Universite Paris Ouest Nanterre La défense
Licence MIA

# Programmation en langage C

Notes de Cours

Emmanuel Hyon

27 novembre 2009

**Résumé**

Ce document archive l'ensemble des notes du cours de langage C donné en première année de la licence MMIA.

# Table des matières

# Première partie

# Semestre 1

# Chapitre 1

# Généralités

## 1.1 Apprentissage de l'informatique et buts poursuivis par ce cours

### 1.1.1 But du cours et prérequis

**Buts du cours**

Les buts de ce cours sont multiples.
- Apprendre les principes généraux de programmation.
- Apprendre les rudiments de l'algorithmique. C'est à dire la manière dont on organise et conçoit un programme.
- Apprendre la syntaxe d'un langage de programmation.

**Besoin de Rigueur**

L'apprentissage de l'informatique ne nécessite pas d'être extrèmement doué mais nécessite par contre de la rigueur. C'est pourquoi il faut : Écrire avec soin ces programmes : deux instructions à priori similaires (ou qui vous semblent identiques) ne donnent parfois pas les mêmes résultats. Oublier certains signes peut rendre un programme non fonctionnable.

**Besoin de travail sur machine**

Une nécessité de travailler sur la machine pour comprendre les phénomènes qui entrent en jeu. Cela permet de confronter la théorie à la pratique. De plus, la programmation ne s'apprend concrètement qu'avec de la pratique

**Environnement** L'environnement de travail, c'est à dire l'infrastructure qui vous est offerte, consiste en un compilateur et en environnement de développement.
Le compilateur : Compilateur GNU.
L'environnement de développement DEV C++.

Ces deux logiciels sont largement répandus et si vous souhaitez les installer chez vous, ceux-ci peuvent être trouvés à l'adresse

http ://www.01net.com/telecharger/windows/Programmation/langage/

**Justification du choix du Langage C**

**Largement répandu** Le langage C est un langage qui a été et reste largement utilisé par nombre de programmeurs même si actuellement il tend à être supplanté. Ainsi Windows (à partir de windows 95) a été écrit en C (les versions de maintenant en C++). En tout cas il s'avère comme nécessaire de le connaitre pour bon nombre d'informaticiens.

**Langage de haut niveau** Ce langage est aussi un langage évolué qui permet de rédiger les programmes sous une forme lisible humainement.

**Possibilité d'évolutions** Une fois le langage C connu et les principes d'algorithmique compris, cela donne les moyens, à chacun de pouvoir apprendre (avec un certain travail néanmoins) d'autres langages tels que

- Le C++
- Le JAVA

**Un bémol** Le langage C est ardu à apprendre et n'est pas forcément le plus pédagogique car :

- C'est un langage syntaxiquement complexe.
- C'est un langage dont le compilateur est très souple. Il peut donc laisser beaucoup d'erreurs.

Aussi le cadre de travail n'est pas aussi normé qu'avec des langages tels que PASCAL.

## 1.2 L'Ordinateur

**Définition 1.2.1** (Ordinateur : Définition "juridique")**.** Le terme ordinateur désigne un appareil électronique qui accepte les données sous un format numérique et les traite en vue d'un résultat.

Ainsi, un ordinateur traite (ou manipule) de l'information. Celle-ci peut être sous forme de

- Multimédia (musique, film, image)
- Données (Description d'un client d'une entreprise)
- Textes
- des pages Web.
- Chiffres (Comptes d'une entreprise, résultats d'expériences, résultat de calculs par exemple des calculs numériques sur des structures mécaniques).

**Exemple 1.2.2** (Un fichier de musique)**.** *Si on prend l'exemple d'un fichier MP3. Ce fichier se présente sous une forme numérique : une suite de* $0$ *et de* $1$ *qui s'affichent sous la forme suivante dans un éditeur de texte*[1]

[1] car les bits sont transformés suivant le code ASCII.

```
ríë= hàÄ<Xÿ<ÄÍ^EÃÎ^OT´Sé6~ND¶ª
```

*Le traitement de l'information consiste donc à faire jouer cette suite de* 0 *ou de* 1 *sous forme de sons. Ceci par l'intermédiaire du périphérique carte son.*

**Exemple 1.2.3** (Un texte)**.** *Un exemple similaire. On prend l'exemple d'un fichier de .doc. Ouvert sous le bloc note cela donne*

```
Ïà¡á;ÿÿÿ!"#$%&'E n t r y ÿ À Microsoft Word-Dokument
   MSWordDoc
  Word.Document.8
```

*Le but ici consiste alors à manipuler cette suite de* 0 *et* 1 *et de le considérer comme un texte par l'intermédiaire d'un Traitement de Texte afin de le modifier, de le mettre en forme.*

#### Structuration de l'information

La suite des opérations possibles pour un ordinateur est limitée. Il s'agit des 4 opérations de bases (addition, soustraction, multiplication, division de nombre) le plus généralement sur des nombres entiers, ainsi que la manipulations de ces nombres vis-à-vis de la mémoire.

Aussi à la vue des exemples précédents on peut se demander comment se passa la transformation d'un flux composé de 0 et 1 en de la musique ou en un fichier texte. Comment indiquer à l'ordinateur ce qu'il doit traiter comment lui notifier le type d'information ? Comment la structurer ?

Cette structuration est obtenue grâce au **Codage de l'information** :
- Codage des Nombres
- Codage des caractères

**Définition 1.2.4** (Codage des nombres)**.** Le codage est la transformation (inversible) d'une suite de 0 et de 1 en un type ou une structure plus complexe donnée.

## 1.3 Programme

Ici arrive la question

Qu'est ce qu'un programme ?

liée avec cette autre question

Qu'est ce qu'un logiciel ?

**Définition 1.3.1.** Un logiciel est un programme ou un ensemble de programmes qui visent à offrir à un utilisateur (ou à l'ordinateur) une fonctionnalité ou un service donné.

Par exemple :
- Traitement de texte

– Un utilitaire de configuration

**Définition 1.3.2** (Programme)**.** C'est la description d'une suite d'opérations élémentaires en vue de la réalisation au cours de son exécution d'une action donnée.

Ces opérations peuvent être
- une suite d'opérations élémentaires différentes
- ou la répétition un grand nombre de fois d'un ensemble d'opérations. On parle dans ce cas là *d'algorithme*.

Ces opérations manipulent des données structurées des : *structures de données*.

On voit donc apparaître les notions *d'exécution*, *d'instructions* et de *données*.

**Exemple 1.3.3** (Recherche d'un fichier par son nom dans un disque dur)**.** *On veut décrire la suite d'opérations qui permet à l'explorateur Windows de trouver un fichier de nom donné dans un disque dur. Ces opérations sont*

1. *Saisie du nom*
2. *Parcours de l'arborescence (récursivement)*
3. *...*

**Exemple 1.3.4** (Affichage d'une liste de participants à un we)**.** *On veut décrire la suite d'opérations qui permettent d'afficher ces données qui sont stockées dans un tableur.*
– *Chargement du fichier*
– *Selection du WE choisi*
– *Parcours du fichier et sélection des participants*

**Entrées d'un programme et manipulation de la mémoire vive**

A la lumière des exemples précédents on observe que les données à l'entrée du programme jouent un grand rôle.

Tout d'abord, retenons que la manipulation de données a lieu uniquement dans la mémoire vive. Dans le langage courant **charger** un fichier consiste à le transférer du disque dur vers la mémoire vive.

Au cours de leurs exécutions les programmes manipulent
– des donnés simples,
– des structures de données,
– des adresses mémoires sur ces données.

**Définition 1.3.5** (Structure de données)**.** Une structure de données peut être vue comme une manière d'organiser des données simples (nombre, caractères,...) dans un ensemble plus complexe en vue de faciliter le maniement de cet ensemble par le programme.

**Les fichiers** Sont une manière de stocker les données sur un périphérique mémoire. Il existe plusieurs type de fichiers. Citons, entre autre, *.doc, .htm, .xls, .dat, .txt* chaque extension "qualifiant" la manière dont le fichier est structuré .... Ces fichiers sont des fichiers de données.

Le *.exe* se distingue ce n'est pas un fichier de données mais un *binaire* ou programme executable qui décrit ce que doit faire la machine (le processeur) interpréter les ordres, manipuler les données,

**Définition 1.3.6** (Fichier exécutable)**.** Programme directement compréhensible par le processeur, contenant une suite des "commandes" qui entraînent des actions ou des opérations de la part du système.

En tant qu'adjectif, caractérise un fichier que l'on peut exécuter.

**Exemple 1.3.7** (Maniement d'un fichier texte par l'intermédiaire de WORD)**.** *En quoi consiste un traitement de texte. Le traitement de texte word consiste simplement en fait en un exécutable appelé word.exe. Il manipule des données structurées qui sont des textes. Ces données sont sauvegardées sous formes de fichiers (les .doc). Ouvrir un fichier .doc consiste alors à :*

- *Exécuter le programme word.exe.*
- *Lui donner des données en entrée : Le fichier* `.doc`.

**Exercice 1.3.8** (Suite d'opérations pour faire cuire un bifteck)**.** *On voudrait ici retranscrire sous la forme d'une suite d'instructions le fait tout simple en apparence de cuisiner de la viande à la poële. Quelles peuvent être les données d'entrée du programme ? Quelles sont les différentes opérations à mettre en oeuvre ?*

## 1.4 Le Langage de Programmation

Nous abordons maintenant la manière dont un humain peut écrire un programme[2]. Le fait d'écrire des programmes s'appelle la *Programmation*.

Le principe est simple : il s'agit de dire à l'ordinateur ce que vous voulez qu'il fasse. Pour ce faire il faut décrire à l'ordinateur, lui dire dans un langage qu'il comprend, la suite d'instructions et les données qu'il va manipuler. Dans ce but, on utilise un langage de programmation.

### 1.4.1 Généralités

**Définition 1.4.1** (Langage de programmation)**.** Un langage de programmation est un ensemble de symboles et de règles destiné à décrire l'ensemble des actions qu'un ordinateur doit exécuter. C'est une manière de donner des instructions à un ordinateur qui est, le plus souvent, lisible par un humain mais pas directement par l'ordinateur.

Le fait d'utiliser un langage de programmation se justifie par la difficulté d'exprimer la suite d'instructions dans le langage de l'ordinateur.

[2] sans se tirer une balle dans la tête.

**Définition 1.4.2** (Langage Machine)**.** C'est le "langage" de base compréhensible par un ordinateur (utilisé par le processeur), soit une suite de zéros et de uns.

Attention, le langage machine n'est pas compréhensible facilement par l'humain moyen. Aussi, il est plus pratique de trouver un langage intermédiaire, compréhensible par l'homme, qui sera ensuite transformé en langage machine pour être exploitable par le processeur.

**Langage de haut niveau de bas niveau** Plus le langage est compréhensible par l'homme, en d'autres termes plus il est évolué, plus on dira qu'il est de haut niveau. Ainsi si la programme est écrit dans un langage proche de l'ordinateur on dira qu'il est écrit dans un langage de bas niveau.

*Remarque* 1.4.3 (Portabilité). Un programme écrit dans un langage machine dépend étroitement du type de processeur utilisé car chaque type de processeur peut avoir son propre langage machine. Pour pouvoir l'utiliser sur une autre machine il faudrait alors le réécrire entièrement.

Ainsi, un langage informatique a donc plusieurs avantages :
- il est plus facilement compréhensible que le langage machine
- il permet une plus grande portabilité, c'est-à-dire une plus grande facilité d'adaptation sur des machines de types différents

**La compilation**

On s'intéresse maintenant à la transformation du programme écrit sous la forme d'une suite d'instructions compréhensibles par l'homme en une suite d'instructions compréhensibles par la machine.

**Définition 1.4.4** (Code source)**.** C'est l'ensemble des instructions d'un programme écrites dans un langage de programmation informatique de haut niveau.

La traduction du code source peut se faire de deux manière soit par **l'interprétation**, soit par **la compilation**.

**L'interprétation** Un programme écrit dans un langage interprété a besoin d'un programme auxiliaire appelé *interpréteur* qui analyse, traduit et exécute les instructions du programme. Le cycle d'un interpréteur est le suivant :
- lire et analyser une instruction ;
- si l'instruction est syntaxiquement correcte, l'exécuter ;
- passer à l'instruction suivante.

Ainsi, contrairement au compilateur, l'interpréteur se charge aussi de l'exécution du programme, au fur et à mesure de son interprétation. Du fait de cette phase d'interprétation, l'exécution d'un programme interprété est généralement plus lente que le même programme compilé. Mais un programme interprété est beaucoup plus portable.

**La compilation** Le code source d'un programme écrit dans un langage dit "compilé" va être traduit une fois pour toutes par un programme annexe (le compilateur) afin de générer un fichier exécutable.

*Remarque* 1.4.5. La compilateur dépend du langage. Un compilateur C ne va pas pouvoir compiler un programme écrit en PASCAL. En effet, le compilateur s'appuie sur la *syntaxe* du langage pour lequel il a été écrit.

Le langage C est un langage compilé.

### 1.4.2 Sémantique et syntaxe d'un langage

On veut ici distinguer la façon d'écrire un programme dans un langage et le but poursuivit qui va déterminer la suite des actions à faire faire. Autrement dit, on veut distinguer le sens des opérations de la manière dont on l'écrit.

La syntaxe est la branche de la linguistique qui étudie la façon dont les mots se combinent pour former des propositions, tandis que la sémantique s'attache le plus souvent à la signification des proposition.

Ces deux concepts s'appliquent aussi en informatique

**Définition 1.4.6** (Syntaxe)**.** La syntaxe est la définition de l'ensemble de règles qui régissent l'agencement des groupes de mots (entités lexicales).

Cette définition des règles est appelée *grammaire formelle* d'un langage.

Un petit rappel, chacun d'entre nous a fait au cours de ces études une étude de grammaire de langues différentes, le français, l'anglais, l'espagnol, l'allemand.... Au cours de ces cours que faisiez vous ?

Lors de la compilation, le compilateur va vérifier si le respect de ces règles est effectif ou non.

**Définition 1.4.7.** La sémantique est l'étude de la signification (le but poursuivi) des programmes.

Ainsi la syntaxe est spécifique à un langage de programmation et consiste à l'organisation **d'une** instruction tandis que la sémantique n'est pas spécifique à un langage (mais plus à un type de programmation) et consiste dans l'organisation **des** instructions du programme.

Il existe plusieurs manières de structurer la façon dont on organise la suite des instructions et leur traitement pour réaliser une même finalité. Ainsi on a différente manière de programmer appelées :

- Programmation **fonctionnelle**,
- Programmation **déclarative**,
- Programmation **impérative**,
- Programmation **objet**

Celle que nous allons utiliser est la programmation impérative

**Définition 1.4.8** (Programmation impérative)**.** Les instructions qui modifient les données sont exécutées simplement les unes après les autres.

L'état du programme est défini par le contenu de la mémoire centrale à un instant et ne peut être modifié que par une instruction,

La plupart des langages impératifs offrent de plus des moyens de structurer les suites d'instructions.

**Définition 1.4.9** (Programmation structurée)**.** Un langage de programmation structuré offre des structures de contrôle standardisées qui permettent d'organiser la suite des instructions (répétitions, branchements conditionnels)

## 1.5 Exercices

# Chapitre 2

# Structure générale d'un programme C

## 2.1 Introduction

On s'intéresse ici à la structure syntaxique d'un programme. En d'autres termes à la façon dont on doit organiser nos groupes de mots (d'instructions) dans un programme en C.

### 2.1.1 Identificateurs

Lors de nos programmes il va falloir identifier certaines parties spécifique, ou certaines entités bien particulières. Pour cela on va devoir les nommer (c'est encore une des manières les plus facile), ce à l'aide d'identificateurs.

**Définition 2.1.1** (Identificateur)**.** Un identificateur est un "*nom*" composé d'une suite de caractères alphanumériques, c'est à dire les chiffres, les lettres, auxquels s'ajoutent le caractère souligné : _ .

Un chiffre ne peut être la première lettre d'un identificateur. Un identificateur peut être tronqué à 32 caractères. Les majuscules et les minuscules sont différenciées (on dit que c'est *Case sensitive*).

Un identificateur sert à désigner :
- Une variable,
- Une fonction,
- Un type.

### 2.1.2 Mots réservés du langage

Attention certains mots sont utilisés par le langage C. Ils ont un rôle et une fonction bien définis et ne peuvent être utilisés n'importe comment. Ces mots sont appelés parfois **mots-Clefs** ou **mots réservés** Il s'agit entre autre des mots :
- Les types
  **char**, **float**, **int**, **void**,
- Des instructions
  **do**, **while**, **else**, **if**,

– divers
**static**, **return**.

**Caractères réservés** Caractères qui ont un rôle particulier en C et qui ne peuvent être utilisé seul comme identificateur.
– =, +, *, &

## 2.2 Structure d'un programme

Comme nous l'avons vu un programme est la description d'un suite d'instructions. Celles-ci doivent être agencées dans un certain ordre pour être comprise par le compilateur.

### 2.2.1 Instructions

Une instructions est un ordre de traitement que l'on donne à l'ordinateur. La syntaxe des composants élémentaires doit donc correspondre à une syntaxe compréhensible et traitable par l'ordinateur. La liste des ordres et leur sens sera vu au fur et à mesure du cours.

Une instruction est **délimitée** à la fin par un **point virgule**. Une instructions s'écrit donc

```
instruction;
```

Le point virgule indique à l'ordinateur qu'il doit **effectuer complètement** les opérations décrites dans l'instruction avant de pouvoir passer à l'instruction suivante. Cela s'appelle la **Programmation séquentielle**.

*Remarque* 2.2.1. Attention certaines instructions attendent un *"signal"* (ou un geste) de la part de l'utilisateur avant de pouvoir s'achever. Tant que l'utilisateur n'aura pas donné ce signal l'instruction complète n'est pas terminée. C'est par exemple le cas des instructions de lecture au clavier `scanf` ou `getchar`.

**Bloc d'instructions** Il est aussi parfois important de grouper un ensemble d'instructions. Ceci se fait en définissant un bloc d'instructions qui regroupe les instructions grâce à des accolades. On aura donc

```
{
instruction1;
instruction2;
instruction3;
}
```

Ce groupe d'instructions est vu par l'ordinateur comme une seule instruction même si l'ensemble des instructions qui le composent sont exécutées. Ou encore, on pourrait considérer le bloc d'instructions comme une seule et même instruction qui doit exécuter plusieurs instructions.

### 2.2.2 Organisation générale

Un programme écrit en C s'écrit toujours de la même manière. La structure générale d'un programme C est la suivante,

```
[Directives du préprocesseur]
// C'est l'ajout de potentialités supplémentaires

[fonctions secondaires]

main(){
      [Déclaration des variables]
      [instructions]
}
```

Représentons ceci sur un exemple simple.

**Exemple 2.2.2** (Affichage à l'écran)**.** *Le programme qui affiche mon nom à l'écran est :*

```
#include <stdio.h>

void main(){
    printf("Mon nom est : Jean-Jacques");
}
```

**Commentaires du programme précédent** Ici il n'y a pas de déclarations de variables.
La seule chose qui est faite est l'affichage à l'écran de la phrase :
`Mon nom est : Jean-Jacques`.

Le terme `#include <stdio.h>` indique à l'ordinateur qu'il faut qu'il aille chercher un certains nombre de *fonctions* prédéfinies qui sont décrites dans le fichier *stdio.h*. Ainsi la description de `printf` est contenue dans *stdio.h*.

Le terme `main` est le *programme principal* en terminologie PASCAL, la *fonction principale* en C. C'est là que l'ordinateur va, tout d'abord, chercher les instructions qu'il doit effectuer. Une suite d'instructions sans un `main` quelque part ne marchera pas.

La seule instruction ici est `printf("Mon nom est : Emmanuel Hyon");` On utilise la *fonction* `printf` qui affiche ce qu'on lui donne entre parenthèses. Ici la donnée est la phrase *Mon nom est : Jean-Jacques*, qui est matérialisée comme une phrase pour l'ordinateur par les guillemets.

### 2.2.3 Les variables

On rappelle que les données que nous manipulons lors d'un programme sont des espaces de la mémoire vive (RAM). Ces données peuvent être constantes (et sont appelées **constantes**) ou varier au cours de l'exécution du programme, auquel cas on les appelle des **variables**.

### Les types

Une variable est une **donnée en mémoire** caractérisée par deux choses :
- Son adresse,
- Sa valeur.

C'est pourquoi, il faut définir cette donnée par une taille et par une "*classe de valeur*" qu'il faut préciser au programme. Cette dernière définit ce que cette variable représente comme type d'information et qualifie le type de valeur de la variable.

Cette qualification s'appelle le **typage** et la **classe de valeur** le **type**.

Nous verrons au début trois types même s'il en existe beaucoup plus.
- Le type **char**
- Le type **int**
- Le type **float**.

**Le type** `char` Ce type désigne un ensemble de caractères. Un `char` contient, au moins, les caractères alphanumériques ainsi que quelques caractères supplémentaires. La façon dont l'octet ou les 2 octets sont reliés aux caractères s'appelle le codage. Il y a différent type de codage des caractères : le code ASCII, le code ISO-Latin (ASCII 8 bits), le code UTF.

**Le type** `int` Le type `int` représente les objets appartenant aux entiers relatif ($Z$). Ils sont codés sur 32 bits et le premier bit représente le signe. [1]. La valeur d'un entier peut varier entre $[-2^{31}, 2^{31}-1]$, c'est à dire :

$$[-2\,147\,483\,648, 2\,147\,483\,647] \tag{2.2.1}$$

**Le type** `float` On croit souvent représenter les réels avec le type `float`, Cela n'est pas vrai, En effet l'infini n'est pas représentable avec un ordinateur. En fait le type float `float` code un ensemble fini de nombre décimaux (fini mais de grande taille). C'est à dire, un sous-ensemble des rationnels. Ils sont appelés nombre à virgule flottante. A virgule puisqu'on parle de nombre décimaux. Très grossièrement, Flottante parce que on peut coder le nombre $16$ se code d'une manière similaire à $1.610^1$ avec un exposant et une mantisse.

### Déclaration de variables

Comme on veut manipuler des variables il va falloir expliquer au programme qu'il doit créer une zone mémoire associée à cette variable. De plus, il vaut que et le programmeur et le programme puissent savoir à quoi correspond cette zone mémoire dans le programme. A cette fin on utilise un identificateur de la variable. Enfin il faut spécifier son type.

**Définition 2.2.3** (Déclaration de variable)**.** La déclaration fournit les informations pour l'utilisation d'une variable. C'est la réservation de l'espace mémoire nécessaire à la manipulation de cette variable dans un programme.

[1] Sur les compilateurs GNU

En C la déclaration se fait de la manière suivante : `type identificateur;`

**Exemple 2.2.4** (Exemple de déclaration)**.**

```
char a; /* Declaration d'une variable a de type caractère */
int b; /* Declaration d'une variable b de type entier */
float c; /* Declaration d'une variable c de type flottant */
```

### Affectation

Mais ce n'est pas tout de posséder des zones de mémoire encore faut-il les remplir. En effet, une variable uniquement déclarée n'a pas de valeur. En d'autres termes on a réservé un espace mémoire par la déclaration mais on ne l'a pas rempli.

Remplir ou donner (voire redonner) une valeur à une variable s'appelle **l'affectation**. La première affectation d'une variable s'appelle **l'initialisation**.

*Remarque* 2.2.5. **Attention** Une variable non-initialisée est considérée comme n'ayant aucune valeur. Sa manipulation est parfois possible mais les données retournées vont être faussées.

**Règle de programmation 2.2.6.** *Gardons alors la règle de base suivant en tête :*
**Toute variable déclarée doit être initialisée avant utilisation.**

*Remarque* 2.2.7. **Attention** Une variable non initialisée peut être, selon les compilateurs et les architectures, affichée comme valant 0. Cela peut être trompeur et ne pas refléter la réalité.

### Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source. Le type est déterminé par l'ordinateur en fonction de la manière dont elle est écrite. Ainsi `const A = 1;` définit une constante entière et `const A=1.0` définit une constante de type float.

### Un exemple

**Exemple 2.2.8** (Affichage à l'écran)**.** *Le programme qui affiche la première lettre de mon nom à l'écran est :*

```
#include <stdio.h>

void main(){
 char c;
 int a,b;

 a=8;
 b=6;
 c='E';
```

```
 printf("La premiere lettre de mon prenom est : ");
 printf("%c",c);
 printf("Son nombre de lettre est : ");
 printf("%d",a);


 printf("La premiere lettre de Pierre est :")
 c='p';
 printf("%c",c);
 a=b;
 printf("Son nombre de lettre est : ");
 printf("%d",a);

}
```

**Commentaires du programme précédent**
Les valeurs de `a`, `b` et `c`.
Le formattage avec `%`.
`a` prend la valeur de `b`.

## 2.3 Les expressions

Une expression en informatique c'est la généralisation de la formule mathématique.

**Définition 2.3.1** (Expression)**.** Suite d'opérateurs et d'opérandes (variables et constantes) décrivant un calcul à effectuer. Ce calcul pouvant porter sur n'importe quel type de données (nombres, caractères...).

Une expression peut être vue comme une opération qui renvoie une **valeur** ET un **type**. Mais une expression n'est pas une instruction dès lors qu'il manque l'affectation [2], cependant l'expression fait partie de l'instruction.

L'expression la plus simple est de la forme :

`variable opérateur variable` ou `opérateur variable`.

Ou sous une forme plus générale

`expression1 op expression2`.

Donnons tout d'abord une liste (non exhaustive) des opérateurs.

[2] Qui fait elle même partie de l'expression en C

### 2.3.1 Opérateurs arithmétiques

les opérateurs arithmétiques sont :
- − (opérateur unaire)
- `+`
- `-`
- `*`
- `/`
- `%` reste de la division entière.

Ces opérateurs agissent comme classiquement en mathématique. Sur les entiers et les nombres à virgule flottante.

*Remarque* 2.3.2. Attention au type des variables sur laquelle s'applique la division /.

```
float x;
x = 3/2;
```

ici `x` vaut 1.

```
float x;
x = 3.0/2.0;
```

ici `x` vaut 1.5.

### 2.3.2 Opérateurs de comparaisons

Ces opérateurs sont aussi parfois appelés opérateurs conditionnels. Ils sont
- (< )> strictement (inférieur) supérieur,
- (<=) >= (inférieur ou égal)
- == != égal ou différent.

Ces opérateurs renvoient une valeur de vérité dont le résultat dépend de la vérification ou non de la condition. Le type et la valeur renvoyés sont différents des autres langages de programmation et on ne s'appesantira pas dessus. Ainsi,
- Valeur de vérité ← *Condition est vérifiée*
- Valeur de vérité ← *Condition non vérifiée*

*Remarque* 2.3.3.

1. Ne pas confondre l'affectation = à la comparaison ==.
2. La comparaison == n'est pas efficace sur les flottants. Car il peut y avoir des problèmes d'imprécision numérique.

**Exemple 2.3.4.** *Donnons un exemple.*

```
int a,b;
a=2;
b=3;
(a==b)
```

*Correspond à une valeur de vérité : la condition n'est pas vérifiée.*

```
int a,b;
a=2;
b=3;
(a!=b)
```

*Correspond à une valeur de vérité : la condition est vérifiée.*

```
int a,b;
a=2;
b=3;
(a <= b)
```

*Correspond à une valeur de vérité de ? ? ?*

**Opérateurs logiques booléens**

Ces opérateurs permettent de composer des expressions comportant des conditions. (reliées à un opérateur de comparaison).

- `&&` le *et*,
- `||` le *ou*,
- `!` la négation,

**Syntaxe** : `expression1 op expression2`.

**Exemple 2.3.5.** *Teste si le nombre est inférieur à* $9$ *et supérieur à* $0$ *:*

```
(a<=9) && (a >= 0)
```

*ou teste que le nombre* $a$ *est compris entre* $0$ *et* $9$ *:*

```
! ((a>9) || (a < 0))
```

*Ou encore teste si* $a$ *est nul.*

```
(a<=0) && (a >= 0)
```

### 2.3.3 Opérateurs affectation composée et d'incrémentation décrémentation

Vu en TD.

### 2.3.4 Opérateur d'adresse

Déclaration variable $a$ réserve une zone mémoire à $a$. Cette zone mémoire fait partie de l'ensemble des zone mémoires de la mémoire vive réservée à la programmation.

*Schema* 2.3.6 (Zone Mémoire). To do.

Sur le schéma la zone mémoire de $a$ est la $i^{eme}$ case du tableau.
L'opérateur `&` renvoie l'adresse de l'opérande. Ainsi `&a` renvoie l'adresse de $a$ (sous une forme cabalistique).

**Exemple 2.3.7** (Scanf)**.** *Pour la saisie au clavier on a*

```
char c;
scanf("%c",&c);
```

### 2.3.5 Sens de l'évaluation d'une expression

On voudrait savoir dans quel ordre et comment l'ordinateur évalue (c'est à dire associe une valeur à l'expression algébrique) une expression. Pour cela il faut savoir que les opérateurs n'ont pas tous la même priorité.

#### Règle de priorité des opérateurs

Les priorités sont lues de haut en bas

| Opérateur | |
|---|---|
| `( )` | → |
| &adresse (type) ! | → |
| `*` `/` `%` | → |
| `+` `-` | → |
| `<` `<=` `>` `>=` | → |
| `==` `!=` | → |
| `&&` | |
| `\|\|` | |

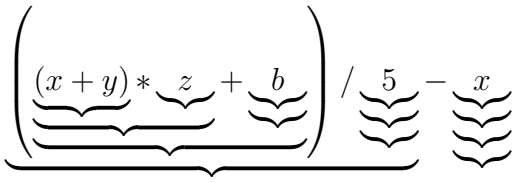
*Remarque* 2.3.8.

1. On remarque que l'ordre peut être forcé par le parenthèsage. Les parenthèses ne donnent qu'un ordre partiel d'évaluation entre les opérateurs.

2. Remarquons encore que l'ordre d'évaluation des arguments n'est pas précisé pour la plupart des opérateurs en C. C'est à dire qu'on peut d'abord évaluer l'opérande de gauche ou l'opérande de droite indifféremment.

3. Les seuls opérateurs qui sont toujours évalués séquentiellement de la gauche vers la droite sont && et ||.

**Exemple 2.3.9** (Evaluation d'une expression)**.**



**Exemple 2.3.10.** *Quelle va être l'expression de la valeur suivante ?*

```
int a,b,c;
float d;
a=2;
b=3;
c=10;
d=1.24;
  ((a+b)*z+d)/5-a
```

### 2.3.6 Transtypage ou conversion de type

A une expression correspond un type. Ce type correspond au type qui prend le plus de place en mémoire (type le plus précis). On parle de promotion. C'est pourquoi il faut parfois modifier le type d'une expression. Il y a deux manières de conversion de type.

#### Conversion implicite

C'est, grosso modo, des conversions faites par l'ordinateur.

```
float c;
int a,b;
a=3;
b=5;
c=a+b;
```

`c` est de type `float`. `a+b` de type `int`. Il y a un conversion de type implicite qui transforme le type du résultat de `a+b` en flottant.

**Conversion implicite non désirée** Il se peut cependant **en C** que l'ordinateur fasse des conversions implicites malvenues. Ces transtypages (conversions) sont maintenant affichés par des avertissements sur certains compilateurs récents.

```
float c;
int a,b;
a=1;
c=1.5;
b=a+c;
```

Ici $b$ vaudra $2$ et non $2.5$ car le transtypage induit une perte de précision

**Exemple 2.3.11.** *Analogie des petites et des grosses boites*

#### Conversion explicite

Quelque fois c'est l'utilisateur qui décide de faire cette conversion. Mais il le fait en toute connaissance de cause.

```
float tranches;
int duree,nombre;

nombre=(int) duree * tranche;
```

**opérateur de transtypage** C'est l'opérateur `( type )` avec type le type de sortie désiré.

## 2.4 Exercices Théoriques

**Exercice 2.4.1** (Questionnaire)**.** *Écrire un programme qui demande à l'utilisateur son âge, sa taille (exprimée en $m$) et son poids (exprimé en $kg$) puis qui affiche à l'écran les informations saisies.*

**Exercice 2.4.2** (Déroulement d'un programme)**.** *Dérouler le programme suivant et donner les valeurs finales de toutes les variables déclarées.*

```
#include <stdio.h>
main()
{
int a,b,c,d;
a=2;
b=a+3;
c=b/a;
d=c+b%a;
a=(a+b+c+d)/2;
printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);
getchar();
}
```

**Exercice 2.4.3** (convertisseur 3)**.** *Écrire un programme qui convertit un horaire exprimé en secondes en heures minutes secondes.*

# Chapitre 3

# Les instructions structurées ou Structures de contrôles

## 3.1 Introduction

Commençons par

*"pour choisir, il faut penser à ce qu'on pourra faire et se remémorer les conséquences de ce qu'on a déjà fait."*
Bergson

eh ben l'informatique pareil.

**Exemple 3.1.1** (Rappel exemple 1.3.3)**.** *Dans la recherche d'un fichier de nom $x$ dans l'ordinateur on était amené à*

- *Répéter un certain nombre de fois la même recherche d'un nouveau fichier.*
- *Tester si c'était celui là le bon.*

Ainsi, l'écriture d'un programme procède de plusieurs possibilités d'ordonner un ensemble d'actions.

- Suite d'actions séquentielles.
- Choix entre plusieurs possibilités qui vont déterminer la suite du programme.
- Répétitions de la même action un certain nombre de fois.

On dit que ces agencements sont des structures car ils décident l'ordre des instructions. Ces structures sont nécessaires car elles permettent de contrôler le comportement du programme en fonction de son déroulement.

## 3.2 Les conditionnelles

On introduit les structures relatives au choix.

**Définition 3.2.1** (But d'une structure conditionnelle)**.** Le but d'une telle structure est de pouvoir faire un **aiguillage** entre différentes possibilités de déroulement de programme.

Ces possibilités d'aiguillage traduisent des choix à effectuer similairement à ceux que l'on fait dans la vie courante.

**Exemple 3.2.2** (Sans alternatives)**.** *Heure fin des cours : Si l'heure de fin des cours est passée je rentre chez moi.*

**Exemple 3.2.3** (Avec Alternative)**.** *J'ai* 16 *ans j'ai le droit en toute modération de prendre de la bière dans un bar, sinon (si je suis plus jeune) je n'ai pas le droit.*

### 3.2.1 Sémantique : Traduction en suite d'actions

Traduisons cette suite d'actions sous la forme d'un schéma.

*Schema* 3.2.4 (Continuation 3.2.2). Représentation algorithmique de l'exemple 3.2.2.

*Schema* 3.2.5 (Continuation 3.2.3). Représentation algorithmique de l'exemple 3.2.3.

### 3.2.2 Syntaxe

Le mot-clé **if** permet d'effectuer une instruction (simple ou composée) de manière conditionnelle. Il peut être suivi du mot-clé **else** pour spécifier l'instruction à effectuer lorsque la condition est fausse (l'alternative).

```
if (condition)
/* si la condition est vraie */
instruction
```

ou

```
if (condition) {
    /* si la condition est vraie */
    instruction 1
    ...
    instruction n
}
```

ou encore si il existe une suite d'instructions lorsque la condition est fausse.

```
if (condition) {
    /* si la condition est vraie */
    instruction 1
    ...
    instruction n
} else {
```

```
        /* si la condition est fausse */
        instruction 1
        ...
        instruction n
    }
```

**Le code des deux exemples**

Exemple 3.2.2.

```
# include<stdio.h>

main(){
    int heure;

    printf(''Saisissez l heure'');
    scanf(``%d'',&heure);
    if (heure >= 18) printf(''Je rentre a la maison'');
}
```

Exemple 3.2.3.

```
#include<stdio.h>

main(){
    int age;

    printf(''Saisissez votre age'');
    scanf(``%d'',&age);
    if (age >= 16){
        printf(''C'est bon tu es autorise'');
        printf(``Un demi ?'');
    }
    else {
        printf(''Tu es trop petit'');
        printf(``Un cacolac  ?'');
    }
}
```

Le déroulement du programme précédent est
quand age=10

quand age=20.

*Remarque* 3.2.6 (Attention au bloc d'instructions). Le positionnement des accolades permet de définir correctement les instructions à effectuer après aiguillage. Que donne le code suivant ?

```
# include<stdio.h>

main(){
    int heure;

    printf(''Saisissez l heure'');
    scanf(``%d'',&heure);
    if (heure >= 18)
printf(''il est 6 heure passee");
printf(''Je rentre a la maison'');
}
```

Comment le corriger pour ne pas passer pour un mauvais élève ?

*Remarque* 3.2.7 (Convention d'écriture). Voici quelques conventions d'écriture qui permettent de rendre son code plus lisible :

- Ouverture après la condition ou après le else.
- Code à effectuer indenté (décalé).
- Fermeture de l'accolade au même niveau que le if et le else.

**Exemple 3.2.8** (Valeur absolue)**.** *Donner le code correspondant au calcul de la valeur absolue de* a-b.
**Etape 1** *: En français*
**Etape 2** *: Schéma Algorithmique*
**Etape 3** *: Programme*

```
#include <stdio.h>

main(){
int a,b;
int valeur_absolue;
scanf("%d",& a);
scanf("%d",& b);
if (a-b < 0){
            valeur_absolue=b-a;
}
else {
            valeur_absolue=a-b;
}
printf("La valeur absolue est %d,valeur_absolue);
}
```

**Conditionnelles imbriquées**

Il est possible d'inclure une structure conditionnelle à l'intérieur d'une autre structure conditionnelle.

**Expression du problème en français** . On reprend l' exemple 3.2.3, mais on voudrait que l'âge soit valide. C'est à dire que ce soit un entier positif et inférieur à 130. Ensuite si l'âge est valide alors je regarde si on est en âge d'être dans un bar.

**Expression de l'algorithme**
Etape 1 -> Si l'age est compris entre 0 et 130
->alors
->->si l'âge est supérieur à 16
-> ->alors droit de rentrer dans le bar
-> ->sinon pas le droit
->sinon afficher que l'âge n'est pas valide

**Le Code**

```
#include<stdio.h>

main(){
    int age;

    printf(''Saisissez votre age'');
    scanf(``%d'',&age);
    if (age > 0) && (age <=130){
        if (age >= 16){
            printf(''C'est bon tu es autorise'');
            printf(``Un perrier ?'');
        }
        else {
            printf(''Tu es trop petit'');
            printf(``Un cacolac  ?'');
        }
    else {
        //alternative du premier if
        printf("Age non valide");
    }
// fin des conditionnelles
printf(''Fin'');
}
```

### 3.2.3 Les boucles : Introduction

**Définition 3.2.9.** On appelle boucle le fait de répéter un nombre de fois (déterminé ou non) la même action ou la même série d'actions.

Reprenons des exemples de la vie courante

**Exemple 3.2.10** (Effacer 5 fichiers dans l'ordinateur)**.** *Cela consiste à répéter* 5 *fois l'opération d'effacement.*

**Exemple 3.2.11** (Effacer tous les fichiers d'un dossier)**.** *Ceci consiste à effacer des fichiers tant qu'il en reste dans le répertoire.*

On se rend compte qu'il y a deux types de boucle
- for : pour
- while : tant que

### 3.2.4 la boucle for

#### Introduction

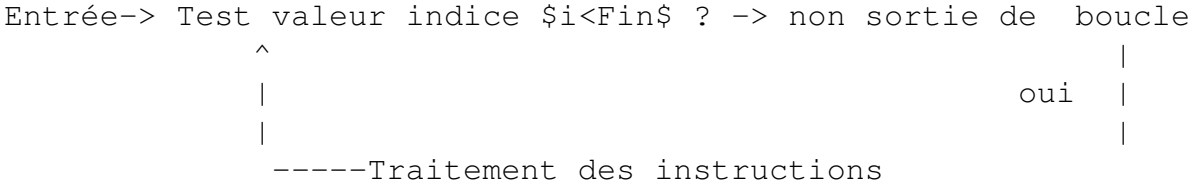
**Utilisation préconisée** La boucle for est utilisée de préférence
- lorsque l'on a un nombre **prédéterminé** d'opérations à faire.
- lorsque ce nombre ne dépend pas de variables dont la valeur est modifiée dans la boucle.

#### Syntaxe

```
int i; // indice courant.
int deb;
inf fin;

for(i=deb;i<fin;i++){
      instruction 1
      ...
      instruction n
}
```

**Sémantique ou expression de l'algorithme** On représente la boucle par le schéma suivant.




On remarque que la boucle apparaît directement sur ce schéma.

**Deux exemples**

**Exemple 3.2.12** (Saisie de $10$ nombres au clavier et affichage.)**.**
**En français :**
*La suite des opérations : Je pars d'un indice égal à* $0$ *(c'est une convention) et je le fait varier jusqu'à* $9$ *inclus ou* $10$ *exclu. Pour chaque valeur d'indice je demande de lire et d'afficher.*

**En algo** *cela donne le schéma suivant Pour* $i$ *variant de* $debut = 0$ *à* $fin = 9$ *répéter : demander une valeur, lire la valeur, afficher la valeur.*

**En code**

```
#include <stdio.h>

main(){
int i; // indice courant.
int entier;
int deb;
inf fin;

  deb=0;
  fin=9;

  for(i=deb;i<=fin;i++){
      printf(''Saisir nombre'');
      scanf("%d",&entier);
      printf("Vous avez saisi le nombre %d",entier);
  }
}
```

**Exécution du programme** *Commenter le déroulement Au début affectations :* $deb = 0$ *et* $fin = 9$*. Ensuite,*

**Autres exemples : des expressions qui dépendent de la valeur d'indice de la boucle**

**Exemple 3.2.13** (Affichage de 5 entiers à partir de 100)**.**
**En français** *:*
*pour l'entier valant* $100$*, j'affiche l'entier,*
*pour l'entier valant* $101$*, j'affiche l'entier,*
*pour l'entier valant* $102$*, j'affiche l'entier,*
*pour l'entier valant* $103$*, j'affiche l'entier,*
*pour l'entier valant* $104$*, j'affiche l'entier,*

**En algo** *: pour* $i$ *de* $100$ *à* $104$ *répéter : afficher l'entier*
**En code**

```
#include <stdio.h>

main(){
int i; // indice courant.
int entier;
inf fin;

 deb=0;
 fin=5;

 for(i=0;i<fin;i++){
      entier=100+i;
      printf("Le nombre %d",entier);
 }
}
```

**Déroulement** *Commentaire du déroulement de l'exécution.*

**Deux boucles imbriquées**

On veut afficher la table de multiplication de $2$, $3$ et $4$ chacune entre $1$ jusqu'à $5$.

**En français** Faire varier la table
Pour $2$ afficher la table des $2$ de $1$ à $5$....
Pour $3$ afficher la table des $3$ de $1$ à $5$....
Pour $4$ afficher la table des $4$ de $1$ à $5$....
Affichage de la table de $x$ de $1$ à $5$
Pour $1$ afficher $x \times 1$
Pour $2$ afficher $x \times 2$
Pour $3$ afficher $x \times 2$
Pour $4$ afficher $x \times 2$
Pour $5$ afficher $x \times 2$

**En Algo**

**Code**

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

```
main(){
int i; // indice courant boucle 1
int j; // indice courant boucle 2
int fin1;
int fin2;
int deb1;
int table;

deb1=2;
fin1=5;
fin2=5;

for(i=deb1;i<fin1;i++){
      // Affichage de la table de i
      for(j=1;j<6;j++){
            printf("%d multiplie par %d vaut %d",i,j,i*j);
      }
}
}
```

### 3.2.5 La boucle while

**Utilisation préconisée** La boucle while est utilisée de préférence

- lorsque le nombre d'opérations à faire dépend directement de la satisfaction d'une condition à l'intérieur de la boucle.
- lorsque cette condition est **modifiée** au moins une fois dans la boucle.

**Règle de programmation 3.2.14.** *Il est extrêmement important de modifier dans la boucle les valeurs qui sont impliquées dans la condition qui détermine la sortie de la boucle. Autrement l'ordinateur ne "s'arrête" pas.*

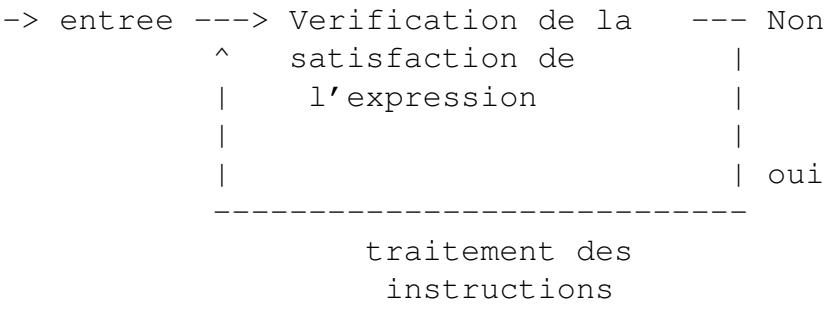
**Syntaxe**

```
/* On note expr(a) une expresson booleenne qui depend de a */
int a; /* type indifférent */
a=10;

while (expr(a) ){
      instruction 1
      ...
      instruction modifiant la valeur de a
      de sorte que la satisfaction ou non de expr(a) est modifiee
```

```
      ...
      instruction n
}
```

**Sémantique ou expression de l'algorithme**




**Quelques exemples**

**Exemple 3.2.15** (Saisie de caractères aux claviers)**.** *On veut saisir des caractères au clavier et les afficher mais dès que le caractère saisi est $n$ alors la saisie s'arrête.*

**En français**
*La condition de satisfaction est : le caractère est différent de $n$. En effet, tant que le caractère est différent alors la boucle continue.*
*Les instructions dans la boucle sont :*
*Demande de saisie du caractère et affichage.*
*Le caractère lu doit être le caractère impliqué dans la condition de satisfaction.*

**En pseudo Langage**
*Donner une valeur au caractère que l'on saisira plus tard dans la boucle de telle sorte que l'expression booléenne puisse être évaluée.*
*Tant que l'expression n'est pas vérifiée faire :*

1. *Lire le caractère.*
2. *Afficher le caractère.*
3. *Affectation de la valeur du caractère lu à la valeur du caractère qui est impliqué dans l'expression booléenne (Peut être fait en $1$).*

**Le programme**

```
#include <stdio.h>

main(){
  char a;
```

```
  a='o';
  while (a <> 'n'){
      printf("saisissez un caractère au clavier");
      scanf("%c",&a);
      printf("Vous avez saisi la caractère %c",a);
  }
printf("Vous avez arrete la boucle");
printf("c est la fin");
}
```

**Déroulement de l'exécution du programme**. *On s'intéresse au déroulement dans deux cas,*

- *Quand ligne2* a='n'.
- *Quand ligne 2* a='o'.

**Exemple 3.2.16** (Reprendre l'exemple avec un âge valide)**.** *On veut maintenant reprendre l'exercice de discrimination des clients du bar. On cherche à ce que le test de l'âge ne puisse se produire que quand l'âge est valide. Cela implique que tant que l'âge n'est pas valide l'utilisateur ressaisi son âge.*

**En pseudo langage**

**En code**

## 3.3 Quelques erreurs classiques

On présente ici un certain nombre d'erreurs classiques qui se produisent lors des premiers maniement de boucle. Ceci sera fait en illustrant la rédaction d'un programme simple.

dont le but est de trouver la valeur d'un indice $n$ tel que son inverse soit plus petite que un nombre donné que l'on nomme précision.

Ci-dessous une suite de programmes faux. Par soucis de concision on se restreint au main.

bf Premier programme

```
main(){
  int n;
  float precision, inverse;

  n=0;
  precision=0.00023;
  while(inverse > precision) {
    printf("Pour %d cela n'est pas bon",n);
    printf("Encore un effort");
  }
```

```
printf("valeur correcte : %d",n);
}
```

Quelles sont donc les erreurs ?

- $n$ n'a pas été initialisé correctement ;
- $1/n$ division de deux entiers donc cela devrait toujours donner $0$.

**Second Programme**

```
main(){
  int n;
  float precision, inverse;

  n=1;
  precision=0.00023;
  inverse= ( (float) 1)/n;
  while(inverse > precision) {
    printf("Pour %d cela n'est pas bon",n);
    printf("Encore un effort");
  }
printf("valeur correcte : %d",n);
}
```

Quelles sont les erreurs ?

- $n$ ne varie pas et l'expression booléenne qui dépend de $n$ ne varie pas non plus ;
- La programme ne va jamais sortir de la boucle [1].

**Troisième Programme**

```
main(){
  int n;
  float precision, inverse;

  n=1;
  precision=0.00023;
  inverse= ( (float) 1)/n;
  while(inverse > precision) {
    printf("Pour %d cela n'est pas bon",n);
    inverse= ( (float) 1)/n;
    printf("Encore un effort");
    n=n+1;
  }
printf("valeur correcte : %d",n);
}
```

[1] On a bien le temps d'aller prendre un café

Il reste une dernière erreur (subtile). En effet le test est fait avec une valeur décalée par rapport à l'indice.

Cas 4 Exemple correct

```
main(){
  int n;
  float precision, inverse;

  n=1;
  precision=0.00023;
  inverse= ( (float) 1)/n;
  while(inverse > precision) {
    printf("Pour %d cela n'est pas bon",n);
    printf("Encore un effort");
    n=n+1;
    inverse= ( (float) 1)/n;
  }
printf("valeur correcte : %d",n);
}
```

## 3.4 Exercices Théoriques

### Structures Conditionnelles

**Exercice 3.4.1** (Pair ou impair ?)**.** *Écrire un programme qui demande à l'utilisateur un entier positif et qui affiche à l'écran s'il est pair ou impair.*

**Exercice 3.4.2** (Equation du second degré)**.** *Écrire un programme qui résoud une équation du second degré dont les coefficients sont entrés par l'utilisateur (on supposera que le coefficient de* $x^2$ *n'est pas nul).*

**Exercice 3.4.3** (Maximum de trois entiers saisis au clavier)**.** *Écrire un programme qui demande trois entiers à l'utilisateur et qui affiche la plus grande valeur saisie.*

### 3.4.1 Boucles

#### Boucles simples

**Exercice 3.4.4.** *Écrire un programme qui affiche à l'écran les carrés de tous les nombres de* $1$ *à* $n$ *où* $n$ *est un entier donné par l'utilisateur. Modifier le programme pour qu'il n'affiche que* $10$ *nombres au plus par ligne.*

**Exercice 3.4.5.** *Écrire un programme qui affiche tous les multiples de* $k$ *compris entre* $a$ *et* $b$ *où* $k$*,* $a$ *et* $b$ *sont trois entiers entrés par l'utilisateur.*

### Boucles et conditionnelles imbriquées

**Exercice 3.4.6.** *Écrire un programme qui demande à l'utilisateur deux entiers $n$ et $p$ puis qui affiche à l'écran tous les cubes des entiers de $n$ à $n + 10 * p$ en revenant à la ligne tous les $p$ nombres.*

### Boucles imbriquées

**Exercice 3.4.7.** *Écrire un programme qui affiche un triangle rempli d'étoiles, s'étendant sur un nombre de lignes entré au clavier, comme dans l'exemple suivant :*

```
Nombre de lignes = 5
*
**
***
****
*****
```

# Chapitre 4

# Les tableaux
# Première partie

## 4.1 Introduction

Un bref rappel : On nomme structure de données une structure logique destinée à contenir des données en leur donnant une organisation permettant de simplifier leur traitement.

Pour prendre un exemple de la vie quotidienne, on peut présenter un annuaire téléphonique sous différentes formes

- par nom,
- par type d'activité (pages jaunes),
- par numéro téléphonique (annuaires inversés),
- par adresse

Dans ce cas, à chaque usage correspondra une structure d'annuaire appropriée. Ici, les données sont toutes les mêmes mais c'est la manière d'aller les chercher qui diffère.

De toutes les structures de données possibles nous nous intéressons à l'une des plus simple : le **tableau** à une seule dimension.

**Définition 4.1.1.** Un tableau est une suite adjacente de variables (éléments du tableau) d'un type donné.

Les éléments sont accessibles uniquement par un mécanisme d'indexage (ou accès indexé).

On voit apparaître (encore) deux notions

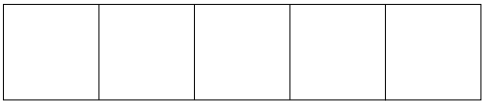
1. celle de *donnée*.
2. celle *d'accès à la donnée*.

*Remarque* 4.1.2 (Allégorie d'un tableau). Un tableau peut être vu comme une suite contiguë de cases mémoire, toutes de la même taille. Cette taille étant déterminée par la taille du type de donnée stockée. On accède à ces cases par un mécanisme d'indexation qui est le mécanisme qui permet d'associer sans ambiguité un contenu et une référence.

**Propriétés** Énonçons quelques propriété de base d'un tableau.

1. Un tableau ne contient que des éléments du même type (entier, caractère, flottant).
2. Un tableau a une taille qui est le nombre de cases qu'il contient.
3. Les cases du tableau ont toute un indice. Cet indice décrit la place de la case parmi l'ensemble des cases.
4. Les numéros des indices sont en ordre croissant. La première case du tableau est indicée par $0$ (en C) et la dernière case par $taille - 1$.

**Schéma d'un tableau de taille** $5$



## 4.2 Manipulation du tableau en C

On présente ici les manipulations d'un tableau , on choisit au début les concepts les plus simples qui concernent la manipulation de tableau.

### 4.2.1 Déclaration d'un tableau

La déclaration d'un tableau est de la forme :
type nom[nombre d'entier] ;

En réalité, pour être plus propre (ce qui veut dire pour éviter des causes supplémentaires d'erreur), on peut définir la taille du tableau comme une constante dans le main :
const taille = valeur.

Ensuite on ne traite plus que la constante taille et non un nombre donné lors des manipulations de tableaux de la même taille. Cela permet de ne changer la taille qu'à un seul endroit du programme.

Les deux déclarations suivantes sont équivalentes :
int tableau[4] ;
et
const taille=4 ;
int tableau[taille] ;

### 4.2.2 Initialisation d'un tableau

L'initialisation d'un tableau consiste à affecter une valeur pour chaque case.

*Remarque* 4.2.1. Attention un tableau non initialisé va (selon les compilateurs) afficher des valeurs qui vous sembleront aberrantes ou parfois des zéros. Considérer, pour être le plus générique possible, que un tableau non initialisé ne contient rien.

Il existe deux manières d'initialiser un tableau à la déclaration ou lors du programme.

**Initialisation à la déclaration** Cette manière convient pour des tableaux de petite taille. On met les valeurs que l'on doit affecter aux cases du tableau entre accolades. Ce qui donne :
int tab[5] = {4, 6, 8, 12, 20} ;

**Initialisation au cours du programme** Cette voie est privilégiée pour des tableaux de grandes tailles. Il faut accéder à chaque case et lui affecter une valeur.

### 4.2.3 Accès à une entrée du tableau

L'accès à la $i$ème case du tableau appelé "tab" (en considérant qu'elle existe) se fait par l'instruction tab[i].

Ainsi pour donner une valeur à la $i+1$ème case, l'instruction à utiliser est :
tab[i]=3 ;

De la même manière, pour : utiliser la valeur de la $i+1$ème case du tableau, soit pour l'affichage soit pour l'affectation d'une autre variable, les instructions sont respectivement
printf("%d",tab[1]) ; (avec tab tableau d'entiers)
et
a=tab[i] ; (a et tab de même type).

Ainsi a=tab[i] ; signifie que la variable a prend comme valeur le contenu de la $i+1$ème case du tableau

Ainsi printf("%d",tab[1]) ; signifie que l'on affiche la première case du tableau.

L'initialisation du tableau de tout à l'heure devient alors .

```
int tab[5];
tab[0]=4;
tab[1]=6;
tab[2]=8;
tab[3]=12;
tab[4]=20;
```

**Précautions d'usage 1 : Indice et case traitée**

**Précautions d'usage 2 : Copie de tableau** Soit A et B deux tableaux d'entiers de même taille. L'affectation de la valeur d'un tableau A à un autre tableau B n'est pas possible. Autrement dit, l'ensemble des valeurs d'un tableau ne peut être passé à un autre tableau en une seule instruction. Il faut affecter les valeurs du tableau, valeur par valeur. D'un point de vue langage C la suite d'instructions :

```
const taille=2;
int A[taille] ={4, 6};
int B[taille];
B= A;
```

n'est **pas** valide.

Celle qui est correcte est

```
const taille=2;
int A[taille] ={4, 6};
int B[taille];
B[0]=A[0];
B[1]=A[1];
```

**Précautions d'usage 3 : Plage des indices** Une valeur d'indice qui n'appartiendrait pas à l'intervalle $[0, taille - 1]$ renvoie une erreur et le programme s'arrête. En effet une telle valeur n'a pas de signification pour l'ordinateur. Elle n'est corrélée à aucune adresse mémoire ce qui fait que une instruction comportant une référence au tableau avec une telle valeur d'indice ne peut pas être traitée correctement.

## 4.3 Manipulations de tableau : Quelques Exemples

### 4.3.1 Affectation, copie et affichage de tableaux de taille donnée

```
#include<stdio.h>

main(){
  const taille=10;
  int i;
  int A[taille];
  int B[taille];

  /* initialisation de A */
  for(i=0;i<taille;i++){
    printf("Entrer la valeur de la %d eme case du tableau,%d);
    scanf("%d",&A[i]);
  }

  /* Recopie de A en B */
  for(i=0;i<taille;i++){
    B[i]=A[i];
  }
```

```
  /* Affichage de A */
  printf("Le tableau A est");
  for(i=0;i<taille;i++){
    printf("A[%d]= %d",i,A[i]);
  }
}
```

# Exercice Théoriques

**Exercice 4.3.1.** *Décrire le résultat produit par le programme suivant :*

```
#include <stdio.h>
main()
{
   int i, b = 0;
   int c[lO] = {1, 2, 3, 4, 5 , 6, 7, 8 , 9, 10};
   for (i = 0; i < 10; i++)
   {
       if ((c[i] % 2) == 0)
       {
           b = b+ c[i];
       }
   }
   printf("dans boucle %d %d",i,b);
}
```

**Exercice 4.3.2.** *On suppose un tableau d'entier de taille* $5$*. On veut échanger le contenue de la case d'indice* $i$ *et le contenu de la case* $j$ *(les indices sont demandés à l'utilisateur).*

**Exercice 4.3.3.** *On suppose un tableau d'entier de taille* $5$*. On veut supprimer la case d'indice* $j$ *et le remplacer par le contenu de la case d'indice* $j + 1$*.*

# Chapitre 5

# Quelques principes de bases de l'algorithmique

## 5.1 Introduction

On ne va pas rentrer dans une définition formelle de l'algorithme en informatique mais uniquement donner un certains nombre de concepts qui visent à mieux faire comprendre ce qu'est un algorithme.

Un algorithme est une suite d' opérations à effectuer en vue de réaliser (ou exécuter) une opération donnée qui peut s'énoncer en langage clair.

Les algorithmes ne sont pas restreints à l'informatique. Ils existent dans la vie quotidienne, ainsi une recette de cuisine peut être considérée comme un algorithme. Une suite d'indications décrivant le chemin pour aller chez quelqu'un aussi. Mais ils restent, quand même, principalement utilisés en mathématique et en informatique. Ils sont notamment essentiels dans la recherche d'une solution avec un ordinateur, parce que pour obtenir une solution on doit décrire une série d'instructions claires.

L'autre aspect qui conduit à leur utilisation en informatique tient au fait que le plus souvent un algorithme se trouvent être une répétition donnée d'un jeu d'instructions. Dans ce cas on parle des étapes d'un algorithme. Enfin, un algorithme *termine* c'est à dire trouve la solution en un nombre fini (pas forcément petit) d'opérations (ou d'étapes).

Un algorithme est **indépendant du langage de programmation**. C'est sa mise en oeuvre (ou implémentation) au moyen d'un langage de programmation qui va le rendre dépendant. L'algorithme mis en oeuvre constitue alors une brique de base d'un programme informatique.

Ces algorithmes utilisent de manières plus ou moins cachés un certain nombre de concepts dont certains ont été codifiés

## 5.2 Accumulateur

**Introduction** Question : Comment calculez-vous une somme de 4 nombres ? Par exemple comment calcule-t-on de tête la somme de 2 + 4 + 5 + 7 ? La plupart du temps on fait $2 + 4 = 6$, puis

$6 + 5 = 11$ et enfin $11 + 7 = 18$. Cela marche parce que l'addition est associative. Que sont donc le $6$, le $11$ et le $18$. Ces nombres sont le stockage du résultat temporaire de votre addition.

Ces nombres sont appelés **accumulateurs**.

**Propriétés** Enumérons quelques propriétés à la vue de cet exemple.

1. La valeur finale de l'accumulateur est le résultat que l'on cherche.
2. La valeur prise par l'accumulateur a chaque étape est le résultat du calcul à une étape qui s'ajoute à ce qu'on a déjà calculé.
3. L'initialisation peut apparaître sous deux formes soit on fixe comme valeur initiale l'élément neutre de l'opération, soit on prend la première valeur qui appairait lors des calculs.
4. La valeur de l'élément ajouté peut dépendre directement de l'étape à laquelle on se trouve ou de la valeur d'un nombre qui dépend de l'étape.
5. L'opération utilisée est associative.

Dans quels cas utilise-t-on a un accumulateur ? Cette question est un peu difficile et viens avec la pratique. Pour qu'un accumulateur doive être utilisé il suffit souvent qu'un certain nombre des conditions ci-dessus soient vérifiées. En fait cela met en ouvre le concept d'invariant de boucle que l'on verra plus tard dans le chapitre[1].

**Attention** l'opération d'ajout n'est pas forcément l'addition cela peut être le maximum ou la multiplication. Toute la difficulté est de deviner quel est l'opération utilisée.

### 5.2.1 Un exemple d'accumulateur simple

On veut calculer la somme des $n$ premiers nombres entiers. En mathématiques cela veut dire calculer $1 + 2 + .... + n$.

Ceci est équivalent à $(((1 + 2) + 3) + \ldots + n)$.

C'est aussi équivalent à la première étape à l'addition de $0 + 1$ puis, à la seconde étape au résultat temporaire on ajoute $2$, à la troisième étape on ajoute au résultat temporaire $3$. Ainsi à la $i$ème étape on ajoute $i$ à l'accumulateur.

**Description de l'algorithme**
Initialisation de l'accumulateur à $0$ (élément neutre de l'addition).
Faire de $1$ jusqu'à $n$ : accumulateur+i (où $i$ est le compteur d'étapes)

Le résultat est-il bien celui attendu ?

Le code est donc

```
#include<stdio.h>

main(){
  int accumulateur;
```

[1] Si vous êtes en train de réviser cela se dit plus tard dans la soirée

```
  int i;
  int n;

  n=21;
  accumulateur=0;
  for(i=1;i<22;i++){
    accumulateur=accumulateur+i;
  }
printf("valeur finale %d",accumulateur);
}
```

### 5.2.2 Le maximum des éléments d'un tableau

**But** : On veut calculer le maximum d'un tableau tab de taille taille dont les valeurs ont déjà été rentrées.

Il faut tout d'abord remarquer que le max est associatif :

$$\max\left(\max(a,b),c\right) = \max(a,b,c).$$

Donc il nous suffit en fait pour connaître le maximum à l'étape $i$, de connaître le maximum à l'étape $i-1$ et de le comparer au terme contenu dans la $i$ème case.

Il y a deux cas possibles

- Soit le maximum de l'étape $i-1$ est plus grand que le $i$ème terme auquel cas le max à l'étape $i$ est le maximum des $i-1$ premiers termes.
- Soit le $i$ème terme est plus grand que le maximum de l'étape $i-1$ auquel cas le max à l'étape $i$ est le $i$ème terme.

**Description de l'algorithme** accumulateur est le premier terme du tableau
de $i$ de 2 à taille **faire**
si tableau[i] > max alors max<-tableau[i] sinon ne rien faire

**Code**

```
#include<stdio.h>

main(){
  const taille = X;
  int accumulateur;
  int i;
  int tab[taille];

  // initialisation du tableau
```

```
  accumulateur=tab[0];
  for(i=0;i<taille-1;i++){
    if (tab[i+1]>accumulateur) {
                acumulateur=tab[i];
    }
  }
  printf("valeur finale le max est %d",accumulateur);
}
```

## 5.3 Suites récurrentes

### 5.3.1 Exemple introductif

On s'intéresse à des suites de valeurs définies d'une manière récurrente. Cela signifie que le $n$ème terme dépend du terme précédent. Sous la forme la plus simple on a :

$$U(n) = f(U(n-1)).$$

Ces suites ont beaucoup d'applications dans la vie courante et selon l'expression de la fonction il n'est pas forcément facile de les calculer par une formule mathématique.

**Un exemple** On veut calculer la 3ème valeur de la suite *géométrique* :

$$U(n) = q \times U(n-1)$$

avec $U(0)$ différent de $0$.

Comment effectuer le calcul ? On a

$$U(0) = a, U(1) = qU(0) = aq, U(2) = qU(1) = aq^2 \text{ et } U(3) = qU(2) = aq^3.$$

La question est comment peut-on écrire cette suite de calculs pour connaître la valeur pour n'importe quel valeur de $n$ ? Si $n$ vaut $1000$ il ne sera pas possible de déclarer $1000$ variables ! ! Comment alors, généraliser ce calcul afin d'effectuer toujours la même suite d'opérations à chaque étape en utilisant une seule valeur ?

On utilise une valeur temporaire qui à chaque étape stocke la valeur de $U(n)$. L'inclusion dans une boucle se fait alors "facilement". C'est un accumulateur en un peu plus compliqué et on verra plus tard le lien.

Si on utilise une variable temporaire. Cette variable temporaire vaut à la fin de la boucle la valeur du terme de la suite qu'on calcule. Le calcul précédent se fait de la manière suivante :
temporaire$= U(0) = a$,
temporaire$=U(1) = f(U(0))$ c'est à dire temporaire$= f($temporaire$)$.
A l'ordre 2 on a
temporaire$= U(2) = f(U(1))$, c'est à dire temporaire$= f($temporaire$)$. Et donc on obtient le calcul pour n'importe quelle valeur de $n$ :

$$\text{temporaire} = U(n) = f(U(n-1)) = f(\text{temporaire}).$$

**Illustration** Illustrons ce mode de calcul sur l'exemple précédent et observons le comportement de temporaire.

...

### 5.3.2 Modélisation et exemple de calcul

Une suite récurrente d'ordre 1 est une suite définie par

$$U(n) = f(U(n-1)).$$

Lorsqu'on cherche à connaître la $n$ème valeur d'une suite récurrente d'ordre 1 on fonctionne de la manière suivante.

- On utilise une valeur temporaire qui va prendre pour valeur à chaque itération la valeur de la suite.
- Cette valeur temporaire est initialisée avec la constante initiale de la suite ($U(0)$).
- Cette valeur est utilisée à chaque étape pour calculer la nouvelle valeur de la suite.
- A chaque étape, la valeur de la variable temporaire à l'entrée de la $i$ème itération est $U(i-1)$ et à la sortie $U(i)$.
- Pour connaître le $n$ème terme de la suite on fait $n$ fois la boucle.

**Exemple 5.3.1** (Conjecture de Syracuse)**.** *La suite de Syracuse est définie de la manière suivante :*

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \textit{si } u_n \textit{ est pair} \\ 3u_n + 1 & \textit{sinon.} \end{cases}$$

On a une suite récurrente dont le calcul dépend de l'itération (selon qu'elle est paire ou non). Mais cela reste le même modèle.

Exprimons donc l'algorithme en fonction de la méthode vue plus haut.

- Initialisation de la variable tempo avec une valeur quelconque rentrée par l'utilisateur
- Boucle pour i variant de 1 à $n$ faire
  Calcul de $U(i)$ à la fin tempo devra valoir $U(i)$
  Au début de cette boucle tempo vaut $U(i-1)$
  - Si $U(i-1)$ est pair alors $U(i) = U(i-1)/2$, on a donc tempo $=$ tempo$/2$
  - Si $U(i-1)$ est impair alors $U(i) = 3U(i-1) + 2$, on a donc tempo $=$ 3tempo $+ 1$

  Ce qui donne en code

```
#include <stdio.h>

main(){
  int tempo;
  int val_init;
  int nb_iter=20;
  int i;
```

```
  scanf("%d",&val_init);
  // Initialisation methode
  tempo=val_init;
  for(i=0;i<nb_iter;i++){
    if (tempo%2==0) {
      tempo=tempo/2;
    }
    else{
      tempo=3*tempo+1;
    }
  }
printf("Le resultat final est %d",tempo);
}
```

### 5.3.3 Suite récurrente d'ordre supérieur à 1

Ces suites se définissent de la manière suivante

$$U(n) = f(U(n-1), U(n-2), \ldots, U(n_k)).$$

avec $k$ fini.

Cela fonctionne strictement de la même manière il suffit juste maintenant de stocker les $k-1$ termes précédents et de les mettre à jour à chaque étape. Ainsi, il faut créer $k-1$ variables temporaires pour stocker les termes et à la fin de chaque calcul de $U(i)$, leur donner la bonne valeur.

**Exemple 5.3.2** (Suite de Fibonacci)**.** *Une suite de Fibonacci est définie de la manière suivante*

$$U(n) = U(n-1) + U(n-2), \text{ avec } U(0) = x \text{ et } U(1) = y.$$

On doit donc créer deux variables temporaires tempo qui stockera la valeur courante et tempo_2 qui stockera l'avant dernier terme. A l'itération $i$ :
On calcule $U(i)$ avec $U(i-1)$ et $U(i-2)$. On aurait donc tempo qui vaut $U(i-1)$ et tempo_2 qui vaut $U(i-1)$.
Soit res=tempo + tempo_2. Alors res est égal à $U(i)$. Maintenant il faut que tempo prenne la valeur $U(i)$ et tempo_2 prenne $U(i-1)$. Comme cela à la prochaine itération on aura tempo qui vaut $U(i-1)$ et tempo_2 qui vaut $U(i-1)$. Pour cela on fait : tempo_2 = tempo et tempo = $res$.

Attention **l'ordre** est important.

**Une petite question** Pourquoi si à la $i$ème itération tempo prend la valeur $U(i)$ alors à l'itération $i+1$ il vaudra $U(i-1)$ ?

**Le code associé**

```
#include <stdio.h>

main(){
  int tempo, tempo_2;
  int i;
  int nb_iter;
  int res;

  nb_iter=15;

  scanf(%d",&tempo);
  scanf(%d",&tempo_2);

  for(i=3;i<(n+1);i++={
    res=tempo+tempo_2;
    tempo_2=tempo;
    tempo=res;
  }
}
```

## 5.4 Equivalence entre accumulateur et suite récurrente

Le principe de l'accumulateur vu au début de ce chapitre est en fait souvent très similaire aux principes des suite récurrentes. Nous allons montrer, ici, que l'exemple du maximum des termes d'un tableau peut se modéliser sous la forme d'une suite récurrente (parfois appelée suite itérative).

**Rappel** On veut calculer le maximum des éléments d'un tableau dont la taille est taille.

Description de l'algorithme :
Accumulateur prend comme valeur la valeur du premier terme du tableau.
de $i$ de 2 à taille **faire**
si tableau[i] > max alors max<-tableau[i] sinon ne rien faire.

Code :

```
#include<stdio.h>

main(){
  const taille = X;
  int accumulateur;
  int i;
```

```
  int tab[taille];

  // initialisation du tableau

  accumulateur=tab[0];
  for(i=0;i<taille-1;i++){
    if (tab[i+1]>accumulateur) {
                acumulateur=tab[i];
    }
  }
  printf("valeur finale le max est %d",accumulateur);
}
```

**Modélisation sous forme d'une suite récurrente** On a, à l'étape $i$ :
$\max$(accumulateur,$i$ème terme du tableau).

On suppose que les valeurs de l'accumulateur, à chaque étape, sont les valeurs des termes d'une suite récurrente. On peut définir alors une suite $V$ telle que $V(i)$ soit la valeur de l'accumulateur à l'étape $i$. On aurait :
Valeur de l'accumulateur en $0 = V(0)$,
Valeur de l'accumulateur en $1 = V(1)$,
...
Valeur de l'accumulateur en $k = V(k)$
...

La formule de calcul de $V(i)$ est alors

$$V(i) = \max(V(i-1), tab[i]). \tag{5.4.1}$$

### 5.4.1 Généralisation

La transformation de l'exemple ci-dessus peut se généraliser très facilement dès lors que le calcul de l'accumulateur dépend uniquement du terme précédent et d'une suite de calculs. A contrario, le calcul d'une suite récurrente (ou itérative) est en fait un accumulateur. La valeur du terme de la suite étant la valeur de l'accumulateur.

A titre d'exercice, il vous est laissé faire la transformation de l'exemple 1 de ce chapitre.

## 5.5 Invariant de boucle

La question qu'on se pose maintenant tient sous deux formes :

1. Etant donné un problème **P** comment vérifier que la suite de calculs effectués est correcte et donne le bon résultat.

2. Comment ce problème peut-il se modéliser sous la forme d'un accumulateur (pris ici au sens accumulateur et suite itérative) ?

### 5.5.1 Raisonnement par récurrence

Tout d'abord un petit rappel sur les raisonnements par récurrence. Un raisonnement par récurrence pour montrer une propriété **A**(i) est de la forme

1. Montrer que **A**(1) est vraie (directement).
2. Montrer, si nécessaire, que les propriétés **A**(2),...,**A**(k) sont vraies (avec $k \in IN$).
3. En supposant que la propriété est vraie jusqu'à l'ordre $n$, Montrer qu'on a alors **A**(n+1).

**Exemple 5.5.1** (Somme des $n$ premiers nombres)**.** *On veut montrer que la somme des $n$ premiers entiers naturels est $\frac{n(n+1)}{2}$.*

*Preuve :*

1. *On a somme du premier entier qui vaut $1$ et $(1 \times 2)/2 = 1$. Donc l'assertion est vraie à l'ordre $1$.*
2. *Supposons que l'assertion soit vraie jusqu'à l'ordre $n$. Montrons que l'assertion est vraie à l'ordre $n+1$. On a $\sum_{n=1}^{n+1} = \sum_{n=1}^{n} + (n+1)$. Ceci vaut donc en utilisant l'hypothèse de récurrence $\sum_{n=1}^{n+1} = \frac{n(n+1)}{2} + (n+1)$, il s'ensuit $\sum_{n=1}^{n+1} = \frac{n(n+1)+2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$*

*Le résultat, est prouvé.*

### 5.5.2 Invariant de boucle et accumulateur

De la même manière, pour montrer la correction du calcul d'une suite itérative on va définir un mécanisme très proche du raisonnement par récurrence : *l'invariant de boucle*. Ce concept a été utilisé tout au long des deux chapitres précédents sans être formellement identifié.

**Définition 5.5.2** (Invariant de boucle)**.** On appelle invariant de boucle une propriété qui reste vraie à chaque passage dans la boucle. Plus précisément à chaque passage dans un point donné de la boucle qui le plus souvent est la fin ou le début.

Cet invariant a pour but d'assurer la correction du programme. En assurant que, à la fin du programme, la propriété voulue est satisfaite.

La démarche est la suivante :

1. Vérifier que la propriété est vraie au début de l'algorithme.
2. Vérifier par récurrence que l'invariant de boucle (la propriété) reste vrai après chaque boucle

Le plus compliqué restant la plupart du temps la recherche de cette propriété invariante. En effet, définir un invariant qui assure qu'à la fin du programme la propriété que l'on cherche sera bien vérifiée n'est pas immédiat.

**Exemple**

Appliquons cela au calcul du maximum des termes d'un tableau. On veut calculer le maximum des termes d'un tableau. Ce qu'on veut c'est trouver la propriété qui assure que, à la fin de l'algorithme, le résultat renvoyé sera bien le plus grand terme du tableau.

**Recherche de l'invariant de boucle** Je définis tout d'abord le terme $U$ calculé à chaque boucle par : $U = \max(U, tab[\cdot])$. On essaye (mais ici cela va marcher) comme invariant de boucle et comme point de vérification le *prédicat* : "*A la fin de la $n$ème étape de la boucle le terme $U$ est le maximum des $n$ premiers termes du tableau*". C'est cette affirmation que je vais prendre comme propriété de l'invariant de boucle.

Vérifions qu'elle satisfait les conditions requises pour assurer le bon résultat.

A l'étape $1$ on a $U = tab[1]$. (Equivalent à $\max(tab[1], U)$ avec $U = -\infty$). Le terme $U$ est bien le maximum du premier terme du tableau.

Supposons que nous soyons à l'étape $n$ et que $U$ est le maximum des $n-1$ premiers termes du tableau. Comme $U = \max(U, tab[n])$ alors à la fin de la boucle $U$ est le maximum des $n$ premiers termes du tableau.

Par récurrence on saura que à la fin du calcul le résultat trouvé sera le maximum des termes du tableau.

*Remarque* 5.5.3. Ceci permet de vérifier que le calcul effectué avec un accumulateur est correct. Cela permet aussi en sachant cela d'obtenir une ligne directrice (grosso modo : "*trouver quelque chose qui bouge pas*") lorsqu'on cherche à trouver un algorithme en vu de résoudre un problème.

**Application au suites itératives** Dans ce cas la propriété à vérifier est beaucoup plus naturelle il s'agit de calculer un terme et de vérifier qu'il est bien au début de chaque boucle le terme de la suite correspondant.

**Généralisation** Cette méthode ne se restreint pas au calcul avec accumulateur ou des suite récurrentes mais bien plus largement à l'algorithmique en général.

# Deuxième partie

# Semestre 2

# Chapitre 6

# Les fonctions

## 6.1 Introduction

En langage C comme dans beaucoup d'autres Langages informatique on peut découper un programme en plusieurs parties, ces parties sont appelées *fonctions*. Une seule fonction en C existe obligatoirement le main. Cette fonction va pouvoir utiliser (on dit *appeler*) les autres fonctions (les autres parties de code) pour réaliser son but.

### 6.1.1 But d'une fonction

On peut se demander pourquoi morceler comme cela un programme.
Bien que cela ne semble pas être apparent. Cela est fait pour différentes raisons.

- Division du programme en sous parties exécutant une action simple (Programme plus structuré).
- Le code est (avec de l'habitude) plus lisible.
- Cela facilite le déboguage.
- On évite de changer tout en programme quand on fait des changements : on ne modifie que la partie incriminée.
- Volonté de développer certaines parties spécifiquement pour être utilisée au court du programme. On regroupe la suite d'instructions pour former une action globale.
- Éviter de réécrire 15 fois la même chose pour refaire la même action.

La sous partie qui exécute une action logique unique (l'action logique) va être appelée par son identificateur et manipulée soit par une autre fonction soit par le main.

**Fonctions déjà connues** Il y a un certains nombre de fonctions que vous manipulez déjà.

- printf, scanf, cos

**Exemple de fonction** Citons quelques d'exemples d'applications qui peuvent être réalisées par une fonction (pas forcément en C).

- Fonction de lecture d'un morceau MP3 (même action sur différents contenus).

- Fonction d'affichage d'un menu.
- Fonction de test de cohérence d'une donnée entrée
- Fonction de calcul d'un écart type, sur des ensembles de données
- Fonction de calcul des valeurs d'une suite
- etc...

**En résumé**

**Définition 6.1.1.** Une fonction est une partie d'un code source qui permet de :
- réaliser la même action logique plusieurs fois
ou
- réaliser la même action logique sur des objets différents.

Les instructions d'une fonction sont définies (en langage C) à l'aide de blocs d'instructions.

## 6.2 Fonctions non paramétrées

On s'intéresse dans un premier temps à des fonctions dites non paramétrées [1].

### 6.2.1 Définition d'une fonction

Comment est définie une fonction ? Une définition de fonction, contient : une *interface* (ou *en-tête* ou encore *prototype* nous verrons la différence plus tard) et un corps de fonction.

Cette définition est sous deux formes

- l'en-tête permet de dire comment la fonction va être utilisée. Les paramètres, au cas échéant, dont elle a besoin et de donner une indication sur le type de son résultat.
- Le corps de la fonction, lui, contient la description de la suite d'instructions que l'on veut faire effectuer à l'ordinateur.

**Les en-têtes de fonctions non paramétrées**

C'est la première ligne de la définition de la fonction. Elle se présente sous la forme :
type nom_fonction() .

Dans cet en-tête, il y a deux éléments :
- Le nom de la fonction : son identificateur qui va permettre de l'identifier vis-à-vis des autres parties du programme. L'appel de la fonction va utiliser ce nom.

- Le type désigne le type de la fonction c'est à dire le type de la valeur qu'elle retourne (car une fonction retourne ou renvoie une valeur).

[1] Attention en C contrairement à d'autres langages il n'existe que des fonctions pas de procédures

**Les types de valeurs retournées** Le type de la fonction est le type de la valeur qu'elle retourne (car une fonction retourne ou renvoie une valeur). Une fonction qui ne renvoie pas de valeur (cela existe aussi) est une fonction dont le type de retour est spécifié par le mot clef void.

Une fonction comme par exemple cos(x) retourne ou renvoie une valeur (pour celle de la bibliothèque math.h cette valeur est de type double). C'est à dire qu'elle transmet cette valeur à la fin de son déroulement au sous-programme appelant. Dans le cas où une valeur est retournée le sous programme appelant devra gérer (et savoir comment gérer) cette valeur récupérée.

### Le corps d'une fonction

Le corps de la fonction est l'ensemble des instructions que l'ont veut faire faire à l'ordinateur. On a parfois besoin d'utiliser des variables propres à la fonction, en ce cas le corps de la fonction débute par la déclaration de ces variables. Il est suivi par une série d'instructions.

Enfin, le corps d'une fonction se termine par l'instruction de retour l'instruction "return" de syntaxe "return(expression) ;". Avec
- valeur de expression valant la valeur que retourne la fonction.
- type de expression devant être identique à celui spécifié dans l'en-tête de la fonction.

Lorsque la fonction ne retourne pas de valeur (type void), le corps de la fonction se termine par return ;.

Lors de l'exécution d'une fonction l'instruction return provoque le retour au programme appelant. Cela signifie qu'aucune autre instruction de la fonction sera exécutée.

*Remarque* 6.2.1. Il peut y avoir plusieurs instructions return dans la déclaration d'une fonction. Mais c'est la première instructions return rencontrée qui provoquera l'arrêt de la fonction et le retour au programme appelant. Ainsi, si on a

```
int fonction(){
  int i;
  i=0;
  return i;
  i=i+1;
  return i+1;
```

la valeur retournée sera 0 et non 1.

### Exemple

On veut une fonction qui retourne un nombre saisi au clavier par l'utilisateur. Ce nombre devant être compris entre 0 et 10 et on veut afficher l'écart entre le nombre saisi et le milieu de l'intervalle. Le code de la fonction doit être.

```
int saisie_nombre(){
      int ecart, nombre;
      printf("Saisissez un nombre entre 1 et 10");
      scanf("%d",&nombre);
```

```
      while ( (nombre >10) || (nombre <0) ){
            ecart=abs(nombre-5);  // utilisation de la fonction abs
            printf("tu t'écartes de %d",ecart);
            printf("Saisissez un nombre entre 1 et 10");
            scanf("%d",&nombre);
      }
return nombre;
} // fin de la fonction
```

**Déclaration d'une fonction**

En C, il faut **déclarer** toute fonction avant de pouvoir l'utiliser. La déclaration informe le compilateur du type des paramètres et du résultat de la fonction et lui permet de vérifier que le nombre et le type des paramètres utilisés dans la définition ou lors d'un appel concordent bien avec le prototype donné. Grâce à cela, le compilateur peut contrôler si l'utilisation d'une fonction est correcte et même possible. La fonction principale main n'a pas besoin d'être déclarée.

La déclaration d'une fonction se fait par l'écriture du **prototype** de la fonction. Ainsi

```
type nom_fonction ()
```

**Règle de programmation 6.2.2.** *Le compilateur doit avoir connaissance de la fonction au moment où celle-ci est utilisée. Comme cette connaissance est faite à l'aide d'une déclaration alors la déclaration doit figurer avant tout autre utilisation de la fonction.*

*Une définition comprend une déclaration. C'est à dire qu'il n'est pas nécessaire de déclarer une fonction déjà définie.*

**L'appel d'une fonction**

L'appel de la fonction correspond à la demande d'utilisation de la fonction par le programme.

L'appel se fait par le nom de la fonction. Ainsi, l'instruction saisie_nombre() ; correspond à un *appel* de la fonction *saisie_nombre* et à son utilisation. par le composant appelant.

Il est parfaitement possible d'appeler une fonction à l'intérieur d'une autre fonction.

**Exemple 6.2.3.**
```
#include <stdio.h>
//declaration
int saisie_nombre();

// definition
int saisie_nombre(){
      int ecart, nombre;
      printf("Saisissez un nombre entre 1 et 10");
      scanf("%d",&nombre);
      ecart=abs(nombre-5);  // utilisation de la fonction abs
      printf("tu t'écartes de %d",ecart);
```

```
      while ( (nombre >10) || (nombre <0) ){
            ecart=abs(nombre-5);  // utilisation de la fonction abs
            printf("tu t'écartes de %d",ecart);
            printf("Saisissez un nombre entre 1 et 10");
            scanf("%d",&nombre);
      }
return nombre;
} // fin de la fonction

main() {
  int recepteur_resultat_saisie;
  // appel de la fonction
  recepteur_resultat_saisie=saisie_nombre();
  // fin du déroulement de la fonction
  printf("Le nombre saisi est %d",recepteur_resultat_saisie);
}
```

**Attention**, l'appel d'une fonction est considéré comme une seule instruction. Ainsi, dans le programme appelant, tant que les instructions de la fonction ne sont pas achevées l'ordinateur ne passe pas à l'instruction suivante du programme appelant.

**Exemple 6.2.4.** *Par exemple le programme suivant*

```
#include <stdio.h>
void affiche()

void affiche(){
int a;
a=0;
       printf(" Je commence \n");
       for(i=0;i<1000000;i++){
           a=a+1;
       }
       printf(" Je finis");
}

main(){
    printf("C est la fin des haricots\n");
    affiche();
    printf("la fonction est finie \n")
}
```

*Donnera à l'exécution*

```
C est la fin des haricots
```

```
Je commence
Je finis
la fonction est finie
```

### 6.2.2 Le cas spécifique du main

Comme nous l'avons déjà dit, le main est une fonction particulière. C'est cette fonction qui est appelée lorsqu'on exécute un programme. Le main peut renvoyer une variable qui sera un entier. Cet entier sera transmis au programme appelant et indique le bon déroulement du programme ou non (cela doit être géré au moment de la conception du code). Enfin le main peut prendre des arguments qui seront transmis par la ligne de commande au programme quand on l'appelle.

## 6.3 Fonctions paramétrées

Les paramètres d'une fonction (ou *arguments*) peuvent être vus comme un ensemble de données que l'on doit traiter. La définition d'une fonction permet de décrire le traitement et le déroulement de la fonction en fonction des valeurs prises par les paramètres. Les valeurs des paramètres sont données lors de l'appel.

On peut imaginer le déroulement d'une fonction comme le travail à faire par un manutentionnaire dans un entrepôt de pièces détachées pour voiture. Les paquets de rétroviseurs sont livrés par un camion. Cela peut être des rétroviseurs extérieurs ou des rétroviseurs intérieurs. Son travail consiste à prendre un paquet dans le camion (sans le faire tomber), regarder le type de rétroviseur.
- Si c'est un rétroviseur intérieur alors il doit enlever la gaine de protection et le ranger
si c'est un rétroviseur extérieur il doit juste l'entreposer.

Le travail et la description du travail du manutentionnaire est le corps de la fonction. Le paramètre est le carton (de type carton de rétroviseurs xx). En fonction des différents contenus du carton de rétroviseurs le travail du manutentionnaire ne sera pas le même mais tous les travaux pour tous les contenus de cartons doivent être définis. L'appel d'une fonction correspond à l'arrivée d'un carton, la valeur du carton correspondant à la classe de rétroviseur qu'il contient.

Lors de la définition les paramètres utilisés sont donc appelés *paramètres formels* ou *paramètres muets*. Puisqu'ils *formalisent* et *définissent* le déroulement de la fonction sans pour autant avoir une valeur réelle. Tandis que lors de l'appel, les paramètres réellement utilisés qui vont avoir une valeur sont les *paramètres effectifs* ou *valeurs réelles*.

Illustrons ceci, lorsqu'en mathématiques vous définissez

$$f(x) = x^2 + 2x + 1.$$

$x$ est un paramètre formel et quand vous utilisez $f(5)$ $5$ est la paramètre effectif.

**Attention** Un paramètre effectif peut être une variable en informatique. Dans ce cas c'est la valeur de la variable qui est utilisée comme valeur pour le paramètre réel.

**Prototype d'une fonction** Le prototype est de la forme

```
type nom_fonction (typeparam1, typeparam2, ... )
```

ou bien

```
type nom_fonction (typeparam1 nomparam1, typeparam1 nomparam2, ... )
```

et indique uniquement le type des données renvoyées et reçues par la fonction.

**Exemple 6.3.1.** *Reprenons l'exemple du manutentionnaire de tout à l'heure. Illustrons son travail par une fonction.*

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

// premiere fonction traitement par le manutentionnaire

void manutentionnaire(char carton){
    if (carton=='e'){
        printf("C'est un retro exterieur je vais le ranger");
    }
    else {
        printf("C'est un retro interieur");
       printf(" je le nettoie et  je vais le ranger");
    }
   return;
}

// Creation des cartons
char camion(){
    int u
    srand(0);
    // je tire une variable aléatoire entre 0 et 1
    u= (int) ((1.0*rand())/RAND_MAX);
   if (u < 0.5) return 'e'; else return 'i';
}

main(){
   char c;
   srand(time(NULL));
   c=camion();
   printf("c vaut %c\n",c);
   manutentionnaire(c);
```

```
   manutentionnaire('e');
   manutentionnaire('i');
}
```

**Commentaires**

## 6.4 Visibilité des variables

On peut se demander quelle est la *portée* d'une variable : une variable déclarée dans une fonction peut-elle être utilisée dans une autre partie du programme ? On peut se demander quelle est sa "durée de vie". Si j'appelle plusieurs fois la même fonction et que la variable change de valeur ou si la variable est utilisée dans une autre fonction.

### 6.4.1 Variables globales

**Définition 6.4.1.** Une variable globale est une variable déclarée en dehors de toute fonction (même le main).

**Exemple 6.4.2.** *Dans l'exemple suivant la variable* note *est une variable globale.*

```
#include <stdio.h>

char note;
note='l';

void fonction();

void fonction()
{
  note='s';
  printf("la note est %c\n",note);
  return;
}

main()
{
    printf("la note est en dehors %c\n",note);
    fonction();
}
```

**portée** Une variable globale est reconnue par le compilateur dans toute le code après sa déclaration.

### 6.4.2 Variables locales

**Définition 6.4.3.** Une variable locale est une variable déclarée à l'intérieur d'une fonction. Par défaut, les variables locales sont visibles uniquement à l'intérieur de la fonction[2] dans laquelle elles sont déclarées. Elles ne gardent pas, par défaut, leurs valeurs d'un appel à l'autre.

**Exemple 6.4.4.** *Dans l'exemple suivant la variable* note *est une variable locale.*

```
#include <stdio.h>


void fonction()
{
  char note;
  note='l';
  printf("Week End à Rome la note est la note est %c\n",note);
  note='s';
  return;
}

main()
{
    fonction();
    fonction();
}
```

*Donne à l'affichage*

```
Week End à Rome la note est la note est l
Week End à Rome la note est la note est l
```

#### Durée de vie

A la sortie de la fonction, les variables locales sont détruites et leur valeur perdues.

#### Exemple de multiple déclaration d'une variable du même nom

Les variables locales n'ont aucun lien avec des variables globales de même nom ni même avec des variables locales d'une autre fonction.

Par exemple,

**Exemple 6.4.5.**
```
#include <stdio.h>
char note1;
char note2;
```

[2] et des fonctions définies à l'intérieur de cette fonction

```
note2='s';
note1='s';

void fonction()
{
    char note3;
    char note2;
    note1='l';
    note2='l';
    note3='l';
    printf("Ds f note 1 est %c\n",note1);
    printf("Ds f note 2 est %c\n",note2);
    printf("Ds f note 3 est %c\n",note3);
    return;
}

main()
{
    char note3;
    note3='r';
    printf("note 1 est %c\n",note1);
    printf("note 2 est %c\n",note2);
    printf("note 3 est %c\n",note3);
    fonction();
    printf("note 1 est %c\n",note1);
    printf("note 2 est %c\n",note2);
    printf("note 3 est %c\n",note3);
}
```

*Donne à l'affichage*

```
note 1 est s
note 2 est s
note 3 est r
Ds f note 1 est l
Ds f note 2 est l
Ds f note 3 est l
note 1 est l
note 2 est s
note 3 est r
```

**Explication**

## 6.5 Passage des paramètres

On s'intéresse au passage des valeurs des paramètres. C'est à dire à la transmission de la valeur d'un des paramètres à la fonction.

### 6.5.1 Passage par valeur

Un paramètre de fonction est assimilé à une variable locale. C'est la valeur du paramètre effectif (ou réel) qui est copiée dans cette "variable locale", cette variable locale étant utilisée dans le corps de la fonction. Lorsque le paramètre effectif est une variable, c'est son contenu (sa valeur) qui est copié dans la variable locale et non la variable elle même. On remarque que la fonction ne travaille que sur une copie qui va être supprimée à la fin de la fonction. C'est pourquoi une modification de la variable locale dans la fonction ne modifie qu'une copie et non la variable passée comme paramètre effectif qui elle reste inchangée.

**Définition 6.5.1** (Passage des paramètres par valeur)**.** On appelle passage par valeur, quand seule la valeur d'un paramètre effectif est connue de la fonction puisque c'est la valeur qui est transmise et copiée.

Le C ne permet de faire que des passages par valeur.

#### Non modification des valeurs

Soit la fonction

```
void echange (int a, int b)
{
  int temporaire;
  printf("Au debut a = %d \t et b = %d\n",a,b);
  temporaire = a;
  a = b;
  b = temporaire;
  printf("A la fin  a = %d \t et  b = %d\n",a,b);
  return;
}

main()
{
  int a = 6, b = 1;
  printf("debut programme a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme a = %d \t b = %d\n",a,b);
}
```

Ceci donnera à l'exécution

```
debut programme a = 6      b = 1
Au debut a = 6     et b = 1
A la fin a = 1     et b = 6
fin programme a = 6      b = 1
```

Cela vous semble surprenant ? Mais que pensez vous de l'effet de echange(6,1) ?

Observons alors le déroulement de cette fonction et le comportement (illustratif) de la mémoire du programme précédent.

**Déroulement de la fonction et effet sur la mémoire**

### 6.5.2 Passage des paramètres par adresse

Une autre manière de transmettre les arguments effectifs est de transmettre non plus leur valeur mais leur adresse (leur adresse mémoire). La fonction ne travaille plus sur une copie de l'objet transmis, mais sur l'objet lui-même (puisque elle en connaît l'adresse). Le paramètre effectif est alors l'adresse de la variable.

**Définition 6.5.2** (Passage par adresse)**.** On appelle passage par adresse, quand le paramètre effectif qui est transmis à la fonction lors de l'appel est l'adresse d'une variable.

La fonction appelée, range alors l'adresse transmise dans un paramètre formel approprié (de type adresse) qui est une variable locale à la fonction appelée. Cette fonction a maintenant accès, via ce paramètre, à la variable de la fonction appelante Le passage par adresse permet donc à une fonction de modifier les valeurs des variables du programme appelant.

*Remarque* 6.5.3. Attention cette définition du passage par adresse est générique mais ne s'applique pas stricto sensu au Langage C. En effet, il n'y as pas de passage par valeur en C (comme en Pascal par exemple). Pour pouvoir avoir un équivalent au passage par adresse en C on utilise les pointeurs. On va utiliser une astuce qui consiste à passer par valeur un paramètre qui est en fait une adresse. Plus de précision dans un prochain cours mais il s'avère qu'on passe un pointeur vers une adresse mais le pointeur en paramètres est passé lui par valeur (vous suivez ?)

```
void echange2 (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
}

main()
{
```

```
  int a = 2, b = 5;
  printf("debut : a = %d et b = %d\n",a,b);
  echange(&a,&b);
  printf("fin : a = %d et b = %d\n",a,b);
}
```

### Commentaires

**Règle de programmation 6.5.4** (Quand utiliser le passage par valeur ou le passage par adresse)**.** *Le passage par valeur s'utilise quand on a pas besoin de faire de modifications sur des variables, tandis que un passage par adresse est utilisé de préférence lorsque l'on doit effectuer des modifications qui doivent être répercutées ensuite dans tout le déroulement du programme.*

D'autre part, mais cela sera vu ultérieurement, on peut (on doit ? ?) parfois utiliser le passage par adresse même si il n'y a pas de modifications. C'est le cas notamment du passage de tableaux en paramètres.

### Passage de paramètres pour les tableaux

Attention, les tableaux ne peuvent être passé QUE par adresse. Plus spécifiquement, on donne la référence (c'est à dire un pointeur) d'une case du tableau (leplus généralement la première case).

Ainsi si on a déclaré un tableau de nom tab, alors tab, &tab[0] et &tab ont la même valeur. C'est à dire qu'ils donne la même adresse qui est la première case du tableau.

Il y a deux manières equivalentes dans le résultat pour déclarer un tableau en paramètres.

### Premier type de déclaration

```
#include <stdlib.h>

void lire_tableau(int tableau[], int taille)
{
  int i;
  for (i = 0; i < taille; i++){
              printf("La %d entree est %d'',tableau[i]);
  }
  return;
}

main()
{
  const N= 10;
  int tab[N];
  tableau(tab[],N);
}
```

De même une fonction ne peut pas renvoyer un tableau directement mais une adresse sur une zone mémoire.

**Second type de déclaration**

```
#include <stdlib.h>

void initialise_tableau(int *tableau, int taille)
{
  int i;
  for (i = 0; i < taille; i++){
              printf("Entrer la %d eme coordonnées",i);
              scanf("%d",&tableau[i]);
  }
  return;
}

main()
{
  const N= 10;
  int tab[N];
  initialise_tableau(tab,N);
}
```

# Chapitre 7

# Les Pointeurs

## 7.1 Introduction

On rappelle qu'une variable que l'on manipule dans un programme doit être stockée dans la mémoire vive de l'ordinateur. La déclaration d'une variable est le fait de réserver un espace mémoire pour cette variable. Ainsi, il y aura correspondance entre une variable que l'on manipule dans le programme (que l'on décrit le plus souvent par son nom ou identificateur) et l'espace réservé dans la mémoire que le système d'exploitation manipule.

Il peut être cependant plus intéressant de décrire une variable non plus par son identificateur (c'est à dire telle quelle) mais plutôt en se donnant la possibilité de manipuler son espace mémoire comme on l'a vu au chapitre précédent.

## 7.2 Adresse Mémoire

Très approximativement, la mémoire de l'ordinateur peut être vue comme une suite de cases mémoires consécutives (un tableau). Ces cases mémoire ont une adresse (qui est l'équivalent des indices du tableau) qui permet de les identifier de manière unique. Ces cases peuvent être manipulées soit individuellement soit par groupe. Ce, selon la taille du type que l'on manipule car l'espace réservé pourra être alors de une ou plusieurs cases mémoires. C'est l'adresse de la case mémoire qui assure la correspondance entre la variable que l'on manipule dans le programme et la mémoire.

Le contenu de ces cases mémoires va être le contenu de la variable et vice versa.

Ainsi une variable va être définie par deux caractéristiques

- Son adresse (ou l'adresse de sa première case mémoire).
- Son contenu qui donne sa valeur.

A quoi correspond alors une affectation ? Une affectation c'est la recopie du contenu d'une variable dans le contenu d'une autre variable. Il n'y a pas de copie d'adresses.

### 7.2.1 Opérateur adresse

Pour connaître la valeur de l'adresse d'une variable on utilise l'opérateur (unaire) &. Par exemple &i renvoie la valeur de l'adresse de la variable i. L'opérateur & ne permet que de récupérer l'adresse d'un objet en mémoire : variable, case de tableau et ne peut s'appliquer à des expressions ou des constantes.

Cet opérateur a déjà été aperçu auparavant, par exemple dans scanf("%d",&a).

### 7.2.2 (Non) Manipulations des adresses mémoire

Si les adresses mémoires sont des entiers (de format interne un peu particulier dépendant de l'architecture 8, 16, 32, ou 64 bits on parle alors d'adressage $x$ bits), on ne peut cependant pas manipuler les adresse mémoires comme des variables. On ne peut donc tenter d'affecter une adresse ainsi, &a=123569 n'a aucun sens. En effet, il faut voir l'adresse comme une constante associée à la variable.

Par contre on peut utiliser & pour indiquer l'adresse d'une variable. C'est ce que fait l'appel à la fonction scanf("%d",&a) qui va utiliser la valeur indiquée par la "constante" &a pour placer le contenu codant l'entier lu au clavier dans l'espace mémoire indiqué par &a.

## 7.3 Les pointeurs

Comme on ne peut manipuler des adresses directement par l'utilisation de l'op rateur (unaire) &, pour pouvoir manipuler des variables via leur adresse on utilise un autre moyen *les pointeurs*. Les pointeurs sont dédiés à la manipulation des variables par le biais de leur adresse.

Un pointeur est un élément de valeur égale à l'adresse d'un élément du programme. Cet élément peut être un type classique int, float, double, char ou encore une case d'un tableau ou encore un pointeur.

Attention, comme nous l'avons déjà dit, si une adresse est un entier la manipulation d'une adresse n'est pas similaire à la manipulation d'un entier. C'est pourquoi le pointeur dépend du type de l'objet vers lequel il pointe. Cette différence provient du fait que selon le type de donnée la valeur de l'adresse renvoyée est soit l'adresse de l'octet où cette donnée est stockée soit l'adresse du premier des $x$ octets où cette donnée est stockée.

### 7.3.1 Manipulation de pointeurs

Regardons comment on peut manipuler des pointeurs.

#### Déclaration

Comme pour toute variable un pointeur doit être déclaré. On déclare un pointeur à l'aide de l'instruction :

```
type *nom_pointeur ;
```

où type est le type de l'objet pointé.

Ceci déclare une donnée pointeur dont l'identificateur est nom_pointeur, dont la valeur est l'adresse d'une variable de type type. On dit dans ce cas là que nom_pointeur pointe sur la variable.

Le nom nom_pointeur est donc une sorte de variable d'adresse. Qui a donc elle-même une adresse et un contenu. La valeur (le contenu) de nom_pointeur est modifiable. Ceci signifie que l'adresse est modifiable et qu'une fois modifiée on pointe sur un autre élément.

Attention il faut bien voir aussi que le contenu de la variable *pointée* par nom_pointeur peut lui aussi être modifié.

### Affectation

On s'intéresse ici à l'affectation et à la modification du contenu du pointeur (le changement d'adresse).

**Affectation** L'affectation d'un pointeur consiste à lui donner une adresse. cette adresse est obtenue par l'opérateur $\&$. Ainsi dans le code suivant on déclare une variable entier à laquelle on affecte la valeur $1$ et une variable pointeur (pointeur sur un entier) à laquelle on affecte l'adresse de entier.

```
int entier;
int *pointeur;

entier=1;
pointeur = &entier;
```

A la fin de ces instructions les adresses et contenu des variables sont :

- entier a pour *valeur* $1$ et pour adresse $X$ (où $X$ est un entier décrivant l'adresse).
- pointeur a pour *valeur* $X$ et pour adresse $Y$ (où $Y$ est un entier qui donne l'adresse de pointeur.

La modification peut se faire (entre autre) en utilisant à nouveau l'opérateur $\&$ ou encore en copiant le contenu d'une autre variable de type pointeur.

**Exemple 7.3.1.** *Soit le programme suivant*

```
main()
{
  int entier1;
  int entier2;
  int *pointeur1;
  int *pointeur2;

  entier1= 1;
  entier2= 2;
```

```
  pointeur1 = &entier1;
  pointeur2 = &entier2;

  pointeur1 = pointeur2;
}
```

*A titre d'exercice vous pouvez listez les différents contenus des variables du programme au cours de son déroulement.*

**Modification de la valeur pointée**

On s'intéresse ici à la modification du contenu de la variable pointée c'est à dire au changement du contenu de la valeur de la variable dont l'adresse est le contenu de la variable pointeur.

Observons le schéma relatif.

Tout d'abord il faut connaître l'instruction qui va permettre d'accéder au contenu de la variable pointée. C'est l'opérateur (*unaire* ici) $*$ qui permet d'accéder directement à la valeur de l'objet pointé.

Soit pointeur un pointeur vers un caractère c, *pointeur désigne la valeur de c. Si on reprend la suite d'instructions ci-dessus adaptée aux caractères.

```
  char c = 'e';
  char *pointeur;

  pointeur = &c;
  *pointeur='d';
```

Donnera à la fin du programme

- c a pour *valeur* 'd' et pour adresse $X$ (où $X$ est un entier décrivant l'adresse).
- pointeur a pour *valeur* $X$ et pour adresse $Y$ (où $Y$ est un entier qui donne l'adresse de pointeur.

Quelles différences y a-t-il entre le programme suivant et le programme de l'exemple 7.3.1 ?

```
main()
{
  int entier1;
  int entier2;
  int *pointeur1;
  int *pointeur2;

  entier1= 1;
  entier2= 2;

  pointeur1 = &entier1;
```

```
  pointeur2 = &entier2;

  *pointeur1 = *pointeur2;
}
```

**Lien avec le passage par adresse**

On peut remarquer en reprenant le passage consacré au passage de paramètres par adresse du chapitre précédent sur les fonctions que le passage par adresse utilise en fait des pointeurs.

Ainsi, la fonction echange2

```
void echange2 (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
}
```

prend comme paramètres deux pointeurs (les pointeurs étant passés par valeur) et la manipulation effectuée consiste en l'échange des valeurs pointées.

# Chapitre 8

# Structure de données 2

Ce chapitre est la suite du chapitre sur le même sujet donné au premier semestre.

## 8.1 Les tableaux à deux dimensions

On peut étendre les tableaux à une dimension et utiliser des tableaux à plusieurs dimensions. Ainsi on peut voir un tableau à deux dimensions comme :

- Un ensemble de cases mémoires ordonnées à la fois sur des lignes et des colonnes.
- Un ensemble de cases mémoires indicées par deux indices l'un désignant la ligne, l'autre désignant la colonne.
- Un tableaux à une dimension (les lignes) dont chaque case mémoire est en fait un tableau (les colonnes) à une dimension.

*Schema* 8.1.1.

Un tableau à deux dimensions se déclare par type nomtableau[nblig][nbcol]. Ainsi,

```
const L=5;
const C=10;

int tableau[L][C];
```

déclare un tableau d'entiers de $5$ lignes $10$ colonnes.

L'accès à la case d'indice de ligne $i$ et d'indice de colonne $j$ se fait par tab[i][j].

**Exemple 8.1.2** (Fonction d'initialisation d'un tableau à deux dimensions)**.** *Le code de la fonction d'initialisation d'un tableau à deux dimensions est*

```
#include <stdio.h>

void initialisation2d(int tab[2][2]);

void initialisation2d(int tab[2][2]){
```

```
  int i,j;
  for(i=0;i<2;i++){
    for(j=0;j<2;j++){
      printf("Entrez la valeur de tab %d %d",i,j);
      scanf("%d",&tab[i][j]);
    }
  }
  return;
}

main(){

  int tableau[2][2];

  initialisation2d(tableau);

}
```

**Commentaires** Explication de la différence entre les déclaration du tableau ici et déclaration des fonctions manipulant des tableaux à une dimension du chapitre précédent.

**Exercice 8.1.3** (Matrice transposée)**.** *On veut la fonction qui permette de calculer la transposée d'une matrice. On suppose que la matrice donnée sera la matrice transformée. On suppose que l'initialisation est faite au préalable.*

*Mathématiquement cela veut dire que le terme de coordonnées* $(i, j)$ *de la matrice* $M_{i,j} = M^t_{j,i}$. *Cela veut dire qu'il faut échanger les termes 2 à deux. Mais attention on ne doit pas échanger les termes deux fois. Cela veut dire, que l'on doit faire tourner notre algorithme sur la moitié triangulaire supérieure ou la moitié triangulaire inférieure.*

*Le code de cette fonction est :*

```
void transposee(int tab[5][5]){
  int i,j;
  int temp;
  for(i=0;i<5;i++){
    for(j=i+1;j<5;j++){
      /* i+1 car on ne veut qu'une partie */
      temp=tab[i][j];
      tab[i][j]=tab[j][i];
      tab[j][i]=temp;
    }
  }
  return;
```

## 8.2 Les structures

Les tableaux qu'ils soient à une dimension ou non ne peuvent stocker qu'un seul type de variable (des entiers, des caractères, etc...) mais ils ne peuvent stocker à la fois des caractères et des entiers (par exemple). Plus généralement ils ne peuvent stocker des varaiables de type différents. C'est un peu gênant dès lors que l'on veut structurer des données plus complexes. Ainsi, si on veut stocker par exemple une base de données de clients d'une banque avec leur nom et leur solde de compte. On veut donc organiser une donnée qui comporte différent type de valeurs (le nom, le solde).

**Définition 8.2.1.** Une structure est un élément qui regroupe et comporte un ensemble fini de plusieurs variables qui peuvent être de type différents. Chaque élément de la structure est appelé *champ*. Chaque champ est identifié par un identificateur.

Pour bien fonctionner il faut indiquer à l'ordinateur lorsqu'on définit une structure quels sont les champs impliqués, leurs identificateurs et leurs type. Il faut donc définir un caneva (ou modèle) de la structure. On déclare le modèle formel de la structure et non la "variable" correspondant à une "réalisation" de la structure.

Une déclaration de caneva se fait par

```
struct caneva_de_la_structure {
 type champ1;

 type champn;
};
```

Ici le terme caneva_de_la_structure est appelé *étiquette de la structure*. L'étiquette permet d'identifier un modèle de structure.

Ainsi les instructions suivantes permettent de déclarer un modèle de structure appelée (d'étiquette) compte. De plus, le second jeu déclare lui des variables compte1, compte2 et compte3 de type structure struct compte.

```
struct compte {
  char nom;
  int solde;
};

struct compte compte1, compte2, compte3;
```

On s'aperçoit donc qu'une structure définit un type.

**Accès à la valeur des membres** Pour pouvoir accèder à la valeur d'un des membres d'une structure on utilise le . qui associe le nom d'une structure (une variable de type struct) au nom du membre. Ainsi,

```
printf("Monsieur Madame %c a %d sur son compte",compte1.nom,compte1.solde
```

### 8.2.1 Manipulation de structures

La structure étant d'un genre un peu particulier on peut se demander comment les manipuler.

#### Opérations permises

Les seules opérations permises sur une structure sont

1. La copie ou l'affectation. Ce, en considérant la structure dans son ensemble.
2. La récupération de son adresse au moyen de l'opérateur &.
3. L'accès à ses membres.

**Initialisation** Voyons comment initialiser une structure compte définie comme ci-dessus.
En accèdant à ses membres (petit 3).

it - Par une énumération des valeur : struct compte compte1='c',10 ;.

- *En affectant les membres 1 à 1* :

```
struct compte compte1;
compte1.nom='c';
compte1.solde=10;
```

Dans ce dernier cas il est plus facile et plus courant de programmer une fonction d'initialisation qui va renvoyer une structure. Remarque il est possible de renvoyer une structure car une structure définit un type. Il est alors important de voir que ce type doit être connu de la fonction au moment de sa déclaration pour la compilation. Cette fonction prendra en paramètres les différentes valeurs des champs.

```
struct compte creationcompte(char c,int solde){
   struct compte temp;
   temp.nom=c;
   temp.solde=solde;
   return temp;
```

qui sera appelée par compte1=creationcompte('c',10) ;.

**Une petite question** Pourquoi est-ce faisable et que alors que temp va disparaitre ?

*Par recopie d'une structure déjà initialisée* : compte2=compte1.

#### Structure et pointeurs

**Quelques exemples**

Voici, nous présentons ici quelques structures et les manipulations afférentes ce pour illustrer le maniement de tels objets.

On s'intéresse à une structure big_brother qui a pour champs : matricule, numero_maison, et present. on veut définir des fonctions est_present, arrivee et depart qui indiquent respectivement si la personne est présente ou non dans l'appartement et qui modifient la valeur selon la présence ou non de la personne dans son appartement.

le code général du programme est

```
#include <stdio.h>

struct big_brother{
   int matricule;
   int numero_maison;
   int present;
}

struct big_brother initialise(int, int)
int est_present(struct big_brother)
struct big_brother arrivee(struct big_brother)
struct big_brother depart(struct big_brother)

struct big_brother initialise(int mat, int num){
  struct big_brother temp;
  temp.matricule=mat;
  temp.numero_maison=num;
  temp.present=0; // non present au depart
}

int est_present(struct big_brother p){
  return p.present;
}

struct big_brother arrivee(struct big_brother p){
  struct big_brother temp;
  temp.matricule=p.matricule;
  temp.numero_maison=p.numero;
  temp.present=1;
  return temp;
}

struct big_brother depart(struct big_brother p){
  struct big_brother temp;
```

```
  temp.matricule=p.matricule;
  temp.numero_maison=p.numero;
  temp.present=1;
  return temp;
}

main(){
  struct big_brother b;
  b=initialise(0,0);
  // note on pourrait faire un menu
  if (est_present(b)==1)
       printf("La personne est présente dans son appartement");
  b=depart(b);
}
```

Et si je faisais une copie des structures dans depart et arrivee plutôt que celle des champs que se passerait-il ?

**Structure et pointeurs** Tout comme les types int, float ... vus auparavant on peut manipuler des pointeurs sur des structures. Cela peut être très utile dès lors qu'on utilise des structures de tailles importantes. Soit une structure nommée blabla avec des champs x et y. La déclaration d'un pointeur sur cette structure se fait alors par l'instruction struct blabla *pointeur. L'accès à ces membres (ou champs) se fait à l'aide de l'instruction (*pointeur).x. Le parenthèsage est dû uniquement à des histoires de priorités des opérateurs.

Cependant comme il était (est ?) fastidieux de tout taper à chaque fois il existe un raccourcis bien pratique pour accéder à la valeur d'un des membre. Si pointeur est un pointeur sur une structure blabla alors p->x permet d'accéder à la valeur du membre x de blabla.

**Tableaux de structures**

Comme le mot clef struct définit un type alors il est parfaitement possible de créer des tableau stockant des structures du même type.

Cela se fait de la manière suivante

```
struct eleve {
   int module1;
   int module2;
}

struct eleve promotion[35];
```

Récupérer la valeur d'un membre d'un élément du tableau devient promotion[25].module1=20.

### 8.2.2 Structure plus complexes

Les structures que nous avons vu jusqu'à maintenant étaient des structures dont les champs sont plutôt simples. Nous allons voir que les champs des membres peuvent être de type plus complexes.

### 8.2.3 Structures comportant des tableaux

Une structure peut avoir des champs qui sont des tableaux. Ilest donc parfaitement possible de mélanger ces deux types de structures de données.

Une telle structure pourrait être de la forme :

```
struct eleve {
   char nom[100];
   double moyennes[15];
}
```

qui stockerait un élève par son nom (le tableau de caractère) et qui stockerait l'ensemble de ses quinze notes dans un tableau.

L'accès à la 8 eme note d'un élève dont la variable associée est notée par a serait donc effectué par a.moyenne[7].

### 8.2.4 Structure comportant des structures

De la même manière on peut avoir un des membre qui est lui-même une structure. Par exemple, on peut définir une structure client qui définit deux champs identifiant et durée de consommation totale duree et qui comporte une structure détaillant le nombre d'heures gratuites (dans le forfait) et le coût des minutes supplémentaires.

Ceci donnerait,

```
struct forfait{
  int heures_gratuites;
  float cout_minutes;
}

struct client {
  int identifiant;
  float duree;
  struct forfait abonnement;
}
```

# Chapitre 9

# Maniement des fichiers

## 9.1 Introduction

Jusqu'à maintenant nous n'avions vu que des entrées sorties (on peut résumer par entrées sorties l'échange de données entre le programme et l'utilisateur dans le cadre de ce cours) sous la forme d'entrée standard (usuellement le clavier) et de sortie standard (usuellement l'écran). Ce type d'entrée sortie peut facilement atteindre ses limites dès lors que l'on souhaite entrer de nombreux paramètres (par exemple des fichiers de configuration) ou garder une trace des résultats de l'exécution d'un programme. Une façon de gérer cela est d'utiliser des fichiers par l'intermédiaire d'un certain nombre de fonctions du langage C.

La transmission des informations du programme vers le disque dur ou vice versa est en fait le traitement d'un flux de données. Les opérations sur les flux que nous allons voir ici passent par un "buffer" (un espace mémoire qui va servir de réservoir mais nous ne préciserons pas où) géré par le système. Ceci signifie qu'une instruction d'écriture dans le programme ne se traduira pas immédiatement par une écriture sur le disque mais par une "écriture" dans le buffer, avec écriture sur disque quand le buffer est plein. Autrement dit, lors du maniement d'un fichier il y a création d'un espace mémoire dans la mémoire vive et il faut qu'il y ait également une association entre le fichier sur le disque et cet espace mémoire et une association entre l'espace mémoire et le programme. Le programme se chargeant d'écrire (ou de lire) dans le buffer tandis que le système d'exploitation se chargera d'écrire directement sur le disque dur (ou la disquette voire tout autre élément de sauvegarde non temporaire).

Les informations nécessaires à la maintenance des associations : programme $\equiv$ espace mémoire et espace mémoire $\equiv$ disque dur sont décrites dans une structure FILE (dans stdio.h) et, lors du programme, on gère un pointeur sur cette structure. Un fichier sera identifié alors par ce pointeur.

## 9.2 Différents types de fichier

On considère deux types de fichiers : les fichiers *binaires* et les fichiers *textes*.

### 9.2.1 Fichiers textes

Par convention, cette extension correspond aux fichiers formatés au format de texte ASCII. Ainsi, par exemple, (un int binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre).

Du fait de l'existence de variantes (accentuation, ...) ce format ne permet pas une portabilité totale (sur tous les ordinateurs) ASCII mais autorise néanmoins une très large palette de possibilités de portabilité.

### 9.2.2 Fichiers binaires

Ce sont des fichiers qui contiennent caractères, int, double écrits en binaire mais qui ne contiennent pas de chaînes de caractères. Lorsqu'on dit écrit en binaire cela signifie que un double sera stocké comme il est codé en mémoire, d'où gain de place mais surtout incompatibilité entre différents codage des nombres.

## 9.3 Opérations

Nous décrivons maintenant les différentes opérations que nous pouvons effectuer sur les fichiers ainsi que leur rôle.

### 9.3.1 Ouverture

Tout d'abord il faut indiquer au programme de faire le lien entre le fichier écrit sur le disque et la variable que l'on manipule dans le programme. Plus précisément, il faut créer une association entre un flux de données et le fichier sur le disque. Ceci est fait au moyen de la fonction fopen (file open). Son prototype est

```
FILE *fopen(char *nom,char *name)
```

La fonction, renvoie un élément de type FILE* (pointeur sur une structure *FILE) qui est le flux de données et ouvre (lui associe) un fichier dont le nom est décrit en paramètre.

Décrivons ses arguments :

1. Le premier argument est le nom du fichier donné sous la forme d'une chaîne de caractères (*char est un pointeur sur le premier caractère de la chaîne).
2. Le second argument est une chaîne de caractères qui spécifie le mode d'accès au fichier, c'est à dire la manière dont on accède au fichier.

**Les modes d'accès** Les différents modes d'accès décrivent la façon d'accéder aux fichier. On ne s'intéresse qu'aux fichiers textes.

| Mode | Action | Conditions |
|---|---|---|
| "r" | lecture $(read)$ | Le fichier doit exister |
| "w" | écriture $(write)$ | Le fichier va être créé.<br>S'il existe l'ancien fichier sera effacé par le nouveau fichier |
| "a" | écriture à la fin $(append)$ | Les données vont être placées à la fin du fichier.<br>S'il n'existe pas sera créé. |
| "r+" | lecture/écriture | |
| "w+" | lecture/écriture | |
| "a+" | lecture/écriture à la fin | |

**Règle de programmation 9.3.1.** *Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur* NULL. *Il est donc recommandé (comme avec un certain nombre de fonctions renvoyant* NULL*) de toujours tester si la valeur renvoyée par la fonction fopen est égale à* NULL *afin de détecter les erreurs (lecture d'un fichier inexistant...).*

**Exemple 9.3.2.** *Un exemple d'ouverture d'un fichier examen.txt à lire sur la clef USB (lecteur D sous windows).*

```
#include <stdio.h>

main(){
FILE *fichier;

  fichier=fopen(``E:\\examen.txt'',''r'');
  if (fichier == NULL){
     printf(``j'arrive pas à lire'');
  }

}
```

### 9.3.2 Fermeture

La fermeture du flot associé à un fopen est fclose (fclose(flot)). Cette fonction casse la relation qui existe entre le flot et le fichier sur disque. Elle vide (flush) également les tampons. De manière générale ne pas fermer les fichiers peut parfois provoquer des erreurs car selon le système il se peut que le fichier reste ouvert.

### 9.3.3 Lecture Écriture

Les lectures et les écritures dans les fichiers textes se font avec les fonctions fprintf, fscanf qui sont similaires à printf, scanf à l'exception de l'adjonction d'un flot.

```
fprintf(flot,''formatage'', expr_1,...,expr_n)
fscanf(flot,''formatage'',expr1,...,exprn)
```

### 9.3.4 Exemple

On veut lire un fichier de dix entiers (chacun d'eux sur une ligne) et leur rajouter à chacun la moyenne temporaire des valeurs qui sera inscrite dans un deuxième fichier. Ceci dans le fichier *utilisateur1.data* et *utilisateur1.data* dont le chemin est (sous linux) /home/user/moi/.

```
#include <stdio.h>

main(){
FILE *lire;
FILE *ecrire;
int somme;
int entier;
int i;

lire=fopen(``/home/user/moi/utilisateur1.data'',''r'');
if (lire != NULL){
   somme=0;
   ecrire=fopen(``/home/user/moi/utilisateur2.data'',''w'');
   if (ecrire != NULL){
      for(i=0;i<10;i++){
         fscanf(lire,''%d'',&entier);
         somme=somme+entier;
         fprintf(ecrire,''%f'',somme/(i+1));
      }
   }
   else{
     printf(''Le programme s'arrete'');
   }
}
else{
   printf(''Le programme s'arrete'');
}

fclose(lire);
fclose(ecrire);
}
```

Remarquons que les lus sont lus les uns à la suite des autres de manière *séquentielle*.