (le fameux **return** `0;`).

3.1.2 Appel de fonctions

Lorsqu'on écrit une fonction, on ne fait que décrire une opération et les paramètres formels servent à définir le nombre, le type et la position des éléments manipulés par cette opération : il ne s'agit pas de variables du programme. L'opération n'est exécutée que lorsque la fonction est ppelée, et il faut alors préciser sur quelles expressions (ou constantes) du programme elle s'applique. Ces expressions (ou constantes) sont appelées *paramètres effectifs* et doivent avoir des types correspondants à ceux des paramètres formels.

Reprenons l'exemple de la fonction `valeur_absolue` ci-dessus, et considérons dans le même programme la fonction `main` suivante :

```
int main() {
   float x = -41.7;
   printf("valeur absolue : %2.1f\n", valeur_absolue(12.53));
   printf("valeur absolue de %f: %2.1f\n", x, valeur_absolue(x));
   return 0;
}
```

Le paramètre effectif du premier appel à `valeur_absolue` est `12.53` : il s'agit bien d'un flottant, comme le paramètre formel *a* de la fonction ; tout comme *x*, le paramètre effectif du deuxième appel. On peut noter que rien n'empêche d'avoir un paramètre effectif qui porte le même nom que le paramètre formel : si la variable *x* s'appelait *a*, l'exécution serait identique.

Lorsque *x* est utilisé comme paramètre effectif, ce qui est transmis à la fonction en réalité est une *copie* de la valeur de *x*. On peut le vérifier sur l'exemple suivant, qui reprend le calcul de la valeur absolue en modifiant cette fois-ci la valeur du paramètre :

```
/* renvoie la valeur absolue de a */
float valeur_absolue(float a) {
   if (a < 0) {
      a = -a;
   }
   return a;
```

```
}

int main() {
   float x = -41.7;
   printf("valeur absolue de x : %2.1f\n", valeur_absolue(x));
   printf("x = %2.1f\n", x);
   return 0;
}
```

La valeur affichée pour x est bien `-41.7`.

Nous verrons dans le chapitre suivant qu'il est possible de contourner ce mécanisme de copies à l'aide des pointeurs.

Certaines fonctions n'ont pas de paramètre. C'est le cas par exemple de la fonction `rand` de la bibliothèque `stdlib` (voir le paragraphe 3.3 de ce chapitre) dont la signature est la suivante :

```
int rand (void);
```

Le symbole **`void`** indiquant ici que la fonction *rand* ne prend par de paramètre.

Il faut cependant conserver les parenthèses à l'appel de la fonction (même si l'on ne met rien entre) pour que le compilateur reconnaisse qu'il s'agit d'un appel de fonction. Ainsi, on écrira :

```
int a = rand();
```

et non pas :

```
/* ATTENTION: L'INSTRUCTION SUIVANTE PROVOQUE
              UNE ERREUR À LA COMPILATION      */
int a = rand;
```

Fonctions et portée de variables

Les zones mémoire auxquelles une fonction a accès au cours de son exécution sont celles correspondant à :
- ses paramètres ;
- ses variables locales ;
- les variables globales du programme : ce sont des variables définies en dehors de toute fonction.

Chacun de ces éléments a une *portée* qui correspond à la portion du programme dans laquelle on peut y accéder. La portée d'une variable globale est l'intégralité du programme. Elle est donc accessible directement par n'importe quelle fonction du programme, ce qui signifie que la fonction peut la modifier sans qu'il soit nécessaire de la passer dans ses paramètres. Outre le fait que cela peut mobiliser inutilement de l'espace mémoire, cela rend difficile la localisation des instructions qui accèdent à cette variable. Il est donc recommandé d'éviter l'usage des variables globales.

Les variables définies dans une fonction sont locales à cette fonction (d'où leur nom) : les zones mémoire correspondantes sont affectées au programme pour la durée d'exécution de la fonction. Les modifications effectuées sur ces variables sont sans effet à l'extérieur de la fonction.

Lors de l'appel d'une fonction, les paramètres formels se comportent comme des variables locales : des zones mémoires lui sont affectées pour recevoir les valeurs des paramètres effectifs et cette affectation ne dure que le temps d'exécuter la fonction.

Tous ces éléments étant définis à différents endroits du programme, il se peut que certains aient des noms identiques. La règle générale est que l'élément désigné est celui dont la définition est "la plus proche" de l'instruction exécutée. Autrement dit :
- une variable locale masque une variable globale de nom identique : en d'autres termes, la variable globale devient inaccessible tant que la variable locale existe (donc pendant l'exécution de la fonction) ;
- un paramètre formel masque une variable globale de nom identique ; bien que n'ayant pas d'existence propre, le paramètre formel sert à désigner la donnée qui sera manipulée à l'appel de la fonction. Il faut donc une règle pour définir si les opérations agissent sur la variable globale ou sur la donnée transmise via le paramètre ;

– une variable locale et un paramètre formel ne peuvent pas avoir le même nom : ils ont la même portée. Cette situation provoque une erreur à la compilation.

3.1.3 Fonctions récursives

L'ensemble d'instructions constituant le corps d'une fonction peut inclure des appels de fonction. Lorsque, pour calculer son résultat, une fonction s'appelle elle-même, on parle de fonction récursive. Pour que cette suite d'appels converge vers un résultat, il y a deux conditions à respecter :

– Pour au moins une liste de valeurs de ses paramètres, la fonction s'exécute sans avoir besoin de s'appeler. Cette liste de valeurs est ce qu'on appelle un cas de base ;
– Lorsque la fonction s'appelle elle-même, elle doit modifier la valeur de ses paramètres à chaque appel. Cette modification doit garantir qu'à un moment, la liste des valeurs de ses paramètres correspondra à un cas de base.

Chaque invocation a ses propres instances de paramètres et ses propres variables locales. Considérons le programme ci-dessous, dans lequel la fonction *aff_liste* affiche les entiers de 0 à la valeur de son paramètre. La fonction `main` appelle la fonction *aff_liste* avec un paramètre effectif $n_0 = 2$ (les indices vont nous permettre de distinguer les différentes instances de paramètres).

```
#include <stdio.h>

void aff_liste(int n) {
    if (n >= 0) {
        aff_liste(n-1);
        printf("%d\n", n);
    }
}

int main() {
    aff_liste(2);
    return 0;
}
```

Nous déroulons ci-dessous l'exécution de l'appel *aff_liste(2)*. Puisque $n_0 \geq 0$, la fonction exécute *aff_liste(1)* puis l'instruction d'affichage avec $n = n_0$, mais seulement lorsque *aff_liste(1)* s'est terminée. *aff_liste(1)* est une nouvelle instance de la fonction à laquelle est associée une nouvelle instance n_1 du paramètre n.

```
aff_liste(n0 = 2)
    | aff_liste(n1 = 1)
    |     | aff_liste(n2 = 0)
    |     |     | aff_liste(n3 = -1)
    |     |     | printf("%d\n", n2 = 0);
    |     | printf("%d\n", n1 = 1);
    | printf("%d\n", n0 = 2);
```

Nous pouvons noter que dans cet exemple, le cas de base n'apparaît pas explicitement. Mais nous pouvons aussi remarquer que pour $n < 0$, la fonction ne fait rien (pas d'appel, pas d'affichage) et que la modification du paramètre entre deux appels aboutira à une valeur négative de n, quelle que soit la valeur d'où on est parti. $n < 0$ constitue donc un cas de base au sens où nous l'avons défini.

L'ordre d'affichage des valeurs est la conséquence directe de l'ordre d'appel des fonctions *aff_liste* et `printf` dans l'exécution de *aff_liste*. Si l'on réécrit la fonction *aff_liste* de la manière suivante :

```
void aff_liste(int n) {
    if (n >= 0) {
        printf("%d\n", n);
        affiche_liste(n-1);
    }
}
```

on commence par afficher la valeur de n avant d'exécuter l'appel récursif. Les valeurs seront donc affichées dans l'ordre n_0 puis n_1 et enfin n_2, soit l'ordre inverse du programme précédent.

3.2 Programme

3.2.1 La fonction main

Il existe deux signatures valides pour la fonction `main` :

- **`int`** `main(`**`void`**`)` : la fonction n'a pas de paramètre.
- **`int`** `main(`**`int`** `argc, `**`char`** `* argv[])` : cette signature est utilisée lorsque l'on veut transmettre des données au programme lors de son lancement. Le paramètre *`argc`* reçoit le nombre d'arguments de la commande d'exécution, y compris le nom du programme. Les éléments du tableau *`argv`* sont les chaînes de caractères représentant chacun de ces arguments (*`argv[0]`* est le nom du programme).

Dans les deux cas, la fonction renvoie une valeur de type entier. Par convention, sous unix, une commande dont l'exécution s'est déroulée sans problème renvoie zéro, les valeurs non nulles étant utilisées pour coder les différents cas d'erreur. Si le programme atteint la dernière instruction de la fonction `main`, c'est qu'il s'est déroulé normalement. On terminera donc la fonction par l'instruction **`return`** `0`.

3.2.2 Structure d'un programme

Un fichier contenant un programme doit être structuré de la manière suivante :

- en tête, les directives au pré-processeur : les opérations correspondantes sont exécutées avant la compilation. On trouve dans l'ordre :
  - l'inclusion des bibliothèques : **`#include`** *`<bib_predefinie.h>`*
  - la définition d'expressions à substituer : **`#define`** `N 100`
- les variables globales : il s'agit de variables accessibles par toutes les fonctions du programme, elles doivent donc être définies avant les fonctions. Il faut essayer d'en limiter au maximum l'utilisation.
- la liste ordonnée des fonctions : si une fonction *`f1`* appelle une fonction *`f2`*, alors la fonction *`f2`* doit figurer dans le programme avant la fonction *`f1`*. Comment faire alors si la fonction *`f2`* appelle à son tour la fonction *`f1`* ? Il faut dans ce cas insérer la signature de la fonction *`f1`* avant l'écriture de *`f2`*. En effet, les informations contenues dans la signature sont suffisantes pour que le compilateur puisse contrôler que l'appel de *`f1`* par *`f2`* est syntaxiquement correct.
- la fonction `main` : elle apparait en dernier car elle ne peut être appelée par aucune fonction du programme. Mais l'exécution du programme commence toujours à la première instruction de cette fonction.

Ainsi, un programme C aura toujours la forme suivante :

```
#include <les_bibliotheques.h>

#define MES_CONSTANTES VALEURS

/* Les variables globales */
int xg, yg;
float zg;

/* Fonctions annexes (dans l'ordre) */
int fonction1(...) {
  ...
```

```
}
int fonction2(...) {
  ... /* peut éventuellement utiliser la fonction 1 */
}
int fonction3(...) {
  ... /* peut éventuellement utiliser les fonctions 1 et 2 */
}

/* la fonction main, qui utilise les fonctions précédentes */
int main() {
  ...
  return 0;
}
```

3.3 Bibliothèques

De nombreuses opérations dans un programme sont effectuées au moyen de fonctions prédéfinies. C'est le cas lorsque ces opérations nécessitent une interaction avec le système d'exploitation et qu'il serait imprudent de les confier à un utilisateur inexpérimenté, par exemple pour les fonctions d'entrée/sortie. Ou alors, il s'agit d'opérations complexes mais souvent utilisées dans les programmes, comme la génération de nombres aléatoires.

Ces fonctions sont regroupées dans des bibliothèques thématiques, par exemple la bibliothèque `time` pour les fonctions de mesure du temps. Pour pouvoir utiliser une fonction prédéfinie, il faut préciser au compilateur dans quelle bibliothèque elle se trouve.

La section 3 du manuel en ligne permet d'obtenir des informations sur les fonctions de bibliothèque : leur signature, le type de valeur retournée et la bibliothèque dans laquelle elles sont définies. L'exécution à partir de la console de la commande :

```
man 3 rand
```

fournit les informations suivantes :

```
SYNOPSIS
     #include <stdlib.h>

     int rand(void);
```

ce qui nous indique que la fonction `rand` se trouve dans la bibliothèque `stdlib`, qu'elle ne prend pas de paramètre et qu'elle renvoie un résultat de type entier.

3.3.1 La bibliothèque *stdio*

Cette bibliothèque contient les fonctions d'entrée/sortie : celles qui permettent de transmettre des données au programme ou d'afficher des résultats. Nous en présentons uniquement deux : la fonction d'affichage `printf` et la fonction de saisie `scanf`.

3.3.1.1 La fonction `printf`

La fonction `printf` a un premier paramètre qui est une chaîne de caractères appelée *format*. Ce format peut inclure des *spécifications de conversion* qui déterminent sous quelle forme doit être affichée l'expression que l'on souhaite inclure à l'emplacement correspondant de la chaîne.

Par exemple, la valeur d'une variable représentant un entier peut être affichée en base 10 ou en base 16. Dans le premier cas, il faut utiliser la spécification de conversion `%d` et dans le deuxième cas `%x`. Ces spécifications s'appliquent à une expression de type **`int`**.

À chaque spécification de conversion dans la chaîne de format doit correspondre une expression d'un type compatible dans la liste des paramètres. Les substitutions se font dans l'ordre : la première spécification de

conversion s'applique au deuxième paramètre de `printf` (le premier paramètre est la chaîne de format), la deuxième au troisième paramètre, etc.

Dans le programme suivant, *i* est d'abord affiché en base 10, puis en base 16 :

```
#include <stdio.h>

int main() {
    int i = 321;

    printf("la representation de %d en base 16 est %x\n", i, i);
    return 0;
}
```

La fonction `printf` a un nombre de paramètres variable. Le premier paramètre est le format, les autres paramètres doivent être aussi nombreux que les spécifications de conversion qui apparaissent dans le format. Si le nombre de paramètres est incompatible avec le nombre de spécifications de conversion, la compilation provoque un avertissement (à condition d'avoir utilisé l'option `Wall`) : `too many` (ou `too few`) `arguments for format`. Dans le deuxième cas, les valeurs affichées lorsqu'aucun paramètre n'est fourni sont indéterminées.

Les formats de représentation en mémoire font que certains types de données sont compatibles. La suite de 8 bits utilisée pour représenter un caractère en mémoire peut aussi être interprétée comme la valeur d'un entier. Réciproquement, tout entier dont la valeur peut être codée sur 8 bits peut être représenté dans un programme par une variable de type caractère (**`char`**). La portion de code suivante est donc valide :

```
unsigned char i = 255;
printf("valeur de i : %d\n", i);
printf("code ASCII du caractere %c : %x\n", 'a', 'a');
```

Une spécification de conversion commence par le caractère `%`. Si l'on veut afficher un caractère `%` dans un texte, il faut utiliser `%%` dans la chaîne de format. Les spécifications de conversion les plus couramment utilisées sont :

Spécification	Type	Explication
%d, %3d, %i	int	notation décimale signée. %3d précise que l'entier est affiché sur exactement 3 caractères
%u	int	notation décimale non signée
%x, %X	int	notation hexadécimale non signée
%c	int	un seul caractère après conversion en unsigned char
%s	char *	chaîne de caractères jusqu'au premier `\0`
%f, %2.3f	double	notation décimale de la forme $[-]mmm.dddddd$ (précision 6 par défaut). L'ajout de `2.3` précise que l'on souhaite afficher 2 chiffres avant la virgule et 3 après
%e, %E	double	notation scientifique de la forme $[-]m.ddddddE[+-]xx$

L'utilisation d'une spécification de conversion incompatible avec le format de la donnée à afficher provoque un avertissement. Par exemple, la compilation du programme ci-dessous :

```
int main() {
  char c = 'z';
  printf("chaine : %s\n", c);
    return 0;
}
```

provoque le message suivant :

```
warning: format '%s' expects type 'char *', but argument 2 has
type 'int'
```

Ce message indique que le caractère est assimilé à un entier (nous venons de voir que ces deux types sont compatibles), alors que la spécification `%s` s'applique à une chaîne de caractères, laquelle est identifiée par l'adresse à laquelle elle est stockée en mémoire, qui est de type pointeur sur un caractère.

3.3.1.2 La fonction `scanf`

La fonction `scanf` utilise elle aussi une chaîne de format et une liste de variables, mais son rôle est d'affecter des valeurs dans les variables en fonction des données saisies par l'utilisateur. Puisque la fonction `scanf` affecte des valeurs dans des variables, il est nécessaire que ces variables soient passées par référence : on transmet à la fonction l'adresse en mémoire de la variable dans laquelle une valeur lue doit être stockée. L'adresse d'une variable *a* est obtenue par l'expression *&a* (voir le chapitre sur les pointeurs).

```
int main() {
   int i;

   printf("saisissez un entier :\n");
   scanf("%d", &i);
   printf("j'ai affecte a i la valeur %d\n", i);
   return 0;
}
```

On peut appeler la fonction une seule fois pour récupérer plusieurs valeurs. Dans l'exemple ci-dessous, l'exécution de `scanf` ne se termine que lorsque l'utilisateur a entré deux valeurs. Le fait de séparer les valeurs saisies par un espace, plusieurs espaces ou un retour à la ligne ne fait aucune différence au niveau de l'exécution.

```
int main() {
   int i;
   float f;

   printf("entrez un entier et un reel :\n");
   scanf("%d %f", &i, &f);
   printf("i = %d et f = %f\n", i, f);
   return 0;
}
```

3.3.2 La bibliothèque *stdlib*

La bibliothèque standard contient, entre autres, les fonctions permettant la génération d'une séquence pseudo-aléatoire.

- **`int`** `rand(`**`void`**`);` : cette fonction renvoie un entier compris entre `0` et la constante `RAND_MAX`, elle aussi définie dans la bibliothèque `stdlib`. L'ensemble des nombres renvoyés par une suite d'appels à la fonction forme une séquence pseudo-aléatoire.

En réalité, la fonction `rand` est complètement déterministe : elle calcule successivement les valeurs d'une suite u : $u_{n+1} = f(u_n)$. La séquence de nombres est donc complètement déterminée par la valeur de u_0, appelé graine (*seed* en anglais) de la suite. Pour obtenir des tirages différents chaque fois que l'on relance le programme, il est donc nécessaire de pouvoir modifier la valeur de la graine.

- **`void`** `srand(`**`unsigned`** `seed)` : cette fonction fixe la valeur de u_0 à la valeur du paramètre *seed*. Pour garantir que la valeur de *seed* est modifiée entre deux exécutions du programme, un artifice couramment utilisé est de lui donner comme valeur celle de l'heure courante Unix, obtenue au moyen de la fonction `time` de la bibliothèque *time*.

3.3.3 La bibliothèque *time*

La seule fonction que vous utiliserez dans la bibliothèque *time* dans le cadre de l'UE CINI est la fonction :

```
time_t time(time_t *tloc)
```

Cette fonction renvoie le nombre de secondes écoulées depuis le 1er janvier 1970, 0h00 (la date de création du système Unix). Le type `time_t` correspond généralement à **`long int`**. Sauf si on veut y recopier le

résultat de la fonction, le paramètre *tloc* reçoit pour valeur la constante NULL. Cette constante, définie dans la bibliothèque stdlib, est utilisée lorsqu'on veut affecter à une variable de type pointeur l'adresse d'une case mémoire dans laquelle on ne pourra pas stocker d'information.

Voici un exemple très simple de programme effectuant des tirages aléatoires à l'aides des bibliothèques *stdlib* et *time* :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i;

    srand(time(NULL));
    for (i = 0; i < 10; i++){
        printf("Valeur tiree aleatoirement : %d\n", rand());
    }
    return 0;
}
```

3.3.4 La bibliothèque *string*

Cette bibliothèque regroupe les opérations relatives aux chaînes de caractères. Parmi celles-ci, nous pourrons être amenés à utiliser :

- **int** strcmp(**const char** *s1, **const char** *s2) : cette fonction renvoie 0 si les chaînes *s1* et *s2* sont identiques, une valeur positive si *s1* est supérieure à*s2*, une valeur négative sinon. La relation d'ordre utilisée est un ordre lexicographique utilisant le codage ASCII des caractères. L'instruction `i = strcmp("abc", "ab_c");` affecte à *i* une valeur positive car le code ASCII de *c* est supérieur à celui de _.
- **char** *strcpy(**char** *dst, **const char** *src) : cette fonction recopie la chaîne *src* dans la chaîne *dst*, y compris le caractère de terminaison. La valeur de retour (que nous n'utiliserons pas) est *dst*. **Attention**, il n'y a pas de contrôle sur le fait que la taille de *dst* est suffisante pour recevoir *src*. Une mauvaise utilisation de la fonction peut donc engendrer les problèmes évoqués dans la section 4.2.5.
- size_t strlen(**const char** *s) : cette fonction renvoie le nombre de caractères qui composent la chaîne *s*, sans prendre en compte le caractère de terminaison \0.
  Par exemple, `strlen("toto")` renvoie la valeur 4.

3.3.5 La bibliothèque graphique de CINI

Les fonctions graphiques utilisées par le module CINI se trouvent dans la bibliothèque *graphics*. Le programme doit donc exécuter la directive **#include** "graphics.h" pour pouvoir y accéder.

Il faut noter ici l'emploi des guillemets au lieu de < >. En effet, cette bibliothèque ne se trouve pas dans le répertoire standard des bibliothèques (sous Unix, généralement /usr/include) mais dans un répertoire défini par la variable d'environnement INCLUDE.

Le programme doit ensuite être compilé avec la commande gcc_graphique (qui admet les mêmes options que gcc). Les principales fonctions de la bibliothèque sont :

- **void** creer_fenetre(**unsigned int** width, **unsigned int** height,
  char *bg_color, **char** *fg_color);

  crée une fenêtre de taille width × height pixels en définissant la couleur de fond (*bg_color*) et celle du crayon (*fg_color*). Une couleur est définie par une chaîne de caractères en anglais, par exemple "white". Le point de coordonnées (0, 0) se trouve en haut à gauche de la fenêtre.

- **void** attendre_fermeture_fenetre(**void**);
  empêche le programme de se terminer tant que l'utilisateur n'a pas fermé la fenêtre graphique (en cliquant sur la croix en haut à gauche). Si le programme n'appelle pas cette fonction, il ne dure que le temps (souvent très bref) d'exécuter ses calculs et sa terminaison entraîne la disparition de la fenêtre graphique qu'il a créée.
- **void** afficher_point(**unsigned int** x, **unsigned int** y);
  affiche le point de coordonnées (x, y).
- **void** afficher_ligne(**unsigned int** x1, **unsigned int** y1,
  unsigned int x2, **unsigned int** y2);

  trace une ligne entre le point de coordonnées (x1, y1) et le point de coordonnées (x2, y2). Ces points doivent se trouver à l'intérieur d'une fenêtre préalablement créée.

Chapitre 4

Pointeurs et tableaux

Note : les notions de ce chapitre sont l'objet des semaines de TD et de TP N° 5 à 7 et de la semaine N°10.

4.1 Pointeurs

Imaginez que vous avez à votre disposition, devant vous sur le sol, un ensemble de billes. Vous pouvez les prendre directement. Maintenant nous les avons rangées dans des casiers, une bille par casier, vous ne pouvez plus directement prendre les billes, vous devez d'abord identifier le casier dans lequel se trouve la bille et ouvrir ce casier pour la prendre. Pour indiquer à quelqu'un quelle bille prendre vous allez lui donner l'identité du casier qu'il doit ouvrir, cette information correspond à l'information contenue par un pointeur : l'endroit où est disponible une information et non l'information elle même.

4.1.1 Définition

En informatique, toutes les variables sont rangées dans des blocs de 8 bits consécutifs, que l'on appelle *octets*. Toutes les variables que vous déclarez sont stockées dans un ou plusieurs octets consécutifs selon leur type. Par exemple, sur une architecture 32 bits, une variable de type **`int`** est stockée sur 4 octets consécutifs. L'association entre une variable déclarée et les octets mémoires qui servent à la stocker s'appelle l'allocation mémoire. L'adresse d'une variable identifie le premier octet à partir duquel elle est stockée. Un pointeur sur une variable contient son adresse. Le type de la variable permet de savoir sur combien d'octets est stockée sa valeur. Si une fonction a accès à l'adresse d'une variable, elle peut modifier sa valeur même si la variable n'est pas locale à la fonction puisqu'elle a directement accès aux octets contenant la valeur de la variable.

4.1.2 Syntaxe

Voici la syntaxe permettant de déclarer un pointeur, d'accéder à l'adresse d'une variable ou d'accéder à sa valeur à partir de son adresse.

Déclaration

La déclaration d'un pointeur nécessite de connaître le type de l'objet dont le pointeur contiendra l'adresse et se fera de la façon suivante.

```
<type> *<nom> ;
```

L'instruction suivante déclare un pointeur `pA` sur un entier.

```
int *pA;
```

Attention, cette déclaration ne déclare pas l'entier mais uniquement le pointeur qui tant qu'il n'a pas été initialisé ne contient pas d'adresse valable. Pour l'initialiser, il faut lui donner l'adresse d'une variable entière qui a été préalablement déclarée.

Adresse d'une variable

Il est possible de connaître l'adresse de toute variable, pour cela, il suffit de faire précéder le nom de la variable du caractère `&`. Les instructions suivantes déclarent une variable `varA` de type réel et un pointeur `pA` sur un réel. On affecte ensuite à `pA` l'adresse de la variable `varA`.

```
float varA;
float *pA;

pA=&varA;
```

Si vous affectez à un pointeur l'adresse d'une variable dont le type n'est pas compatible avec la déclaration du pointeur vous obtiendrez le message suivant lors de la compilation : `warning : assignment from incompatible pointer type`. C'est ce qui se passe lorsque vous affectez l'adresse d'une variable de type entier à un pointeur sur un réel par exemple. Vous devez absolument considérer ces `avertissements` comme des `erreurs` et les corriger avant d'exécuter vos programmes.

Accès à une adresse

Il est possible d'accéder à la valeur se trouvant à l'adresse contenue dans un pointeur, l'accès peut se faire aussi bien en lecture qu'en écriture. Pour cela, il suffit de faire précéder le nom du pointeur du caractère `*`. Le programme suivant déclare une variable `varB` de type entier et un pointeur `pB` sur un entier.

```
#include <stdio.h>

int main() {
 int varB;
 int *pB;

 pB=&varB;

 varB=5;
 printf("1.1 : varB = %d\n",varB);
 printf("1.2 : *pB = %d\n",*pB);
 *pB=7;
 printf("2.1 : varB = %d\n",varB);
 printf("2.2 : *pB = %d\n",*pB);
 return 0;
}
```

Voici l'exécution du programme précédent qui montre que `varB` et `*pB` identifient le même espace mémoire puisque les modifications de la valeur de `varB` modifient celle de `*pB` et réciproquement.

```
1.1 : varB = 5
1.2 : *pB = 5
2.1 : varB = 7
2.2 : *pB = 7
```

Si vous essayez d'accéder à une adresse non initialisée il ne va pas y avoir d'erreur lors de la compilation mais l'exécution ne va pas se passer correctement. Considérons le programme précédent dans lequel l'instruction `printf("0.0 : *pB = %d\n",*pB);` a été ajoutée avant l'initialisation de la variable `pB` par l'instruction `pB=&varB;`. Comme toute variable, la variable `pB` est stockée en mémoire sur un ou plusieurs octets. Puisqu'elle n'a pas été initialisée de façon explicite, sa valeur initiale correspond à la valeur qui se trouve dans les octets qu'elle occupe, vous ne pouvez faire aucune hypothèse sur cette valeur qui dans ce cas précis sera considérée comme étant une adresse. Soit le programme n'a pas le droit d'accéder à cette adresse et l'exécution s'interrompt et le message `Erreur de segmentation` ou `Bus error` s'affiche. Soit le programme a le droit d'accéder à cette adresse et interprète la valeur contenue à l'adresse comme étant un entier, l'exécution obtenue est alors similaire à la suivante avec n'importe quelle valeur possible pour le premier

affichage de *pB. Il se peut aussi qu'à cette adresse soit stockée une autre variable, dont on peut alors modifier la valeur sans s'en rendre compte. Un exemple d'un tel comportement est donné dans la section 4.2.5.

```
0.0 : *pB = -1125351077
1.1 : varB = 5
1.2 : *pB = 5
2.1 : varB = 7
2.2 : *pB = 7
```

4.1.3 Exemples d'utilisation

Une fonction ayant comme paramètre un pointeur (on parle alors de « passage par référence » ou « par adresse ») recupère l'adresse de stockage d'une variable. Elle a donc accès à l'information se trouvant à l'adresse contenue dans le pointeur, elle peut lire et modifier la valeur s'y trouvant même s'il ne s'agit pas d'une variable locale à la fonction. Voici quelques utilisations des paramètres de type pointeur.

Fonction `scanf`

La fonction `scanf` permet d'initialiser des variables dont l'adresse est passée en paramètre de la fonction. L'appel `scanf("%d",&c)` initialise les octets se trouvant à l'adresse passée en paramètre (ici l'adresse de la variable `c`) à la valeur entière saisie au clavier. Cet appel permet donc d'initialiser la variable `c`. Il faut bien sûr que `c` soit une variable entière préalablement déclarée.

Plusieurs valeurs de retour

Une fonction a au plus une valeur de retour, dont le type est déterminé par la signature de la fonction. Si on souhaite récupérer plusieurs résultats, il est possible de le faire par les paramètres en utilisant les pointeurs. La fonction a alors à sa disposition l'adresse d'une variable et peut directement modifier sa valeur. Considérons une fonction `operations` qui prend deux entiers en paramètres et doit retourner la somme et le produit des deux entiers. Voici une solution dans laquelle les deux résultats sont récupérés via des paramètres.

```
void operations(int nb1, int nb2, int* resAdd, int * resProd) {
   *resAdd =  nb1  +  nb2;
   *resProd =  nb1  *  nb2;
}
```

Le programme appelant cette fonction doit bien sûr contenir la déclaration des variables dont l'adresse sera passée en paramètre. Voici un exemple de programme principal utilisant la fonction `operations`.

```
int main() {
   int a,b;
   int add, prod;
   printf("Veuillez saisir un premier entier\n");
   scanf("%d",&a);
   printf("Veuillez saisir un deuxieme entier\n");
   scanf("%d",&b);
   operations(a,b,&add,&prod);
   printf("%d + %d = %d\n",a,b,add);
   printf("%d * %d = %d\n",a,b,prod);

   return 0;
}
```

Lors de l'exécution de ce programme, la fonction `operations` prend en paramètres effectifs la valeur de `a`, la valeur de `b`, l'adresse de la variable `add` et l'adresse de la variable `prod`. Les résultats calculés par la fonction `operations` seront écrits dans les octets où sont stockées les variables `prod` et `add`.

Effets de bord

L'exemple précédent utilise les pointeurs pour palier le fait qu'une fonction ne renvoie qu'un seul résultat d'un type donné. De façon plus générale, on peut utiliser des paramètres de type pointeur lorsqu'on souhaite qu'une fonction ait des effets de bords, c'est-à-dire qu'elle modifie l'environnement à partir duquel elle a été appelée ; ce qui est fait par les fonctions `operations` et `scanf`. C'est aussi le cas lorsqu'on utilise une fonction pour permuter la valeur de deux variables : après l'appel à la fonction les deux variables concernées doivent avoir échangé leur valeur. Voici le code de la fonction `permute` qui remplit cette fonction.

```
void permute(int * var1, int * var2) {
   int tmp;
   tmp = *var1;
   *var1  = *var2;
   *var2 = tmp;
}
```

Le programme appelant cette fonction doit bien sûr contenir la déclaration des variables dont l'adresse sera passée en paramètre. Voici un exemple de programme principal utilisant la fonction `permute`.

```
int main() {
   int n1 = 4, n2 = 7;

   printf("Avant permutation\n");
   printf("nombre1 = %d, nombre2 = %d\n",n1,n2);
   permute(&n1,&n2);
   printf("Apres permutation\n");
   printf("nombre1 = %d, nombre2 = %d\n",n1,n2);

   return 0;
}
```

Pointeurs sur des variables locales

Attention, une fonction ne peut pas retourner un pointeur sur une de ses variables locales. L'espace mémoire occupé par la variable locale n'est théoriquement plus accessible dès que la fonction est terminée. Dans les faits, l'accès à ces octets se fait sans problème mais il n'y a aucune garantie sur la valeur récupérée. Ces situations rendent l'exécution d'un programme tout à fait instable et ne sont pas souhaitables. Ces cas sont détectés lors de la compilation, le message `warning: function returns address of local variable` s'affiche. Il ne s'agit que d'un `avertissement` mais vous devez le considérer comme une erreur car l'espace mémoire où a été stockée la variable locale a pu être alloué à une autre variable et son contenu modifié. Un tel `avertissement` s'affiche lors de la compilation de la fonction suivante.

```
/* ATTENTION : CETTE FONCTION N'EST PAS CORRECTE */
int* addition_Probleme(int nb1, int nb2) {
   int res;

   res = nb1 + nb2;
   return &res;
}
```

Lorsqu'une fonction produit des effets de bords et que les variables qu'elle modifie sont passées par adresse, il faut bien faire attention à ce que les variables soient déclarées et pas seulement les adresses car dans ce cas l'adresse identifie un octet qui ne contient aucune variable déclarée. La déclaration du pointeur ne fait qu'allouer de l'espace mémoire pour stocker une adresse mais ne fait en aucun cas l'allocation de la mémoire nécessaire pour stocker la variable à laquelle le pointeur est supposé donner accès. Voici un programme principal d'appel de la fonction `operations` dans lequel les variables devant recevoir les résultats des opérations n'ont pas été déclarées, seules des adresses ont été déclarées.

```
/* ATTENTION : CE PROGRAMME PRINCIPAL N'EST PAS CORRECT */
int main() {
   int a,b;
   int *add, *prod;
   printf("Veuillez saisir un premier entier\n");
   scanf("%d",&a);
   printf("Veuillez saisir un deuxieme entier\n");
   scanf("%d",&b);
   operations(a,b,add,prod);

   printf("%d + %d = %d\n",a,b,*add);
   printf("%d * %d = %d\n",a,b,*prod);

   return 0;
}
```

L'exécution de ce programme peut se passer sans problème, afficher des résultats incohérents, ou s'interrompre après l'affichage du message `Erreur de segmentation` ou `Bus error`. Il est important de noter que le programme peut s'exécuter correctement une fois et poser problème une autre. La compilation de ce programme ne signale ni `avertissement` ni `erreur`, vous devez donc faire extrêmement attention à déclarer toutes les variables dont vous utiliserez l'adresse par la suite.

4.2 Les tableaux

4.2.1 Définition

Il est parfois nécessaire de déclarer un ensemble de variables. Si elles ont toutes le même type, il est possible de les regrouper dans un tableau. La déclaration d'un tableau nécessite de connaître sa taille, c'est-à-dire le nombre d'éléments qu'il contient, et le type de ces éléments. Un tableau représente un ensemble ordonné d'éléments. L'ordre est défini par la position dans le tableau. Les éléments d'un tableau peuvent être organisés sur plusieurs lignes et colonnes, on parle alors de tableaux à plusieurs dimensions. Un tableau à une dimension peut être vu comme un vecteur ; le nombre d'éléments stockés est égal à la taille du vecteur. Un tableau à deux dimensions peut être vu comme une matrice (une table à deux entrées), il faut préciser la taille de chacune des entrées (nombre de lignes et de colonnes) ; il y a (nombre de lignes * nombre de colonnes) éléments stockés dans la matrice. On peut déclarer des tableaux avec autant de dimensions que souhaitées, nous nous limiterons aux tableaux à une et à deux dimensions.

4.2.2 Syntaxe

Déclaration

La déclaration d'un tableau se fait en donnant le type des éléments contenus, le nom du tableau et sa taille.

– La déclaration d'un tableau à une dimension se fait de la façon suivante :

  `<type> <nom>[entier];`

  Ainsi un tableau de 10 entiers sera déclaré par l'instruction suivante où `tableau` est le nom de la variable déclarée :

  ```
  int tableau[10];
  ```

– La déclaration d'un tableau à deux dimensions se fait de la façon suivante :

  `<type> <nom>[entier1][entier2];`

  Ainsi une matrice d'entiers de 10 lignes et 5 colonnes sera déclarée par l'instruction suivante où `matrice` est le nom de la variable déclarée :

  ```
  int matrice[10][5];
  ```

Initialisation

Comme pour toute autre variable, il est possible d'initialiser un tableau lors de sa déclaration. La syntaxe est la suivante :

– pour un tableau à une dimension

`<type> <nom> [entier] = {`val_1`,`val_2`,...,`val_{entier}`};`

– pour un tableau à deux dimensions (on donne la valeur de chaque ligne)

`<type> <nom> [entier1][entier2] =`
`{{`$val_{1,1}$`,...,`$val_{1,entier2}$`},...,{`$val_{entier1,1}$`,...,`$val_{entier1,entier2}$`}};`

Il faut bien sûr que les valeurs soient du type des éléments du tableau. **Attention**, car si vous ne respectez pas cette règle il se peut que rien ne soit signalé à la compilation et à l'exécution. Ceci se produit si vous avez utilisé des types compatibles pour lesquels une conversion est faite automatiquement (réel vers entier, entier vers caractère par exemple). Dans ce cas, vous risquez de ne pas obtenir le résultat attendu lors de l'exécution.

Si vous donnez moins de valeurs que le tableau ne peut en contenir, les valeurs données serviront à initialiser les premiers éléments du tableau, les éléments suivants auront comme valeur initiale la valeur se trouvant dans les octets les stockant en mémoire. Vous ne pouvez faire aucune hypothèse sur ces valeurs.

Si vous donnez plus de valeurs que le tableau ne peut en contenir l'avertissement suivant sera affiché lors de la compilation.

```
warning: excess elements in array initializer
```

ou « éléments en excès dans l'initialisation de tableau » en français.

Lorsqu'un tableau à une dimension est initialisé à la déclaration, il est possible de ne pas donner sa taille, elle est égale au nombre de valeurs données. La déclaration suivante définit un tableau de 4 entiers :

```
int tab[] = {1,5,4,0};
```

Ce calcul implicite de la taille du tableau lors de la déclaration n'est pas possible pour les tableaux à plusieurs dimensions. Si vous compilez la déclaration suivante :

```
int mat[][]=\{\{1,2\},\{2,4\},\{7,2\}\};
```

Vous obtiendrez le message d'avertissement :

```
warning: braces around scalar initializerattention
```

(ou « accolades autour d'une initialisation de scalaire » en français) et lors de l'utilisation du tableau, le message d'erreur suivant s'affichera :

```
error: subscripted value is neither array nor pointer
```

(ou « la valeur indicée n'est ni un tableau ni un pointeur » en français). Ce message vous indique que la variable n'est pas identifiée comme étant un tableau (ni un pointeur), vous pouvez donc en conclure que la déclaration n'est pas correcte.

4.2.3 Tableaux et représentation mémoire

Une variable de type tableau est en fait un pointeur sur le premier élément du tableau (c'est-à-dire qu'elle contient l'adresse du premier élément du tableau). Les éléments d'un même tableau étant stockés dans des octets mémoire contigus, il est possible à partir de l'adresse du premier d'accéder à l'ensemble des éléments. Dans le cas d'un tableau à deux dimensions, les lignes sont stockées les unes après les autres.

Il est important de noter que les octets suivants la dernière case du tableau sont considérés comme libres et peuvent donc être utilisés par d'autres variables. Cela nécessite de la part du programmeur de prendre quelques précautions :

1. Une fois la déclaration effectuée, la dimension d'un tableau ne peut pas être modifiée. On peut cependant stocker dans le tableau moins de valeurs que la taille ne le permet. Il est donc d'usage de prévoir dès l'écriture du programme une taille de tableau assez grande pour stocker toutes les données dont on pourrait avoir besoin en cours d'exécution.

2. En mémoire, il n'y a aucun moyen de déterminer la fin du tableau (contrairement aux chaînes de caractères que nous verrons au paragraphe 4.2.5). C'est au programmeur de savoir quelle est la taille de son tableau et de vérifier à tout instant que son programme ne va pas écrire dans des octets au delà de la fin du tableau. Le compilateur C (contrairement à ce qui se fait dans d'autres langages) ne donnera jamais d'avertissement si vous « sortez » du tableau. Malheureusement, cette zone étant probablement utilisée par une autre variable, cela va provoquer des modifications inattendues dans l'état du programme et des bugs souvent difficiles à détecter.

   Un cas particulier est celui des tableaux à deux dimensions. En effet, la valeur de la case $tab_{i,M-1}$ est immédiatement suivie en mémoire par la valeur de la case $tab_{i+1,0}$. Le compilateur C ne sachant pas que la case $tab_{i,M}$ n'existe pas, il écrira dans $tab_{i+1,0}$ sans le moindre avertissement.

Si on creuse...

Note préliminaire : Ce paragraphe est à l'usage des étudiants avancés qui voudraient aller plus loin dans la représentation mémoire. Sa compréhension n'est pas nécessaire pour l'UE CINI.

En fait, la variable "pointeur" vers le premier élément du tableau dont nous parlions précedemment n'existe pas en mémoire. Une variable de type tableau se comporte comme un pointeur, mais ça n'en est pas un. Un examen du code assembleur produit par le compilateur montre que la variable de type tableau est simplement représentée comme une étiquette vers la zone des données, mais pas comme une donnée elle-même, contrairement à une variable pointeur qui, au delà de l'étiquette, possède une adresse mémoire. Le morceau de code suivant illustre cette propriété :

```
int a = 3;
int *t1 = &a;
int t2[] = {1,2,3};

printf("%p %p %p %p\n",&t1,t1,&t2,t2);
```

L'affichage produit est le suivant :

```
0xbf82f53c 0xbf82f540 0xbf82f530 0xbf82f530
```

Les deux première adresses correspondent à l'adresse de la variable de type pointeur `t1` et à l'adresse pointée par `t1` (en l'occurence, c'est l'adresse de la variable `a`). Elles sont bien différentes. Les deux dernières adresses correspondent à l'adresse de la variable `t2` (ce que nous avons appelé le pointeur vers le début du tableau ci-avant) et à celle du premier élément du tableau (`t2 = &(t2[0])`). Comme nous pouvons le constater, la valeur fournie pour l'adresse `t2` est identique à l'adresse du premier élément du tableau. En fait, il n'y a pas de « pointeur vers le tableau ». Le compilateur C a choisi de donner l'adresse du premier élément du tableau car il identifie le tableau à son étiquette en assembleur.

C'est bien sûr tout à fait différent lorsqu'on passe le tableau en paramètre d'une fonction. En effet, le paramètre est véritablement une variable de type pointeur (même lorsqu'il a été déclaré en utilisant le symbole des tableaux []) et la valeur qui lui est attribuée est l'adresse du premier élément du tableau. C'est ce qu'illustre le programme suivant :

```
#include <stdio.h>

void f1(int t[]) {
  printf("f1: %p %p\n", t, &t);
}

void f2(int *t) {
  printf("f2: %p %p\n", t, &t);
}

int main() {
  int tab[] = {1,2,3};
```

```
  printf("main: %p %p\n", tab, &tab);
  f1(tab);
  f2(tab);
  return 0;
}
```

qui produit l'affichage suivant :

```
main: 0xbf83ad48 0xbf83ad48
f1: 0xbf83ad48 0xbf83ad30
f2: 0xbf83ad48 0xbf83ad30
```

Dans les deux appels de fonctions, le paramètre `t` a bien une adresse mémoire.

Nous verrons dans le chapitre 5 que ce choix de représentation mémoire a un impact sur les enregistrements.

4.2.4 Accès aux éléments d'un tableau

Une des caractéristiques des tableaux est la possibilité de pouvoir accéder directement à un élément simplement en connaissant sa position dans le tableau. Cet accès peut se faire de deux façons, soit en utilisant la notation propre aux tableaux (les []), soit en utilisant le fait qu'un tableau est un pointeur. Comme pour les autre variables la notation est la même que l'accès se fasse en écriture ou en lecture.

- Pour accéder directement au i^{eme} élément d'un tableau nommé `tab` il suffit d'utiliser la notation suivante :

  ```
  tab[i-1]
  ```

  Attention, la numérotation des éléments commence obligatoirement à 0. L'élément `tab[i]` d'un tableau est appelé élement d'indice `i`.
  Dans le cas des tableaux à deux dimensions, il suffit de préciser la ligne et la colonne, pour accéder à l'élément se trouvant en i^{eme} ligne, j^{eme} colonne il suffit d'utiliser la notation suivante.

  ```
  tab[i-1][j-1]
  ```

- En utilisant le fait qu'un tableau est un pointeur l'accès au i^{eme} élément d'un tableau nommé `tab` se fait en utilisant la notation suivante :

  ```
  *(tab+i-1)
  ```

  Nous vous rappelons que `tab` contient l'adresse du premier élément du tableau, `tab+i-1` contient l'adresse du i^{eme} élément. Lorsqu'un pointeur est augmenté de 1, la valeur qu'il contient n'est pas simplement augmentée de 1, elle est augmentée de façon à contenir l'adresse de la variable suivante (c'est-à-dire d'autant d'octets que le type du tableau le nécessite : 4 pour le type **`int`**, *etc*). Dans notre cas, si `tab` est un tableau d'entiers, `tab+1` contiendra l'adresse à partir de laquelle est stocké le deuxième entier du tableau et ainsi de suite.
  Pour utiliser cette notation pour les tableaux à deux dimensions, il faut se les représenter comme un vecteur contenant d'abord les éléments de la première ligne, puis ceux de la deuxième, ... L'accès à l'élément se trouvant en i^{eme} ligne, j^{eme} colonne correspond donc à l'accès au (nombre_de_colonnes * (i-1) + (j-1))^{eme} élément, il se fera donc par la notation suivante :

  ```
  *(tab+nombre_de_colonnes*(i-1)+(j-1))
  ```

  Attention, comme nous l'avons évoqué précedemment, aucun contrôle n'est fait sur les indices lors de l'accès aux éléments d'un tableau. Si vous accédez à un indice hors du tableau (indice négatif ou supérieur à la taille déclarée du tableau (-1)), vous aurez les mêmes problèmes que ceux évoqués dans la sous-section ≪ Accès à une adresse ≫ de la section 4.1.2.

4.2.5 Les chaînes de caractères

Dans cet enseignement nous assimilons les chaînes de caractères à des tableaux de caractères se terminant pas le caractère `\0`. Pour stocker une chaîne de `n` caractères, il faut déclarer un tableau de `n+1` caractères.

Le dernier caractère est utilisé pour stocker le caractère \0 qui représente la fin de la chaîne. Vous pouvez initialiser une chaîne de caractères lors de la déclaration ou ultérieurement dans le programme.

Initialisation lors la déclaration

– Les instructions suivantes déclarent chacune une chaîne de 17 caractères (16 caractères significatifs + le caractère \0). La taille est déterminée par la valeur initiale. Le caractère \0 est automatiquement ajouté après le dernier caractère.

```
char chaine1[]="voici un exemple";
char *chaine2="voici un exemple";
```

– L'instruction suivante déclare une chaîne pouvant contenir 10 caractères significatifs plus le \0. La valeur initiale ne contenant que 4 caractères, le cinquième contient le \0.

```
char chaine[11]="mot";
```

Attention, si la valeur initiale contient plus de 11 caractères le \0 ne sera pas inséré en 11^{eme} position, les caractères supplémentaires seront écrits en mémoire, à la suite de la valeur de la variable, dans des octets qui ne sont pas alloués à la variable chaine, ils peuvent donc modifier la valeur d'autres variables. Cette erreur n'est détectée ni à la compilation ni à l'exécution dont le résultat n'est plus prévisible.

Initialisation lors d'une saisie

Si vous précisez sa taille, vous pouvez déclarer une chaîne de caractères sans l'initialiser. Vous devez ensuite lui donner une valeur avant de l'utiliser. Vous pouvez le faire en donnant une valeur à chaque élément du tableau, en n'oubliant pas de donner la valeur \0 au caractère suivant la dernière valeur significative. Vous pouvez aussi utiliser la fonction scanf en utilisant la spécification de conversion %s. Les instructions suivantes permettent de déclarer et d'initialiser une chaîne de 10 caractères significatifs.

```
char chaine[11];
scanf("%s",chaine);
```

Soulignons que chaine est déjà un pointeur, donc il ne faut pas rajouter le symbole & devant, qui permet d'obtenir l'adresse d'une variable (nous aurions alors l'adresse du pointeur vers le tableau)...

Attention, la fonction scanf ne fait pas de contrôle sur le nombre de caractères saisis. S'il y en a moins de 10, la caractère \0 inséré permet d'identifier la fin de chaîne. S'il y en a plus de 10, on a les mêmes problèmes que dans le cas précédent.Voici un exemple de programme et une exécution possible illustrant ces problèmes.

```
#include <stdio.h>

int main() {
   char mot1[5]="abcd";
   char mot2[10];

   printf("Saisissez une chaine de caracteres\n");
   scanf("%s",mot2);
   printf("mot1=%s\n",mot1);
   printf("mot2=%s\n",mot2);

   return 0;
}
```

Dans l'exécution suivante les caractères en trop saisis pour la variable mot2 ont "écrasé" la valeur de la variable mot1 (la valeur saisie par l'utilisateur est en italique). La variable mot2 contient 12 caractères significatifs alors qu'elle devait en contenir uniquement 9. Ceci est dû au fait que mot2 contient l'adresse du premier caractère et que la fin de la chaîne est déterminée par le caractère \0 et non par sa taille déclarée. Cette même raison explique aussi pourquoi la chaîne mot1 ne contient plus que 2 caractères significatifs. Le

troisième carcatère est le \0. Pour que la saisie de la valeur de mot2 modifie la valeur de mot1 il faut que les octets mémoire occupés par mot1 soient juste après ceux occupés par mot2.

```
Saisissez une chaine de caracteres
abcdefghijkl
mot1=kl
mot2=abcdefghijkl
```

Initialisation obligatoire lors de la déclaration

Les déclarations `char chaine1[];` et `char *chaine2;` ne font que déclarer un pointeur sur un caractère et n'entraînent pas l'allocation mémoire des octets nécessaires au stockage des caractères de la chaîne. Si vous déclarez des chaînes de caractères de cette façon, il faut les initialiser en même temps. L'exécution suivante n'entraîne aucun message à la compilation mais une erreur à l'exécution avec affichage du message `Erreur de segmentation` ou `Bus error`.

```
/* ATTENTION : CE PROGRAMME PRINCIPAL N'EST PAS CORRECT */
#include <stdio.h>

int main() {
   char* mot;

   printf("Saisissez une chaine de caracteres\n");
   scanf("%s",mot);

   return 0;
}
```

La compilation du programme suivant affiche l'erreur `array size missing in 'mot'`.

```
/* ATTENTION : CE PROGRAMME PRINCIPAL N'EST PAS CORRECT */
#include <stdio.h>

int main() {
   char mot[];

   printf("Saisissez une chaine de caracteres\n");
   scanf("%s",mot);

   return 0;
}
```

Copie de chaînes de caractères

Dans le programme C, vous ne manipulez qu'un pointeur sur une chaîne de caractère. Il n'est donc pas possible de recopier une chaîne dans une autre en utilisant simplement l'opération d'affectation :

```
char *mot1 = "toto", *mot2 = "titi";
mot1 = mot2;
```

Cela aurait simplement pour effet de faire pointer mot1 vers la même chaîne que mot2, comme l'illustre le programme suivant :

```
#include <stdio.h>

int main() {
  char mot1[5] = "toto", mot2[5];
```

```
  mot2=mot1;
  mot1[0] = 'm';

  /* Ce programme ecrit deux fois ``moto'' */
  printf("%s %s\n",mot1,mot2);

  return 0;
}
```

Dans ce programme, la modification sur `mot1` impacte `mot2`, car les deux variables pointent vers la même chaîne. Il aurait fallu utiliser la fonction `strcpy` de la bibliothèque *string* :

```
#include <stdio.h>
#include <string.h>

int main() {
  char mot1[5] = "toto", mot2[5];
  strcpy(mot2,mot1);
  mot1[0] = 'm';

  /* Ce programme ecrit ``moto toto'' */
  printf("%s %s\n",mot1,mot2);

  return 0;
}
```

4.2.6 Parcours d'un tableau

Il est assez fréquent de devoir effectuer le même traitement sur l'ensemble des éléments d'un tableau (saisie des valeurs, affichage du tableau, ...). Dans ce cas, il est nécessaire d'effectuer un parcours complet du tableau. Lorsqu'il s'agit d'un tableau, la fin du parcours est déterminée par la taille du tableau, lorsqu'il s'agit d'une chaîne de caractères elle est déterminée par le caractère `\0`.

Parcours d'un tableau à une dimension

Voici un programme contenant la structure de boucle de parcours d'un tableau.

```
#define TAILLE 20

int main() {
   int tab[TAILLE];
   int i;

   for (i=0; i<TAILLE;  i++) {
       // instructions a repeter pour chaque element du tableau
   }

   return 0;
}
```

Si on souhaite afficher les éléments d'un tableau, l'instruction à répéter peut être :

```
printf("tab[%d]=%d",i,tab[i]);
```

Si on souhaite saisir les éléments d'un tableau en utilisant l'accès direct, les instructions à répéter peuvent être :

```
printf("Saisie de l'element tab[%d]",i);
scanf("%d",&tab[i]);
```

Si on souhaite saisir les éléments d'un tableau en utilisant la notation « pointeur », les instructions à répéter peuvent être :

```
printf("Saisie de l'element tab[%d]",i);
scanf("%d",tab+i);
```

Dans certains cas, le parcours ne se fait pas sur le tableau entier mais tant qu'une valeur particulière n'est pas trouvée. Il faut quand même s'assurer qu'on ne « sort » pas du tableau. La boucle suivante permet de parcourir un tableau d'entiers jusqu'au premier 0 (ou jusqu'à la fin s'il n'y a pas de 0).

```
#define TAILLE 20

int main() {
   int tab[TAILLE];
   int i;

   i=0;
   while ((i < TAILLE) && (tab[i] != 0)) {
       // instructions a repeter pour chaque element du tableau
        i++;
   }

   return 0;
}
```

Comme signalé dans la section 2.4, cette boucle peut s'écrire avec une boucle `for` mais comme nous ne connaissons pas initialement le nombre de fois où la boucle va être exécutée nous utilisons une boucle `while`.

Soulignons que l'ordre des expressions booléennes dans la condition de la boucle **`while`** est très importante : si `i>=TAILLE`, la deuxième expression n'est pas évaluée, ce qui évite une erreur de segmentation malheureuse.

Parcours d'une chaîne de caractères

Le parcours d'une chaîne de caractères se fait de la même façon qu'un tableau à une dimension, à la différence que la condition d'arrêt n'est pas déterminée par la taille du tableau mais par le caractère de fin de chaîne `'\0'` (dont la valeur entière est 0). Voici un programme contenant la structure de boucle de parcours d'une chaîne de caractères.

```
#define TAILLE 20

int main() {
   char mot[TAILLE];
   int i;

   i=0;
   while (mot[i] != '\0') { /* ou simplement mot[i] != 0 */
       // instructions a repeter pour chaque element du tableau
        i++;
   }

   return 0;
}
```

Parcours d'un tableau à deux dimensions

Pour parcourir complètement un tableau à deux dimensions il faut parcourir ses deux dimensions, pour chaque ligne il faut parcourir toutes les colonnes. Ce parcours se fait donc avec deux boucles imbriquées. Voici un programme contenant la structure de boucle de parcours d'un tableau à deux dimensions.

```
#define TAILLE1 20
#define TAILLE2 10

int main() {
   int tab[TAILLE1][TAILLE2];
   int i,j;

   for (i=0; i<TAILLE1; i++) {
      for (j=0; j<TAILLE2; j++) {
         // instructions a repeter pour chaque element du tableau
      }
   }

   return 0;
}
```

Si on souhaite afficher les éléments d'un tableau l'instruction à répéter peut être :

```
printf("tab[%d,%d]=%d",i,j,tab[i][j]);
```

Si on souhaite saisir les éléments d'un tableau en utilisant l'accès direct, les instructions à répéter peuvent être :

```
printf("Saisie de l'element tab[%d,%d]",i,j);
scanf("%d",&tab[i][j]);
```

Si on souhaite saisir les éléments d'un tableau en utilisant la notation « pointeur », les instructions à répéter peuvent être :

```
printf("Saisie de l'element tab[%d,%d]",i,j);
scanf("%d",tab+TAILLE2*i+j);
```

4.2.7 Fonctions et tableaux

Un tableau peut être passé en paramètre d'une fonction. Il est important de garder à l'esprit que le paramètre passé à la fonction est un pointeur (l'adresse du premier élément du tableau). Donc toute modification faite sur le tableau dans la fonction est faite directement sur les éléments du tableau.

Attention, de la même façon qu'elle ne peut pas retourner un pointeur, une fonction ne peut pas retourner un tableau déclaré localement. Dans ce cas, la valeur de retour de la fonction serait l'adresse d'une variable locale.

Voici trois entêtes possibles de fonctions ayant un tableau d'entiers en paramètre.
- `void fonction_tab1(int tab[TAILLE])` où `TAILLE` est définie par une primitive `#define`,
- `void fonction_tab2(int tab[])`,
- `void fonction_tab3(int *tab)`.

La fonction `fonction_tab1` n'est correctement définie que pour des tableaux de `TAILLE` entiers alors que les déclarations des fonctions `fonction_tab2` et `fonction_tab3` sont équivalentes et définies pour des tableaux d'entiers de taille quelconque. Pour pouvoir effectuer correctement le parcours du tableau à l'intérieur de ces deux fonctions, il faut donc passer en paramètre la taille des tableaux. On obtient alors les déclarations suivantes :
- `void fonction_tab2(int tab[], int taille)`,
- `void fonction_tab3(int *tab, int taille)`.

Dans le cas où on ne souhaite pas accéder au tableau en entier mais uniquement à des éléments précis, on passe leur indice en paramètre. C'est ce qui se passe si on souhaite permuter deux éléments d'un tableau dont les indices ont été préalablement identifiés. Voici un exemple d'une telle fonction.

```
void permute_element_tab(int tab[], int ind1, int ind2) {
        int tmp;
        tmp=tab[ind1];
        tab[ind1]=tab[ind2];
        tab[ind2]=tmp;
}
```

Il faut cependant s'assurer **avant l'appel à la fonction** que les indices sont bien dans le tableau !

On peut aussi souhaiter ne travailler que sur une partie du tableau. Dans ce cas, il faut passer en paramètre la valeur du premier indice (si différent de 0) et celle du dernier indice de la partie qui nous intéresse. La recherche dichotomique présentée en section 6.2 illustre cette possibilité.

Attention, aucune vérification n'est faite sur la taille des tableaux passés en paramètre et sur la valeur des indices lors de l'accès à leurs éléments. Si vous ne faites pas attention, vous vous retrouverez avec les mêmes problèmes que ceux évoqués dans la sous-section « Accès à une adresse » de la section 4.1.2.

4.2.8 Fonctions récursives et tableaux

En général la récursivité sur les tableaux se fait en rappelant la fonction récursive sur un « sous-tableau ». Les appels récursifs s'arrêtent lorsqu'on atteint une valeur particulière ou plus généralement lorsque le tableau passé en paramètre ne posséde qu'un élément.

Considérons le cas simple du parcours d'un tableau (par exemple pour faire la somme des éléments). On peut soit :

- parcourir le tableau à partir du dernier élément, le « sous-tableau » est alors le même tableau mais avec une taille diminuée de 1. C'est le principe utilisé pour la fonction `somme_tab1` suivante qui calcule de façon récursive la somme des éléments d'un tableau.

  ```
  int somme_tab1(int tab[], int taille) {
    if (taille == 1)
      return(tab[0]);
    else
      return(tab[taille-1]+somme_tab1(tab,taille-1));
  }
  ```

- parcourir le tableau à partir du premier élément, le « sous-tableau » est alors le tableau commençant au deuxième élément avec une taille diminuée de 1. Nous vous rappelons qu'une variable de type tableau contient l'adresse du premier élément du tableau. Donc, pour passer en paramètre le tableau commençant au deuxième élément, il suffit de donner l'adresse du deuxième élément (celle du premier + 1). Ce principe est utilisé pour la fonction `somme_tab2` suivante qui calcule de façon récursive la somme des premiers éléments d'un tableau tant que la valeur 0 n'est pas rencontrée. A chaque appel récursif la fonction teste le premier élément du tableau (celui d'indice 0), c'est la modification du paramètre lors de l'appel récursif qui fait passer à l'élément suivant. Vous remarquerez que, même dans ce cas, il est nécessaire de passer la taille du tableau en paramètre, c'est indispensable pour pouvoir traiter correctement les tableaux ne contenant pas de 0.

  ```
  int somme_tab2(int tab[], int taille) {
          if (taille == 1) {
                  return(tab[0]);
          }
          else {
                  if (tab[0] == 0) {
                          return 0;
                  }
  ```

```
                else {
                        return(tab[0]+somme_tab2(tab+1,taille-1));
                }
        }
}
```

– Nous avons déjà vu que lors du parcours d'une chaîne de caractères, il n'est pas nécessaire de connaître sa taille car le caractère `\0` nous informe de la fin de la chaîne. Le même constat est vrai pour les fonctions récursives. La fonction suivante affiche une chaîne de caractères en commençant par la fin. Avant d'afficher le premier caractère, il faut avoir affiché la chaîne à partir du deuxième caractère (en commençant par la fin bien sûr), c'est pour cette raison que l'affichage se trouve après l'appel récursif.

```
void inverse(char *mot) {
        if (mot[0] != '\0') {
                        inverse(mot+1);
                        printf("%c",mot[0]);
        }
}
```

Chapitre 5

Les enregistrements

Note : les notions de ce chapitre sont l'objet de la semaine de TD N°11.

5.1 Principe général

Dans le cadre de l'UE CINI, nous verrons deux structures de données (qui sont à la base de la plupart des structures de données que vous rencontrerez dans la suite de vos études) : les tableaux (section 4.2) et les enregistrements.

Un enregistrement est un ensemble de variables hétérogènes (c'est-à-dire de types possiblement différents) accessibles par leur nom, regroupées au sein d'une même structure. Au contraire, les tableaux ne peuvent contenir que des valeurs homogènes (c'est-à-dire nécessairement toutes de même type) et ces données sont accessibles par leur position dans la structure.

5.1.1 Déclaration de type

Un type enregistrement se déclare de la manière suivante en C :

```
struct nom_struct {
  type1 nom1;
  type2 nom2;
  ...
};
```

Cette expression définit le type **`struct`** *`nom_struct`* comme un enregistrement de variables de type $type_1$, $type_2$, ..., référencées par leur noms nom_1, nom_2, ... Ces variables sont appelées *champs* de l'enregistrement.

Par exemple, nous pouvons définir le type `struct date` suivant :

```
struct date {
  int jour;
  char *mois;
  int annee;
};
```

c'est-à-dire qu'une `date` sera un enregistrement à trois champs : un entier (le champ `jour`), un pointeur vers une chaîne de caractères (le champ `mois`) et un entier (le champ `annee`).

5.1.2 Déclaration de variables

Une définition de type enregistrement est une instruction de déclaration. Elle peut donc être apparaître soit au début d'une fonction, soit de manière globale en dehors des fonctions (comme pour les déclarations de variables globales). Un type défini au niveau du programme peut être utilisé dans les signatures et dans les corps des fonctions. Par exemple :

```
void afficher_date (struct date d) {
  ...
}
int main() {
  struct date ma_date;
  ...
}
```

Comme le montre l'exemple ci-dessus, une variable de type enregistrement est déclarée comme une variable de type primitif. Il est aussi possible de définir le type enregistrement directement lors de la déclaration d'une variable de ce type. Par exemple :

```
struct date {
  int jour;
  char *mois;
  int annee;
} ma_date;
```

La variable `ma_date` est du type enregistrement **`struct`** `date` défini avant.

Enregistrements non nommés

Dans le cas très particulier où le type n'est utilisé que pour déclarer une variable et n'est pas utilisé dans le reste du programme, le nom du type peut être omis. Par exemple :

```
struct {
  int jour;
  char *mois;
  int annee;
} ma_date;
```

5.1.3 Accès aux données

Les données d'un enregistrement sont accessibles par les noms des champs de l'enregistrement. Si *var* est une variable de type **`struct`** *nom_struct* (défini par les champs nom_1, nom_2, etc), l'accès au champ nom_i se fait en utilisant la notation suivante :

`var.`nom_i

Par exemple, le champ `jour` de l'enregistrement `ma_date` est accessible par :

```
ma_date.jour
```

De plus, si `pvar` désigne un pointeur vers un enregistrement, il est possible d'utiliser indifféremment les deux notations suivantes :

`(*pvar).`nom_i ou `pvar->`nom_i

5.1.4 Initialisation de variables

Une variable de type enregistrement peut être initialisée de deux manières différentes :

– Au moment de la déclaration, en utilisant la notation suivante :

nom_struct nom_var = { $valeur_1$, $valeur_2$, ... };

dans laquelle $valeur_1$ est une expression de type $type_1$, $valeur_2$ est de type $type_2$, etc, exactement dans l'ordre dans lequel les éléments du type enregistrement ont été déclarés lors de la définition du type. Par exemple, dans le cas de la date :

```
struct date ma_date = { 12, "janvier", 2003 };
```

– En initialisant chacune des variables de l'enregistrement séparément (comme pour les éléments d'un tableau). Par exemple :

```
struct date ma_date;
ma_date.jour = 12;
ma_date.mois = "janvier";
ma_date.annee = 2003;
```

ou bien avec des pointeurs :

```
struct date ma_date, *pdate;
pdate = &ma_date;
pdate -> jour = 12;
pdate -> mois = "janvier";
pdate -> annee = 2003;
```

5.1.5 Enregistrements, tableaux et pointeurs

5.1.5.1 Enregistrements et tableaux

Il est possible de déclarer un type enregistrement dont un ou plusieurs champs sont de type tableau, mais la même contrainte de déclaration de taille s'impose que pour les tableaux : il n'est pas possible de déclarer un tableau de taille quelconque, sauf si cette taille est fixée par l'initialisation de la variable au moment de la déclaration. En effet, le compilateur doit savoir quelle taille l'enregistrement va prendre en mémoire.

Ainsi, ce morceau de code est correct car on donne la taille du tableau dans la définition du type enregistrement :

```
struct toto {
  int tab[10];
};
struct toto t1;
```

De même, ce morceau de code est correct car, bien que la taille du tableau ne soit pas donnée lors de la définition du type, la variable `t1` est initialisée lors de la déclaration avec un champ `tab` de taille 3 :

```
struct toto {
  int tab[];
};
struct toto t1 = {{1,2,3}};
```

Mais par contre, ce morceau de code provoque une erreur lors de la compilation :

```
/* ATTENTION : CE MORCEAU DE CODE N'EST PAS CORRECT */
struct toto {
  int tab[];
};
struct toto t1;
```

Le compilateur répondra :

```
erreur: membre flexible de tableau dans une structure vide par ailleurs
```

Cependant, il faut garder à l'esprit qu'il n'existe pas vraiment de pointeur vers le tableau : les données du tableau sont directement stockées dans l'enregistrement, entre le champ précédent et le champ suivant, et le champ de type tableau ne définit qu'une étiquette pour l'adresse du premier élément du tableau.

Comme nous allons le voir, cela a un impact non négligeable lorsque l'on recopie des enregistrements, et en particulier des enregistrements avec des chaînes de caractères.

5.1.5.2 Recopie d'enregistrements

Comme pour les types primitifs, il est possible de recopier la valeur d'une variable de type enregistrement dans une autre variable en utilisant le symbole d'affectation =.

Cependant, il est important de bien comprendre ce qu'il se passe lors de la recopie d'un enregistrement :

- Les valeurs des champs de types primitifs (comme le champ `jour` dans notre exemple) sont recopiées. Ainsi, le morceau de programme suivant produit l'affichage `1 2` :

```
struct {
  int x;
} v1 = {2}, v2;
v2 = v1;
v1.x = 1;
printf("%d %d\n",v1.x,v2.x);
```

- Les valeurs des champs de types tableaux sont aussi recopiées, aussi surprenant que cela puisse paraître, et le champ du nouveau tableau « pointe » correctement vers le nouveau tableau.
  Ainsi, le morceau de programme suivant produit l'affichage `[1 2] [3 2]` :

```
struct {
  int tab[2];
} v1 = {{1, 2}}, v2;
v2 = v1;
v2.tab[0] = 3;
printf("[%d %d] [%d %d]\n",
  v1.tab[0],v1.tab[1],v2.tab[0],v2.tab[1]);
```

  Le tableau de la première variable n'est pas modifié lorsqu'on modifie celui de la seconde.

- Les valeurs des champs de types pointeurs sont évidemment recopiés et, par conséquent, le champ d'origine et le nouveau champ pointent tous les deux vers une seule et même variable. Toute modification dans l'un se répercutera dans l'autre.
  Ainsi, le morceau de programme suivant produit l'affichage `2` :

```
int a = 1;
struct {
  int *p;
} v1 = {&a}, v2;
v2 = v1;
*(v2.p) = 2;
printf("%d\n",*(v1.p));
```

  La valeur pointée par le champ du premier enregistrement est bien la même que celle pointée dans le second.

En particulier, dans notre exemple, la recopie d'une date recopie intégralement le champ `mois` s'il est déclaré comme un tableau, mais ne fait que recopier le pointeur s'il est déclaré comme un pointeur.

Pour recopier une chaîne de caractère, il est préférable d'utiliser alors la fonction `strcpy` de la bibliothèque `string`, vue section 3.3.

5.2 Un exemple d'utilisation

Voici un programme complet qui permet à un utilisateur de saisir un ensemble de dates et de les afficher :

```
#include <stdio.h>
struct date {
  int jour;
  char *mois;
  int annee;
};
```

```
void affiche_date(struct date d) {
  printf("%d %s %d\n", d.jour, d.mois, d.annee);
}

void saisir_date(struct date *d) {
  int m;

  printf("Saisissez une date sous la forme JJ/MM/AAAA : ");
  scanf("%d/%d/%d", &(d->jour), &m, &(d->annee));
  switch (m) {
    case 1: d->mois = "janvier"; break;
    case 2: d->mois = "fevrier"; break;
    ...
    case 12: d->mois = "decembre"; break;
    default: d->mois = "???";
  }
}

int main() {
  struct date ma_date;

  saisir_date(&ma_date);
  affiche_date(ma_date);
  return 0;
}
```

5.3 Quelques remarques pour finir

Pointeurs dans les enregistrements

Dans l'exemple précédent, nous utilisons un pointeur vers une chaîne de caractères pour le mois dans notre enregistrement. Cela ne fait évidemment aucune différence : il faut comme toujours faire attention à ce que la valeur pointée ne soit pas modifiée par une autre partie du programme, sans quoi le contenu de notre enregistrement serait modifié sans préavis.

Fonctions et enregistrements

Les enregistrements sont passés par valeur dans les fonctions qui les utilisent, contrairement aux tableaux qui sont passés par référence. Cela signifie que les modifications locales à la fonctions ne sont pas répercutées en dehors. Cela n'a rien d'original, mais c'est aussi une source d'erreur fréquente dans les programmes des étudiants.

Le polycopié de cours en présentiel contient plusieurs exemples d'utilisation de fonctions avec des enregistrements.

Chapitre 6

Algorithmes classiques

Note : les notions de ce chapitre sont reliées aux semaines N°6 à 10 du polycopié de TD et TP.

Nous allons reprendre ici en détail un certain nombre d'algorithmes classiques vu en cours.

6.1 Produit de matrices

D'un point de vue mathématique, le produit de deux matrices A de dimension $n \times m$ et B de dimensions $m \times p$ est un tableau C de dimensions $n \times p$ tel que

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \times B_{k,j}.$$

Nous écrivons tout d'abord un programme qui déclare les dimensions `n`, `m` et `p` comme constantes et dont la fonction `main` contient à la fois l'initialisation de deux matrices `A` et `B`, le calcul de leur produit `C` et l'affichage de la matrice `C`.

```
#include <stdio.h>

/* Dimensions */
#define n 5
#define m 3
#define p 4

int main() {
  /* Initialisation des matrices A et B */
  int A[n][m]={{1,3,7}, {5,2,6}, {3,1,4},
               {2,1,6}, {5,1,9}};
  int B[m][p]={{3,1,7,8}, {2,5,5,2}, {9,4,3,1}};
  /* Déclaration de la matrice résultat C */
  int C[n][p];
  /* Déclaration des trois indices de la formule */
  int i,j,k;

  /* Calcul de C */
  for (i=0;i<n;i=i+1) {
    for (j=0;j<p;j=j+1) {
      /* Calcul de C[i][j] */
      C[i][j]=0;
      for (k=0;k<m;k=k+1) {
        C[i][j]=C[i][j]+A[i][k]*B[k][j];
      }
    }
```

```
  }

  /* Affichage de C */
  for (i=0;i<n;i=i+1) {
    for (j=0;j<p;j=j+1) {
      printf("%d\t",C[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Dans ce programme, nous avons trois boucles imbriquées : la première (`i`)parcourt les `n` valeurs de la pemière dimension de `C`, la deuxième (`j`) parcourt les `p` valeurs de la deuxième dimension et la troisième permet de calculer la caleur de `C[i][j]` en faisant la somme des `A[i,k]*B[k,j]`, pour `k` de `0` à `m`.

Nous allons voir à présent une version où nous supposons toujours que les dimensions des matrices A et B sont connues à l'avance, mais le calcul du produit de ces matrices s'effectue dans une fonction. La matrice C doit être le résultat de la fonction. Un tableau ne peut être le résultat d'une fonction en C que s'il est passé en paramètre supplémentaire à la fonction. Le programme appelant aura réservé la place nécessaire pour ce tableau et la fonction ira y placer les valeurs souhaitées.

```
#include <stdio.h>

/* Dimensions */
#define n 5
#define m 3
#define p 4

/* C = AxB */
void prod (int A[n][m], int B[m][p], int C[n][p]) {
  int i,j,k;
  for (i=0;i<n;i=i+1) {
    for (j=0;j<p;j=j+1) {
      /* Calcul de C[i][j] */
      C[i][j]=0;
      for (k=0;k<m;k=k+1) {
        C[i][j]=C[i][j]+A[i][k]*B[k][j];
      }
    }
  }
}

int main() {
  /* Initialisation des matrices A et B */
  int A[n][m]={{1,3,7}, {5,2,6}, {3,1,4},
               {2,1,6}, {5,1,9}};
  int B[m][p]={{3,1,7,8}, {2,5,5,2}, {9,4,3,1}};
  /* Déclaration de la matrice résultat C */
  int C[n][p];
  /* Déclaration des deux indices de parcours de C pour son affichage */
  int i,j;

  /* Calcul de C */
  prod(A,B,C);
```

```
  /* Affichage de C */
  for (i=0;i<n;i=i+1) {
    for (j=0;j<p;j=j+1) {
      printf("%d\t",C[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Écrire une version où les dimensions sont passées en paramètres à la fonction `prod` conduit à utiliser des doubles pointeurs sur les tableaux et de l'allocation dynamique, ce qui dépasse largement le cadre de ce cours.

6.2 Recherche dichotomique

Lorsqu'on cherche un élément dans un tableau, si le tableau n'est pas trié on est condamné à parcourir toutes les cases du tableau pour s'assurer que l'élément est absent du tableau puisqu'il n'y a aucun emplacement naturel à un élément.

En revanche lorsque le tableau est trié selon un ordre total, ce qui est le cas si l'élément est d'un type entier de C, il y a un emplacement naturel pour chaque élément puisqu'on sait le comparer à n'importe quel élément du tableau et dire s'il doit se trouver avant ou après *s'il est présent dans le tableau*.

La recherche dichotomique d'un élément dans un tableau a pour but de tirer parti optimalement du fait que le tableau est trié. On compare l'élément recherché avec l'élément situé au milieu du tableau et si on n'est pas tombé pile dessus, on sait qu'il suffit de chercher dans la moitié gauche du tableau si l'élément est plus petit que celui du milieu et dans la moitié droite sinon.

Après il faut bien évidemment peaufiner les détails :

- on se retrouve rapidement à travailler sur des tranches de tableau qui ne touchent la plupart du temps aucun des deux bords du tableau initial, qu'on va donc caractériser par l'indice de début et l'indice de fin. Pour un tableau de longueur n, on commencera bien évidemment avec l'intervalle $0..n-1$, puis on se rabattra sur l'intervalle $0..m-1$ ou $m+1..n-1$ si l'élément n'est pas exactement au milieu m, et ainsi de suite ; on a donc naturellement 4 arguments à notre fonction : l'élément recherché, le tableau, les indices de début et de fin du sous-tableau
- dans le cas où il y a un nombre impair d'éléments, il n'y a pas d'ambiguïté en ce qui concerne la position du milieu, il y a un unique élément qui a le même nombre d'éléments à sa gauche et à sa droite ; en revanche il faut choisir quel est l'indice du milieu dans le cas où il y a un nombre pair d'éléments, il y aura alors une différence d'un (dans un sens ou dans l'autre) entre le nombre d'éléments à sa gauche et ceux à sa droite. On prend par exemple systématiquement l'arrondi inférieur de la moitié

$$m = \left\lfloor \frac{\text{debut} + \text{fin}}{2} \right\rfloor$$

- il faut déterminer les cas d'arrêt de l'algorithme :
  - si le sous-tableau est vide, on ne peut y trouver celui qu'on cherche
  - si l'élément recherché est justement celui situé au milieu du tableau, on peut immédiatement dire que l'élément recherché est présent.
- enfin il faut décider de ce que doit être le résultat. Plutôt que de se contenter d'indiquer si oui ou non l'élément est présent, ce qui dans la plupart des cas sera suivi d'un travail équivalent pour trouver la position de l'élément dans le tableau lorsqu'il est présent, on choisit de rendre directement la position dans le tableau, c'est-à-dire un entier compris entre 0 et $N-1$ si l'on manipule des tableaux de longueur N. On sait évidemment quelle position rendre quand l'élément est présent, mais que rendre quand il est absent ? puisqu'une position est un entier compris entre 0 et $N-1$, tous les autres entiers sont susceptibles de fournir un nombre *identifiable* correspondant à une situation particulière, traditionnellement on utilise -1.

Voici une version récursive que nous ne détaillerons pas beaucoup plus car l'explication ci-dessus y conduit naturellement d'une part et d'autre part le sujet est traité *in extenso* dans l'exercice 2 du TP de la semaine 10.

```
#include <stdio.h>
#define N 10

int pos_tab(int tab[], int e, int debut, int fin) {
  int m;

  /* cas de base: tableau vide, e non trouvé */
  if (fin < debut) { return -1; } else {
    m=(debut+fin)/2;
    /* cas de base: e trouvé */
    if (e==tab[m]) { return m; } else {
      /* cas récursif */
      if (e<tab[m]) {
        return pos_tab(tab, e, debut, m-1);
      } else {
        return pos_tab(tab, e, m+1, fin);
      }
    }
  }
}

void test (int t[], int n, int e) {
  int pos=pos_tab (t, e, 0, n-1);
  if (pos < 0) {
    printf("La valeur %d ne se trouve pas dans le tableau.\n", e);
  } else {
    printf("La valeur %d se trouve à la position %d.\n", e, pos);
  }
}

int main () {
  int tab[N]={3, 4, 15, 16, 18, 76, 89, 90, 92, 234};
  test (tab, N, 0);
  test (tab, N, 3);
  test (tab, N, 17);
  test (tab, N, 92);
  test (tab, N, 234);
  test (tab, N, 239);
  return 0;
}
```

Quant à la version itérative, l'évolution de la tranche du tableau sur laquelle on travaille va se faire dans une boucle. Nous avons dit précédemment que nous continuons tant que le tableau n'est pas vide et qu'on n'a pas trouvé l'élément. On identifie donc une boucle **`while`**.

Lorsqu'on trouve l'élément, on sort de la fonction par **`return`** donc on peut se contenter de calculer m à l'intérieur de la boucle et d'y tester si `e` est l'élément milieu `tab[m]` plutôt que calculer `m` prématurément à l'extérieur de la boucle ce qui n'est pas nécessaire si le tableau est vide.

On va donc se servir d'une seule condition `fin-debut>=0` qui nous garantit que le tableau n'est pas vide et lorsqu'on sort de la boucle le sous-tableau est nécessairement vide, l'élément est absent du tableau et on renvoie donc `-1`.

Voici donc le code de la version itérative.

```
#include <stdio.h>
```

```
#define N 10

int pos_tab (int t[], int n, int e) {
  int debut=0, fin=n-1, m;

  while (fin-debut>=0) {
    m=(debut+fin)/2;
    if (e==t[m]) { return (m); }
    if (e<t[m]) { fin=m-1; } else { debut=m+1; }
  }
  return (-1);
}

void test (int t[], int n, int c) {
  int pos=pos_tab (t, n, c);
  if (pos < 0) {
    printf("La valeur %d ne se trouve pas dans le tableau.\n", c);
  } else {
    printf("La valeur %d se trouve à la position %d.\n", c, pos);
  }
}

int main () {
  int tab[N]={3, 4, 15, 16, 18, 76, 89, 90, 92, 234};
  test (tab, N, 0);
  test (tab, N, 3);
  test (tab, N, 17);
  test (tab, N, 92);
  test (tab, N, 234);
  test (tab, N, 239);
  return 0;
}
```

6.3 Tris

La version itérative du tri par sélection du minimum est décrite en détail dans le polycopié du cours en présentiel que vous pouvez trouver sur le site de l'UE. En voici le code complet :

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_LONGUEUR 40
#define MAX_VALEUR 100

void echanger (int tab[], int i, int j) {
  int tmp=tab[i];
  tab[i] = tab[j];
  tab[j] = tmp;
}

int indice_min_fin_tableau (int tab[], int taille, int debut) {
  int imin=debut, i;

  for (i=debut+1; i < taille; i++) {
```

```
    if (tab[i] < tab[imin]) { imin = i; }
  }
  return imin;
}

void tri_selection (int tab[], int taille) {
  int i, j;

  for (i=0;i<taille;i++) {
    j = indice_min_fin_tableau (tab, taille, i);
    if (j > i) { echanger (tab, i, j); }
  }
}

void print_tableau (int tab[], int taille) {
  int i;

  for (i=0;i<taille;i++) {
    if (i % 20 == 19)
      { printf ("%2d\n", tab[i]); }
    else { printf ("%2d ", tab[i]); }
  }
}

int main () {
  int tab[MAX_LONGUEUR], i;

  /* Remplissage du tableau */
  for (i=0;i<MAX_LONGUEUR;i++) { tab[i] = random () % MAX_VALEUR; }
  /* Tri */
  printf ("Avant le tri\n");
  print_tableau (tab, MAX_LONGUEUR);
  tri_selection (tab,MAX_LONGUEUR);
  printf ("Apres le tri\n");
  print_tableau (tab,MAX_LONGUEUR);
  return 0;
}
```

Autant avec une boucle il est naturel d'aller placer le plus petit élément dans la première case du tableau et ainsi de suite, autant avec une fonction récursive il est naturel que la longueur du (sous)-tableau décroisse et pour éviter d'avoir à traîner avec soi l'indice de début de sous-tableau, on va plutôt aller placer le plus grand élément dans la dernière case du tableau ce qui permettra de travailler toujours sur un sous-tableau qui commence à l'indice 0 et dont la longueur décroît.

La version récursive par sélection du maximum est détaillée dans l'exercice 3 du TP de la semaine 10. En voici le listing complet :

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_LONGUEUR 40
#define MAX_VALEUR 100

void echanger (int tab[], int i, int j) {
  int tmp=tab[i];
  tab[i] = tab[j];
  tab[j] = tmp;
```

```
}

int indice_max_fin_tableau (int tab[], int n) {
  int imax=n, i;

  for (i=n-1; i >= 0; i--) {
    if (tab[i] > tab[imax]) { imax = i; }
  }
  return imax;
}

void tri_selection (int tab[], int n) {
  int imax;

  if (n > 0) {
    imax = indice_max_fin_tableau (tab, n);
    if (imax < n) { echanger (tab, n, imax); }
    tri_selection (tab, n-1);
  }
}

void print_tableau (int tab[], int taille) {
  int i;

  for (i=0;i<taille;i++) {
    if (i % 20 == 19)
      { printf ("%2d\n", tab[i]); }
    else { printf ("%2d ", tab[i]); }
  }
}

int main () {
  int tab[MAX_LONGUEUR], i;

  /* Remplissage du tableau */
  for (i=0;i<MAX_LONGUEUR;i++) { tab[i] = random () % MAX_VALEUR; }
  /* Tri */
  printf ("Avant le tri\n");
  print_tableau (tab, MAX_LONGUEUR);
  tri_selection (tab,MAX_LONGUEUR-1);
  printf ("Apres le tri\n");
  print_tableau (tab,MAX_LONGUEUR);
  return 0;
}
```

6.4 Un exemple très complet : le pivot de Gauß

6.4.1 Systèmes linéaires

Commençons par un exemple issu d'un livre de mathématiques de seconde : *Dans une classe, la taille moyenne des filles est de 1,66m et celle des garçons de 1,74m. La taille moyenne des 32 élèves de la classe est 1,68m. Trouver le nombre de filles et le nombre de garçons de cette classe.*

Notons x_1 le nombre de filles et x_2 le nombre de garçons, alors cet exercice peut se modéliser par le système

d'équations suivant :

$$\begin{cases} x_1 + x_2 = 32 \\ 1,66\,x_1 + 1,74\,x_2 = 32 \times 1,68 = 53,76 \end{cases}$$

dont l'unique solution est $x_1 = 24$ et $x_2 = 8$, c'est-à-dire qu'il y a 24 filles et 8 garçons dans cette classe.

L'objectif de cette section est d'écrire un programme C qui trouve les solutions d'un système linéaire en appliquant un algorithme fondé sur le pivot de Gauß que nous allons détailler dans ce qui suit.

Quelques définitions pour préciser les choses :

– Un *système linéaire* S de n équations à n inconnues $x_1, \ldots, x_n$ est un système de la forme :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \quad (L_1) \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \quad (L_2) \\ \qquad\qquad \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \quad (L_n) \end{cases}$$

noté $A\,x = b$ où A est une matrice carrée de dimension $n \times n$, x et b sont des vecteurs de dimension n. Dans le cas général, le nombre d'équations diffère du nombre d'inconnues, mais nous nous restreindrons ici au cas simple d'un système carré.

– Une *solution d'un système linéaire* S de dimension n est un vecteur $x = (x_1, \ldots, x_n)$ où chaque x_i est un nombre réel et x vérifie chacune des n équations (lignes) dans S.

6.4.2 Algorithme du pivot de Gauß

Principe

1. Étant donné un système linéaire S, il est possible d'appliquer les opérations suivantes sans modifier l'ensemble des solutions du système :
   – permutation de deux lignes : $L_i \leftrightarrow L_j$
   – multiplication d'une ligne par un scalaire non nul : $L_i := \lambda * L_i$
   – addition à une ligne L_i du produit d'une ligne L_j par un scalaire λ : $L_i := L_i + \lambda * L_j$
2. La méthode du pivot de Gauß est fondée sur l'idée qu'on peut simplifier un système S en appliquant systématiquement les opérations précédentes pour obtenir un système S' équivalent où chaque ligne i ($i \leq n$) ne contient que les variables x_i jusqu'à x_n, ce qu'on appelle un *système trinagulaire supérieur*.
3. On dispose d'un algorithme simple pour résoudre un système triangulaire supérieur (la méthode de remontée).

La résolution d'un système linéaire va donc consister à transformer le système en un système triangulaire supérieur équivalent par l'algorithme du pivot de Gauß puis à résoudre ce système triangulaire supérieur par la méthode de la remontée.

Nous allons détailler à présent ces deux algorithmes.

6.4.3 Méthode du pivot de Gauß

Étape 1 pour annuler le coefficient de x_1 pour chaque équation après la première, on ajoute à chaque ligne $L_i^{(1)} = L_i$ ($i > 1$) le produit $\lambda_i^{(1)} \times L_1^{(1)}$ avec $\lambda_i^{(1)} = -a_{i1}^{(1)}/a_{11}^{(1)}$ ($a_{11}^{(1)} \neq 0$ est appelé *le pivot* pour l'étape 1) et on obtient un nouveau système :

$$\begin{cases} a_{11}^{(2)}x_1 + & a_{12}^{(2)}x_2 + \ldots a_{1n}^{(2)}x_n = b_1^{(2)} \quad (L_1^{(2)}) \\ & a_{22}^{(2)}x_2 + \ldots a_{2n}^{(2)}x_n = b_2^{(2)} \quad (L_2^{(2)}) \\ & \vdots \\ & a_{p2}^{(2)}x_2 + \ldots a_{pn}^{(2)}x_n = b_p^{(2)} \quad (L_p^{(2)}) \end{cases}$$

Étape k de la même façon, pour annuler le coefficient de x_k pour chaque équation après la k^{e}, on ajoute à chaque ligne $L_i^{(k)}$ $(i > k)$ le produit $\lambda_k^{(k)} \times L_k^{(k)}$ avec $\lambda_k^{(k)} = -a_{ik}^{(k)}/a_{kk}^{(k)}$ ($a_{kk}^{(k)} \neq 0$ est le pivot pour l'étape k) et il en résulte le nouveau système

$$\left\{\begin{array}{rrl}
a_{11}^{(k+1)}x_1 + a_{12}^{(k+1)}x_2 + \ldots + a_{1k}^{(k+1)}x_k & +a_{1,k+1}^{(k+1)}x_{k+1} + \ldots + a_{1n}^{(k+1)}x_n = b_1^{(k+1)} & (L_1^{(k+1)}) \\
a_{22}^{(k+1)}x_2 + \ldots + a_{2k}^{(k+1)}x_k & +a_{2,k+1}^{(k+1)}x_{k+1} + \ldots + a_{2n}^{(k+1)}x_n = b_2^{(k+1)} & (L_2^{(k+1)}) \\
\ddots & & \\
a_{kk}^{(k+1)}x_k & +a_{k,k+1}^{(k+1)}x_{k+1} + \ldots + a_{k,n}^{(k+1)}x_n = b_k^{(k+1)} & (L_2^{(k+1)}) \\
& a_{k+1,k+1}^{(k+1)}x_{k+1} + \ldots + a_{k+1,n}^{(k+1)}x_n = b_{k+1}^{(k+1)} & (L_2^{(k+1)}) \\
& a_{n,k+1}^{(k+1)}x_{k+1} + \ldots + a_{nn}^{(k+1)}x_n = b_n^{(k+1)} & (L_n^{(k+1)})
\end{array}\right.$$

Si S est un système de Cramer (le système est de rang n) on obtient après $n-1$ étapes un système :

$$\left\{\begin{array}{rrl}
a_{11}^{(n)}x_1 + & a_{12}^{(n)}x_2 + \ldots & a_{1n}^{(n)}x_n = b_1^{(n)} \quad (L_1^{(n)}) \\
& a_{22}^{(n)}x_2 + \ldots & a_{2n}^{(n)}x_n = b_2^{(n)} \quad (L_2^{(n)}) \\
& \ddots & \\
& & a_{nn}^{(n)}x_n = b_n^{(n)} \quad (L_n^{(n)})
\end{array}\right.$$

Ce système possède une *unique solution* qu'on peut trouver par l'algorithme de remontée que nous allons décrire à présent.

6.4.4 Résolution d'un système triangulaire supérieur par l'algorithme de remontée

Pour le système triangulaire supérieur

$$\left\{\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1,n-1}x_{n-1} + a_{1n}x_n &=& b_1 \quad (L_1) \\
a_{22}x_2 + \ldots + a_{2,n-1}x_{n-1} + a_{2n}x_n &=& b_2 \quad (L_2) \\
\vdots & & \\
a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n &=& b_{n-1} \quad (L_{n-1}) \\
a_{nn}x_n &=& b_n \quad (L_n)
\end{array}\right.$$

l'équation L_n nous fournit $x_n = b_n/a_{nn}$.

En injectant cette valeur dans L_{n-1}, on obtient une équation en $x_{n-1} = (b_{n-1} - a_{n-1,n}x_n)/a_{n-1,n-1}$, et ainsi de suite…

À l'étape $k+1$ de remontée, on a obtenu les valeurs de x_n jusqu'à x_{n-k+1}, on injecte toutes ces valeurs dans L_{n-k} et on en déduit x_{n-k} jusqu'à la n^{e}étape qui fournit la valeur de x_1.

6.4.5 Pivot de Gauß en C

Pour écrire le programme attendu, il faut tout d'abord répondre à plusieurs questions :

- Comment représente-t-on les systèmes linéaires d'un point de vue mathématique d'une part et en C d'autre part ?
- Quelles sont les sources d'erreurs de calcul possibles et comment y remédier ?

Représentation mathématique du système linéaire

En mathématiques, on représente la partie gauche sous forme d'une matrice carrée A, c'est-à-dire d'un tableau à double entrée $A = (a_{i,j})$ et la partie droite sous forme d'un vecteur b, c'est-à-dire d'un tableau $b = (b_i)$.

L'exemple est représenté par la résolution du système linéaire $Ax = b$ avec la matrice :

$$A = \begin{pmatrix} 1 & 1 \\ 1.66 & 1.74 \end{pmatrix}$$

et le vecteur :

$$b = \begin{pmatrix} 32 \\ 53.76 \end{pmatrix}$$

La résolution produit le vecteur :

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 24 \\ 8 \end{pmatrix}$$

Représentation informatique d'un système linéaire

Nous allons considérer deux solutions :

1. coller à cette représentation mathématique, en utilisant deux tableaux `A[n][n]` et `b[n]`, ce qui nous donne sur notre exemple :

```
double A[N][N]={
  {1.0, 1.0},
  {1.66, 1.74}
};
double b[N]={
  32.0,
  53.76
};
```

2. considérer que le second membre se comporte tout au long de l'algorithme du pivot comme les coefficients de la matrice et utiliser un tableau de n lignes et $n+1$ colonnes où l'on intègre le second membre comme colonne $n+1$, colonne qui retrouve toute sa particularité lors de l'algorithme de remontée, ce qui donne sur notre exemple :

```
double tab[N][N+1]={
  {1.0, 1.0, 32.0},
  {1.66, 1.74, 53.76}
};
```

Quels sont les avantages et inconvénients de ces deux solutions :

- La seconde solution a l'avantage d'éviter à chaque étape de l'algorithme du pivot de Gauß de prévoir une instruction spéciale pour le second membre.
- Mais la première solution a l'avantage de mieux s'adapter au fait de résoudre un système $Ax = b$ pour différentes valeurs de b sans effectuer de multiples fois les mêmes opérations sur la matrice A, ce qui est fréquent (penser au calcul de l'inverse par exemple)

Nous adopterons donc cette seconde solution.

Sources d'erreurs possibles dans le calcul

Lors de l'étape k on ajoute à chaque ligne $L_i^{(k)}$ $(i > k)$ le produit $\lambda_k^{(k)} \times L_k^{(k)}$ avec $\lambda_k^{(k)} = -a_{ik}^{(k)}/a_{kk}^{(k)}$. On a donc supposé hardiment que $a_{kk}^{(k)} \neq 0$. Malheureusement c'est effectivement rarement le cas. On procède de la manière suivante :

- On vérifie que $a_{kk} \neq 0$ à chaque étape k.
- Si $a_{kk}^{(k)} = 0$, on permute la ligne k avec une ligne $j > k$ où $a_{jk} \neq 0$.
- Si une telle ligne n'existe pas, on arrête le programme (il peut exister aucune, une solution, ou plusieurs solutions...). En tout cas la matrice A n'est pas inversible !

Stratégie de pivotage

Pour choisir la ligne avec laquelle on va échanger la ligne k, on peut procéder de diverses façons :

- prendre la première ligne d'indice $j > k$ telle que $a_{jk}^{(k)} \not= 0$
- prendre j tel que $|a_{jk}^{(k)}| = max|a_{\ell k}^{(k)}|, k \leq \ell \leq n$, c'est-à-dire prendre le pivot le plus grand possible en valeur absolue dans la portion de colonne concernée

– prendre le pivot le plus grand possible en valeur absolue dans la portion de carrée concernée (de k, k à n, n)

La deuxième stratégie s'appelle la stratégie du *pivot partiel*, la troisième s'appelle la stratégie du *pivot total*.

Il a été démontré que la stratégie du *pivot partiel* donne les meilleurs résultats du point de vue de la complexité et du point de vue de la précision des résultats. Le coût est en $\frac{2n^3}{3}$ et il a été démontré qu'il ne pouvait exister d'algorithme pour résoudre des systèmes linéaires quelconques de dimension n de complexité plus faible que $\mathcal{O}(n^3)$. Le coût de l'algorithme de remontée est quant à lui en $\mathcal{O}(n^2)$ et donc globalement la résolution d'un système linéaire est en $\mathcal{O}(n^3)$.

Et enfin le code commenté

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 2

/* Résolution d'un système triangulaire Ax=b */
void remontee (double A[N][N], double b[N], double x[N]) {
  int i, k;
  double somme;

  for (i=N-1; i>=0; i--) {
    somme = 0;
    for (k=i+1; k<N; k++) {
      somme = somme+A[i][k]*x[k];
    }
    x[i] = (b[i]-somme)/A[i][i];
  }
}

/* Échange de lignes dans une matrice */
void echanger_lignes (double A[N][N], int i1, int i2) {
  int j;
  double tmp;

  for (j=0; j<N; j++) {
    tmp = A[i1][j];
    A[i1][j] = A[i2][j];
    A[i2][j] = tmp;
  }
}

/* Échange de valeurs dans un vecteur */
void echanger (double b[N], int i1, int i2) {
  double tmp;

  tmp = b[i1];
  b[i1] = b[i2];
  b[i2] = tmp;
}

/* Triangulation par la méthode du pivot de Gauss */
void pivot_de_Gauss (double A[N][N], double b[N]) {
  int i, j, k, i_pivot;
```

```
  double pivot, alpha;

  for (i=0; i<N; i++) {
    i_pivot = i;
    pivot = fabs (A[i][i]);
    for (k=i+1; k<N; k++) {
      if (fabs (A[k][i]) > pivot) {
        i_pivot = k;
        pivot = fabs (A[k][i]);
      }
    }
    /* On permute les lignes d'indice i et i_pivot */
    echanger_lignes (A, i, i_pivot);
    /* On échange aussi les second membres */
    echanger (b, i, i_pivot);
    /* Maintenant le pivot se trouve en A[i][i]
       et s'il est nul la matrice est non inversible */
    if (A[i][i] == 0) {
      printf ("pivot de Gauss: matrice non inversible");
      exit (1);
    } else {
      for (k=i+1; k<N; k++) {
        /* Coefficient alpha pour toute la ligne */
        alpha = A[k][i]/A[i][i];
        for (j=i+1; j<N; j++) {
          A[k][j] = A[k][j] - alpha*A[i][j];
        }
        A[k][i] = 0;
        b[k] = b[k] - alpha*b[i];
      }
    }
  }
}

void resolution (double A[N][N], double b[N], double x[N]) {

  pivot_de_Gauss (A, b);
  remontee (A, b, x);
}

int main () {
  int i, j;
  double A[N][N]={{1,1},{1.66,1.74}}, b[N]={32, 53.76}, x[N];

  /* Affichage du système */
  printf ("Systeme\n");
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      printf ("%fx%d", A[i][j], j);
      if (j < N-1) {
        printf ("+");
      } else {
        printf ("=%f\n", b[i]);
```

```
      }
    }
  }
  /* Résolution du système */
  resolution (A, b, x);
  /* Affichage du résultat */
  printf("Solution\n");
  for (i=0; i<N; i++) {
    printf ("x[%d]=%f\n", i, x[i]);
  }
  return 0;
}
```

et le résultat de son exécution :

```
> ./pivot
Systeme
1.000000x0+1.000000x1=32.000000
1.660000x0+1.740000x1=53.760000
Solution
x[0]=24.000000
x[1]=8.000000
```

Index

`++`, 13
`--`, 13
`->`, 46
`#define`, 6
`#include`, 6
`afficher_ligne`, 28
`afficher_point`, 28
`break`, 19
`char*`, 10
`char`, 7
`continue`, 19
`creer_fenetre`, 28
`double`, 7
`do`, 17
`else`, 14
`exit`, 21
`float`, 7
`for`, 18
`graphics`, 28
`if`, 14
`int`, 7
`long`, 7
`main`, 24
`printf`, 25
`rand`, 27
`return`, 20
`scanf`, 25
`short`, 7
`srand`, 27
`stdio`, 25
`stdlib`, 27
`strcmp`, 28
`strcpy`, 28
`strlen`, 28
`struct`, 45
`switch`, 15
`time`, 27
`unsigned`, 7
`while`, 16

Aléatoire, 27

Bibliothèque graphique, 28
Boucles, 16

Chaînes de caractères, 10, 28, 41
Commentaires, 5
Constantes, 6

Dessins, 28

Fonction `main`, 24
Fonction récursive, 23

Graphiques, 28

Locale (variable), 22

Main, fonction, 24

Paramètres, 21
Portée, 22

Récursive (fonction), 23, 43

Signature de fonction, 20

Tirage aléatoire, 27
Transtypage, 9, 10
Types, 7

Variable locale, 22