Ecole Supérieure d'Ingénieurs de Luminy
Université de la Méditerranée

# Guide du langage C
# ESIL
# GBM -1ère année

Claudine Chaouiya
2002/2003

chaouiya@esil.univ-mrs.fr
http ://www.esil.univ-mrs.fr/~chaouiya

- Ceci est une première version, sûrement imparfaite, merci de me soumettre vos critiques !
- Ce document s'inspire largement du support de cours écrit par Touraivane

# Table des matières

# Chapitre 1

# Les bases

## 1.1 Introduction

Une des branches de l'informatique consiste à écrire des **programmes** pour résoudre des **problèmes**. Avant la phase d'écriture d'un programme et de son implémentation (que nous verrons plus loin), il faut d'abord bien définir le problème (et les données associées) et c'est **l'algorithmique** qui permet de le résoudre.

On ne peut réduire la notion d'algorithme aux seules méthodes informatiques de résolution de problèmes. L'algorithmique (ou algorithmie) est un terme dont l'origine remonte au XIIIème siècle, du nom d'un mathématicien perse Al-Khwarizmi. Mais on peut faire remonter la pratique algorithmique bien avant, avec les savants babyloniens de 1800 avant J.C. Un algorithme est une séquence d'opérations visant à la résolution d'un problème en un temps fini (mentionner la condition d'arrêt). On a tous des dizaines d'exemples d'algorithmes en tête :

l'algorithme de multiplication scalaire,

l'algorithme d'Euclide (calcul du pgcd de deux entiers),

l'algorithme de recherche dans un ensemble ordonné (dictionnaire), ...

La conception d'un algorithme se fait par étapes de plus en plus détaillées. La première version de l'algorithme est autant que possible indépendante de son implémentation, la représentation des données n'est donc pas fixée. A ce niveau, les données sont considérées de façon abstraite : un nom, les valeurs prises, les opérations possibles et les propriétés de ces opérations. On parle souvent de **type abstrait de données** (ou TAD). La conception de l'algorithme se fait en utilisant les opérations du TAD.

La représentation concrète du type de données est définie en termes d'objets du langage de programmation utilisé. Il se peut que ce type soit déjà défini dans le langage (c'est le cas par exemple des entiers et leurs opérations) ou qu'il faille le définir comme on le fera pour les listes chaînées.

Une fois l'algorithme et les structures de données définis, on les **code** en un langage informatique et on obtient un **programme** qui est défini comme une suite d'instructions permettant de réaliser une ou plusieurs tâche(s), de résoudre un problème, de manipuler des données. On parle souvent de **source** (au masculin). Il s'agit en fait

du texte source du programme, texte original d'un programme, dans le langage informatique compréhensible par un être humain, donc destiné à être **compilé**. Quand on vous demande **d'implémenter** un algorithme bien spécifié, il s'agit d'écrire le programme correspondant. La compilation est le passage du code source d'un programme à un **exécutable** à l'aide du logiciel approprié (le compilateur). L'exécutable est le programme en code binaire directement compréhensible par le processeur.

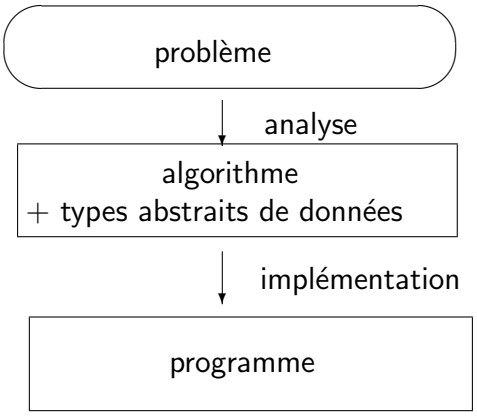



FIG. 1.1 – Etapes de développement

Le langage de base compréhensible par un ordinateur, appelé **langage machine** est constitué d'une suite de 0 et de 1. C'est du binaire, bien rébarbatif pour un être humain ! Le langage le plus proche est **l'assembleur**, pour lequel les instructions sont celles de la machine, mais des mnémoniques sont utilisées à la place du code binaire.

Pour le confort du programmeur, de nombreux langages de plus haut niveau ont été définis. On peut citer différents styles de programmation, avec pour chacun d'eux, un ou des langages particulièrement adaptés :

la programmation structurée (`Fortran, Pascal, C, Perl`),

la programmation structurée et modulaire (`Ada, Modula`),

la programmation fonctionnelle (`Lisp`),

la programmation logique (`Prolog`),

la programmation objet (`Smalltalk, Eiffel, C++, Java`).

Tout code écrit dans l'un de ces langages doit être "traduit" pour être compris de la machine. C'est le rôle des *interpréteurs* et *compilateurs*. Un compilateur est un logiciel de traduction qui traduit les textes sources (en langage source) en langage objet (souvent binaire exécutable par la machine).

Dans ce document, on décrit le langage `C ANSI`, qui est une version normalisée du langage.

## 1.2 Structure générale d'un programme C

Un programme C est un texte (il faudra utiliser un éditeur de textes), respectant la syntaxe du langage, et stocké sous la forme d'un ou plusieurs fichiers (avec l'extension .c). Il est composé d'un ensemble de **fonctions**, dont une est privilégiée, la fonction main, qui est le point d'entrée (ou point de départ) de tout programme. Voici le premier exemple (inévitable !) de programme en C :

```
#include <stdio.h>
int main() {
   printf(''Bonjour'');
   return 1;
}
```

Ce programme affiche la chaîne de caractères "Bonjour" à l'écran. Certaines fonctions, très utilisées, sont déjà écrites (fonctions prédéfinies, on parle de bibliothèque) et sont définies dans des fichiers que l'on évoquera plus loin. La fonction printf qui apparaît dans l'exemple qui précède fait partie de ces fonctions prédéfinies.

Chaque fichier (appelé **module**) est composé de :
- directives du préprocesseur (lignes commençant par #),
- définitions de types,
- définitions de fonctions,
- définitions de variables,
- déclarations de variables et fonctions externes.

## 1.3 Les phases de compilation

Les compilateurs C procèdent en deux étapes de traduction :
- le *préprocesseur* traite les directives au préprocesseur (lignes commençant par un dièze #) et réalise des transformations purement textuelles, prenant en compte les demandes d'inclusions, de compilation conditionnelle, de définition de macros (cf. annexe A),
- le *compilateur* proprement dit prend le texte généré par le préprocesseur et le traduit en langage machine.

Dans tout fichier, toute structure de donnée, variable ou fonction doit être déclarée. Quand il s'agit de définitions externes (fonctions de bibliothèques, variables externes, ...), ces informations sont contenues dans des fichiers (appelés fichiers d'entête ), avec l'extension .h. Ces fichiers doivent être inclus dans le fichier que l'on souhaite compiler. C'est le rôle de la directive du préprocesseur #include *nom de fichier*.

Dans l'exemple de l'affichage du message "Bonjour", on a inclus le fichier stdio.h qui contient les déclarations de variables externes et les prototypes de la bibliothèque d'entrée-sortie standard.

Pour plus de détails sur les directives du préprocesseur consultez l'annexe A, une introduction sur la compilation séparée est présentée dans l'annexe B.

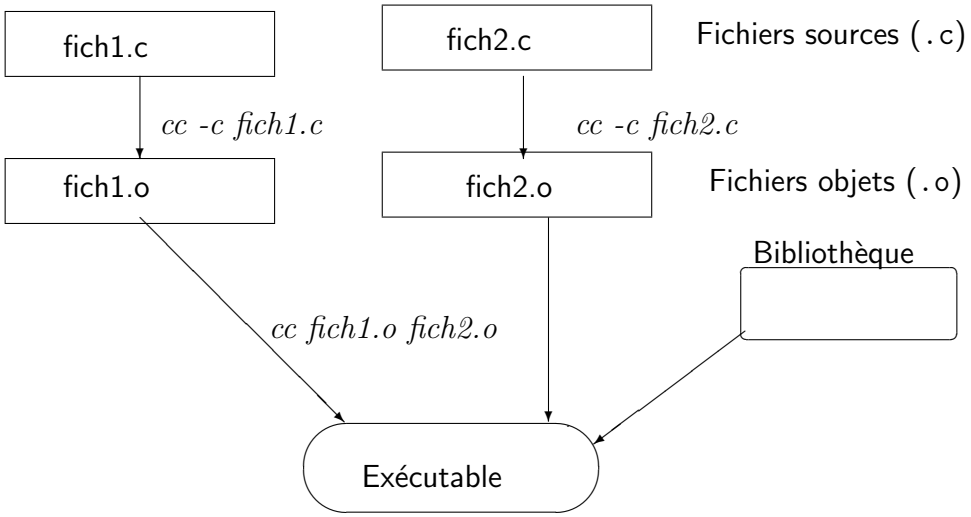



FIG. 1.2 – Etapes de compilation et édition de liens

## 1.4 Les unités lexicales

Le langage C comporte six types d'unités lexicales : les mots-clés (ou mots réservés), les identificateurs, les constantes, les chaînes de caractères, les opérateurs et les signes de ponctuation.

### 1.4.1 Les mots-clés

Certains mots sont réservés pour le langage et ne peuvent donc pas être utilisés comme identificateurs. En voici la liste :

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
const     loat    short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

### 1.4.2 Les identificateurs

Ils servent à donner un nom aux entités du programme (variables, fonctions, ...). Il sont formés d'une suite de lettres, chiffres et du signe souligné (_), le premier caractère étant impérativement une lettre ou un caractère souligné. Attention, les lettres utilisées peuvent être minuscules ou majuscules, mais doivent faire partie de l'alphabet anglais (pas d'accents). On conseille de choisir des identificateurs évocateurs !

## 1.5 Les commentaires

Il est **indispensable** d'inclure des commentaires dans vos sources. Ils doivent être concis (style télégraphique) et sont sensés guider le lecteur (vous par exemple) et expliciter les étapes du programme.

- un commentaire sur une ligne commence par les caractères `//`.
- un commentaire multilignes commence par les caractères `/*` et se termine par `*/`. A l'intérieur de ces délimiteurs, vous avez droit à toute suite de caractères (sauf évidemment `*/`).
- attention : on ne peut donc pas imbriquer des commentaires.

```
/**********************************/
/*  ce programme vous dit bonjour  */
/**********************************/

/*
fichier d'entete pour utiliser la fonction printf
*/
#include <stdio.h>

/* le main */
main(){
  printf(``Bonjour''); // afficher Bonjour
}
```

## 1.6 Les types de base

Un programme `C` manipule deux structures de données de base : les **entiers** et les **flottants**. Les **pointeurs** constituent un autre type de données élémentaire. Toute autre structure de données est dérivée de ces types de base. Les caractères, booléens, constantes symboliques, etc. ne sont rien d'autre que des entiers.

### 1.6.1 Les caractères

Le mot-clé désignant les caractères est `char`. Un objet de ce type peut contenir le code de n'importe quel caractère de l'ensemble des caractères utilisés (en général le code utilisé est le code ASCII, sur un octet).

Le jeu de caractères est formé d'un ensemble de caractères de commandes et de caractères graphiques. Les caractères de commandes que vous serez le plus susceptibles de rencontrer sont donnés dans le tableau 1.1

### 1.6.2 Les entiers

Le mot-clé désignant les entiers est `int`. Les entiers peuvent être affectés d'un attribut de précision (d'occupation en mémoire) et d'un attribut de représentation.

Les attributs de précision sont `short` et `long`. On a donc trois types d'entiers :

| **commande** | **nom** |
|---|---|
| *carriage return* | CR |
| *line feed* | LF |
| *backspace* | BS |
| *horizontal tabulation* | HT |
| *space* | SP |
| *escape* | ESC |
| *device control 1 (Ctrl-s)* | DC1 |
| *device control 3 (Ctrl-q)* | DC3 |
| *delete* | DEL |

TAB. 1.1 – Quelques caractères de commandes

| code | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT |
| 10 | LF | VT | NP | CR | SO | SI | DLE | DC1 | DC2 | DC3 |
| 20 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS |
| 30 | RS | US | SP | ! | " | # | $ | % | & | ’ |
| 40 | ( | ) | * | + | , | - | . | / | 0 | 1 |
| 50 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 60 | < | = | > | ? | @ | A | B | C | D | E |
| 70 | F | G | H | I | J | K | L | M | N | O |
| 80 | P | Q | R | S | T | U | V | W | X | Y |
| 90 | Z | [ | \ | ] | ^ | _ | ‘ | a | b | c |
| 100 | d | e | f | g | h | i | j | k | l | m |
| 110 | n | o | p | q | r | s | t | u | v | w |
| 120 | x | y | z | { | ‖ | } | ~ | DEL | | |

TAB. 1.2 – Codes ASCII en décimal

- les `short int`, sont codés, si possible sur une longueur inférieure aux `int` (2 octets, de -32768 à 32767),
- `int`, sont codés sur 4 octets sur nos machines (de -2147843648 à 2157843647),
- `long int` sont codés, quand c’est possible sur une longueur plus grande que les `int` (ce n’est pas le cas pour nos machines).

L’attribut de représentation est `unsigned` qui permet de spécifier un type d’entier non signés, en représentation binaire. La question de la représentation des entiers en complément à 2 sera vue en TD. Par exemple, le type `unsigned int` représente un entier non signé codé sur 4 octets (de 0 à 4294967295).

Attention, les occupations en mémoire mentionnées ci-dessus peuvent varier d’une plateforme à l’autre...

### 1.6.3 Les flottants

Les nombres à virgule flottante (abusivement appelés réels), permettent une représentation **approchée** des nombres réels. Un nombre à virgule flottante est composé d'un signe, d'une mantisse et d'un exposant. On dispose des types :

- `float` codés sur 4 octets avec 1 bit de signe, 23 bits de mantisse et 8 bits d'exposant (valeurs comprises entre 3.4 $10^{-38}$ et 3.4 $10^{38}$),
- `double` codés sur 8 octets avec 1 bit de signe, 52 bits de mantisse et 11 bits d'exposant (valeurs comprises entre 1.7 $10^{-308}$ et 1.7 $10^{308}$),
- `long double` codés sur 10 octets avec 1 bit de signe, 64 bits de mantisse et 15 bits d'exposant (valeurs comprises entre 3.4 $10^{-4932}$ et 3.4 $10^{4932}$).

**Attention**, les nombres à virgule flottante sont des valeurs approchées, et les opérations sur ces nombres peuvent conduire à des erreurs d'arrondi. Les représentations des types décrits ci-dessus peuvent encore varier selon les plateformes.

## 1.7 Les constantes littérales

### 1.7.1 Les constantes entières

Elles s'expriment selon trois notations :

- notation décimale : 123, -123, ...
- notation octale (avec un 0 en première position) : 0123,
- notation hexadécimale (avec les caractères `0x` ou `0X` en première position) : 0x1bn 0X2C, 0X2c,...

Le type d'une constante entière est le *plus petit type* dans lequel elle peut être représentée.

### 1.7.2 Les constantes flottantes

Une constante flottante est représentée sous forme d'une suite de chiffres (partie entière), un point jouant le rôle de virgule, une suite de chiffres (partie fractionnaire), la lettre `e` (ou `E`), éventuellement le signe + ou - suivi d'une suite de chiffres (valeur absolue de l'exposant). La partie entière ou la partie fractionnaire peuvent être omises, mais pas les deux. Le point ou l'exposant peuvent être omis (mais pas les deux). Une constante flottante est supposée être de type `double`, le suffixe `F` indique qu'elle est de type `float`, le suffixe `LF` indique qu'elle est de type `long double`. Des exemples : `3.14, .5e4, 5.E3, 5000`

### 1.7.3 Les constantes caractères

Elles se notent entre apostrophes : `'a','1','/',...`

Le caractère ' se note `'\''`, on peut aussi représenter des caractères non imprimables à l'aide de ce qu'on appelle des séquences d'échappement, le tableau ci-dessous vous donne les principales :

| Séquence | |
|---|---|
| \n | nouvelle ligne |
| \t | tabulation horizontale |
| \v | tabulation verticale |
| \f | saut de page |
| \a | beep |
| \' | apostrophe |
| \" | guillemets |
| \\ | anti-slash |

### 1.7.4 Les constantes chaînes de caractères

Elles se notent entre guillemets " :

```
"coucou", "il faut apprendre la programmation !"
```

Une chaîne de caractères est une suite de caractères (éventuellement vide) entre guillemets. Il en découle que l'on peut utiliser les séquences d'échappement dans les chaînes. L'instruction suivante :

```
printf("Bonjour, \n\tcomment vas-tu ?\n") ;
```

produit la sortie :

```
Bonjour,
comment vas-tu ?
```

Il faut savoir que la représentation en mémoire d'une chaîne de caractères se fait dans des emplacements contigüs, et que le dernier élément est le caractère nul '\0'. Une chaîne doit être écrite sur une seule ligne, si elle est trop longue pour cela, on masque le retour à la ligne par le caractère \. Ainsi les 2 instructions suivantes sont équivalentes :

```
x="abcdefghijklm\
nopqrstuvwxyz\
ABCDEFGHIJKLM\
NOPQRSTUVWXYZ";
```

et,

```
x="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

### 1.7.5 Des booléens ?

Contrairement à d'autres langages (Pascal par exemple), il n'y a pas de type booléen en C. On le représente comme un entier et il se comporte comme la valeur booléenne *vrai* si cet entier est non nul, la valeur *faux* correspondant à la valeur nulle. Nous reverrons cela plus loin.

## 1.8 Les variables

Une variable est caractérisée par :
- un nom, ou *identificateur* composé d'une suite de caractères commençant par une lettre et suivie de caractères alphanumériques ou du caractère _ (souligné) : `x, x1, cpt, var_locale,...`
- un type qui permet d'allouer l'espace mémoire nécessaire pour stocker la valeur de la variable. On peut considérer la mémoire comme une suite d'octets dans laquelle on réserve des octets pour les variables en fonction de leur taille (laquelle est donnée par le type de la variable).
- une valeur arbitraire si la variable n'est pas explicitement affectée. Une variable, une fois définie **contient toujours une valeur**.
- une adresse qui désigne l'emplacement de la variable en mémoire. Cette adresse est définie une fois pour toutes et ne varie pas au cours du programme. Le contenu de la variable peut être modifié autant de fois que l'on veut, ce qui ne change pas, c'est son adresse.

**Attention :** la notion de variable en informatique n'a rien à voir avec celle de variable mathématique. En informatique, il s'agit d'une adresse en mémoire et l'expression x=x+1 a un sens (ajouter 1 au contenu de l'adresse de x).

### Déclarer et définir des variables

En `C` toute variable utilisée dans un programme doit auparavant avoir été définie. Cette **définition** consiste à la nommer, à lui donner un type et, éventuellement lui donner une valeur initiale. C'est cette définition qui réserve (on dit **allouer**) la place mémoire nécessaire en fonction du type. Initialiser une variable consiste à remplir la zone mémoire réservée à cette variable avec la valeur d'une constante.

Une **déclaration** de variable définit un nom et un type mais n'alloue pas la place mémoire. En effet, si le programme est composé de plusieurs modules, une même variable, utilisée dans différents modules, devra être déclarée dans chacun de ces modules. Par contre, on ne définit cette variable que dans un seul module.

```
int x=2 ;
char c='c' ;
float f=1.3 ;
```

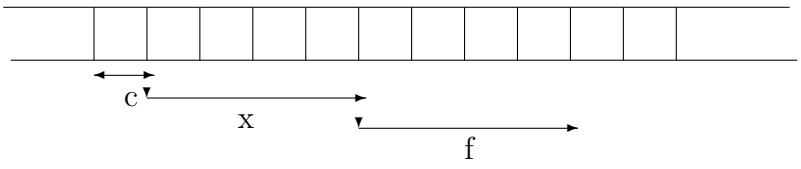



FIG. 1.3 – Représentation interne de variables

La syntaxe de la déclaration/définition la plus simple est la suivante :

```
type identificateur <=expression>
```

Nous verrons plus tard que cette syntaxe peut s'enrichir de mots clés complémentaires (`auto, extern, static, register, const...`).
**Attention :** certains identificateurs sont réservés et ne peuvent être utilisés comme noms de variables.

#### Portée d'une variable

Il est un peu tôt pour en parler, mais sachez qu'une variable a une portée (une durée de vie et une visibilité) et celle-ci dépend uniquement de la position de sa déclaration ou définition. Il y a trois types de variables :
- les variables **globales** déclarées en dehors de toute fonction,
- les variables **locales** déclarées à l'intérieur de fonctions,
- les **arguments formels** (ou paramètres) qui sont déclarés dans les entêtes des fonctions.

## 1.9 Fonctions de base pour les entrées/sorties

Il est fondamental de pouvoir communiquer avec un programme, lui fournir des données, et voir s'afficher des résultats. Les fonctions d'entrées/sorties permettent de réaliser cette interaction. Ici, on ne présente que les fonctions de base qui vont nous permettre de commencer à programmer. Le chapitre 7 est consacré aux entrées/sorties.

### 1.9.1 Les sorties

Les résultats calculés par un programme sont affichés à l'écran (sortie standard). On verra plus tard comment écrire ces résultats dans des fichiers.

#### Sortie formatée : `printf`

La fonction `printf` permet d'afficher à l'écran les arguments qu'on lui fournit. La syntaxe est la suivante :

`printf("`*format*`", arg1, arg2,...,argN);`

L'argument *format* est une chaîne de caractères qui peut contenir des spécifications de format données par le caractère spécial % suivi d'une ou plusieurs lettres significatives. Voici quelques spécifications usuelles (il y en a d'autres que nous verrons plus tard) :

| | |
|---|---|
| `%c` | caractère |
| `%s` | chaîne de caractères |
| `%d` | entier |
| `%f` | flottant |

**Exemples :**

`printf("Coucou")`, pas d'arguments, *format* est réduit à la chaîne "Coucou",

`printf("%s","Coucou")`, un argument de type chaîne qui est la chaîne "Coucou",

`printf("%d",3)`, un argument de type entier qui est l'entier 3,

`printf("La somme de %d et %d vaut %d" ,3,5,3+5)`, trois arguments entiers.

**Attention :** il n'y a aucune vérification de cohérence entre le *format* et le type des arguments fournis. L'instruction `printf("%c %d",65,65) ;` affichera à l'écran `A 65`.

### 1.9.2 Les entrées

En général un programme a besoin que vous lui fournissiez des données. Par exemple, si votre programme calcule la solution d'une équation de degré 2, il lui faudra trois coefficients `a,b,c` (l'équation s'écrit $ax^2 + bx + c$). Une façon de fournir des données à un programme consiste à les taper au clavier (entrée standard) de manière interactive. Le programme peut aussi aller lire ses données dans un fichier.

#### Entrée formatée : `scanf`

La fonction `scanf` permet de lire des données au clavier. La syntaxe est de la forme :

`scanf("`*format*`", &arg1, &arg2,...,&argN);`

L'argument *format* est le même que précédemment. Le symbole & devant chaque argument signifie *adresse*, nous le reverrons plus tard. Pour l'instant contentons nous de comprendre le mécanisme des entrées avec les exemples donnés ci-après.

#### Autre fonction d'entrée

La fonction `int getchar(void)` retourne un caractère lu au clavier.

**Exemples :**

`scanf("%d",&x) ;` l'adresse de la variable `x` est remplie avec ce que l'utilisateur entrera au clavier (qui doit être un entier).

`scanf("%s",ch) ;` l'adresse de la variable `ch` est remplie avec ce que l'utilisateur entrera au clavier (qui doit être une chaîne).

`c=getchar() ; c` reçoit la valeur du caractère entré au clavier.

# Chapitre 2

# Expressions et opérateurs

## 2.1 Expressions, ordres et priorités d'évaluation

Une expression est un objet syntaxique obtenu en assemblant `correctement` des constantes, variables et opérateurs. Par exemple `x+3` est une expression. Dans le langage `C` il y a une quarantaine d'opérateurs. Nous décrirons ici les plus usuels.

Il faut savoir qu'il existe des priorités et des ordres d'évaluation qui régissent l'évaluation des expressions. Vous avez l'habitude de manipuler des expressions arithmétiques, et vous devez savoir que l'expression $2+x+4*x$ est équivalente à $(2+(x+(4*x)))$. Les parenthèses permettent de définir les expressions sans ambiguité. Pour limiter l'usage de parenthèses, on a fixé des règles pour l'évaluation des expressions. Par exemple l'expression $2+3*4$ équivaut à $2+(3*4)$, l'opérateur $*$ ayant une priorité supérieure à celle de $+$. De même l'expression $2-3-4$ équivaut à $(2-3)-4$, l'ordre d'évaluation de l'opérateur $-$ étant de *gauche à droite*. Ces règles ne dispensent pas de l'utilisation de parenthèses, par exemple l'expression $(2+3)*4$ nécessite l'utilisation de parenthèses.

## 2.2 Les opérateurs

### 2.2.1 L'opérateur d'affectation =

L'opération la plus importante dans un langage de programmation est probablement celle qui consiste à donner une valeur à une variable. En `C` cette opération est notée `=`.

**Attention :** il ne s'agit pas d'un symbole d'équation mathématique mais du remplissage d'une zone mémoire avec une valeur donnée.

Comme l'affectation place une valeur dans une variable, il est impératif que le membre gauche d'une affectation représente une zone mémoire. C'est ce qu'on appelle une *lvalue* (pour *left value* ou *location value*). Une constante littérale, par exemple, n'est pas une *lvalue* car elle ne désigne pas une zone mémoire ; elle ne peut donc pas figurer comme membre gauche d'une affectation.

Le membre droit d'une affectation peut être une constante, une variable ou encore

une expression.

**Exemples :**

l'affectation `x=2` range la valeur 2 dans la variable `x`,
l'affectation `x=y` range le contenu de la variable `y` dans la variable `x`,
l'affectation `x=y+1` range le contenu de la variable `y` incrémenté de 1 dans la variable `x`.

Une affectation est une expression. Elle a donc une valeur qui est celle de son membre gauche après exécution de l'affectation. Si la variable `y` contient la valeur 20 alors l'affectation `x=y-1` a pour valeur 19. En conséquence, une affectation peut figurer comme membre droit d'une affectation. Par exemple, l'affectation `x=y=1` est valide et range la valeur 1 dans la variable `y` puis la valeur 1 (valeur de l'affectation `y=1`) dans la variable `x`.

### 2.2.2 Les opérateurs arithmétiques `+,-,*,/,%`

Ils se comportent comme on s'y attend, pour ce qui concerne `+,-,` et `*`. Par contre, il faut faire un peu attention avec l'opérateur `/`, car si les opérandes sont des flottants, le résultat est un flottant obtenu en divisant les deux nombres **mais** si les deux opérandes sont des entiers, le résultat est un entier obtenu comme résultat de la division entière. L'opérateur `%` n'est défini que pour les entiers et le résultat est le reste de la division entière des opérandes (modulo).

| Opérateur | Nom | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `+` | addition | `x+y` | 12 | gauche-droite |
| `-` | soustraction | `x-y` | 12 | gauche-droite |
| `*` | multiplication | `x*y` | 13 | gauche-droite |
| `%` | modulo | `x%y` | 13 | gauche-droite |
| `-`(unaire) | négation | `-x` | 14 | droite-gauche |

### 2.2.3 Les opérateurs de comparaison `==, !=,<=,>=,<,>`

En théorie, le résultat d'une comparaison est un booléen (*vrai* ou *faux*). En `C`, le résultat d'une comparaison est 1 ou 0 selon que cette comparaison est `vraie` ou `fausse`.
**Attention !** Ne confondez pas la comparaison `==` et l'affectation `=`.

| Opérateur | Nom | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `==` | test d'égalité | `x==y` | 9 | gauche-droite |
| `!=` | test de non égalité | `x !=y` | 9 | gauche-droite |
| `<` | test inférieur strict | `x<y` | 10 | gauche-droite |
| `>` | test supérieur strict | `x>y` | 10 | gauche-droite |
| `<=` | test inférieur ou égal | `x<=y` | 10 | gauche-droite |
| `>=` | test supérieur ou égal | `x>=y` | 10 | gauche-droite |

### 2.2.4 Les opérateurs logiques `&&`, `||`, `!`

**Tout petit rappel de logique booléenne**

Une variable booléenne est une variable pouvant prendre la valeur `vrai` ou `faux`. L'algèbre de Boole étudie les formules construites à partir de constantes booléennes (`vrai` et `faux`), de variables booléennes et des connecteurs logiques (`non` noté $\neg$, `et` noté $\wedge$, et `ou` noté $\vee$). Voici la table de vérité pour ces trois connecteurs :

| $a$ | $b$ | $\neg a$ | $a \wedge b$ | $a \vee b$ |
|---|---|---|---|---|
| vrai | vrai | faux | vrai | vrai |
| vrai | faux | faux | faux | vrai |
| faux | vrai | vrai | faux | vrai |
| faux | faux | vrai | faux | faux |

En `C`, on dispose de ces connecteurs (avec une syntaxe particulière) et on peut construire des expressions booléennes. La valeur d'une expression booléenne est une valeur entière, tout comme le résultat d'une comparaison.

**Table des opérateurs logiques**

| Opérateur | Nom | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `&&` | ET | `x&&y` | 5 | gauche-droite |
| `\|\|` | OU | `x\|\|y` | 4 | gauche-droite |
| `!` (unaire) | NON | `!x` | 14 | droite-gauche |

**Evaluation des expressions booléennes**

Comme pour les autres opérateurs, l'évaluation d'une expression booléenne obéit aux règles de priorité et d'ordre d'évaluation. Il est bon de préciser que cette évaluation est arrêtée dès lors que l'on est capable de donner la valeur de l'expression. Par exemple, supposons que la variable $x$ contienne la valeur 5, l'évaluation de l'expression $(x >= 2)||(x > y)$ s'arrête avant d'avoir évalué l'expression $(x > y)$, car $(x >= 2)$ étant *vrai*, on peut conclure que $(x >= 2)||(x > y)$ est *vrai*. Cette remarque est importante pour les tests d'arrêt et les expressions avec effet de bord[1]. Par exemple, selon la valeur de $x$, l'évaluation de l'expression $(x >= 2)||(x = x+1)$ aura pour effet de bord l'incrément de $x$ ou non.

**Exemple :**

```
#include <stdio.h>
int main(){
  int i=1;
  printf("%d\n",i=i+1);
  if (i) printf("oui\n");
```

[1] on parle *d'effet de bord* quand l'évaluation d'une expression produit la modification de la valeur d'une opérande

```
  if (i=i-2) printf("non\n");
}
```

Notez tout de même que les exemples qui précèdent ne sont pas des modèles de programmation élégante ! On vous demandera même d'éviter à tout prix ce style qui produit un code peu lisible.

### 2.2.5 Opérateurs binaires d'affectation `+=, -=, *=,...`

Ce ne sont que des raccourcis de notation et il vaut mieux éviter d'en abuser !

| Opérateur | Equivalent | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `+=` | `x=x+y` | `x+=y` | 2 | droite-gauche |
| `-=` | `x=x-y` | `x-=y` | 2 | droite-gauche |
| `*=` | `x=x*y` | `x*=y` | 2 | droite-gauche |
| `/=` | `x=x/y` | `x/=y` | 2 | droite-gauche |
| `%=` | `x=x%y` | `x%=y` | 2 | droite-gauche |

### 2.2.6 Opérateurs unaires d'affectation `++, --`

On les appelle des opérateurs d'incrément et de décrément. Ils sont en position de préfixe ou de suffixe : l'expression `y=x++` est le raccourci de `y=x ;x=x+1 ;` et l'expression `y=++x` est le raccourci de `x=x+1 ;y=x ;`. Ces opérateurs produisent un *effet de bord* c'est-à-dire la modification d'une opérande lors de l'évaluation de l'expression.

| Opérateur | Equivalent | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `++` | `x=x+1` | `++x` ou `x++` | 14 | droite-gauche |
| `--` | `x=x-1` | `--x` ou `x--` | 14 | droite-gauche |

### 2.2.7 Autres opérateurs importants `sizeof, ? :, &, (type), (), ., ->, [],`

**L'opérateur de dimension `sizeof`**

Il donne l'occupation mémoire (en octets) d'une variable ou d'un type de données.

| | | | | |
|---|---|---|---|---|
| `sizeof` | Opérateur de dimension | `sizeof(e)` | 14 | droite-gauche |

**L'opérateur conditionnel `? :`**

C'est le seul opérateur ternaire, il s'agit d'une sorte de *si... alors... sinon* sous forme d'expression. Si la condition `e` est *vraie* alors cette expression vaut `x`, sinon, elle vaut `y`. Il est conseillé de ne l'utiliser que pour des conditions très simples.

| Opérateur | Nom | Notation | Priorité | Ordre |
|---|---|---|---|---|
| `? :` | Opérateur conditionnel | `e? x : y` | 3 | droite-gauche |

### L'opérateur d'adressage `&`

Il donne l'adresse en mémoire d'une variable ou d'une expression qui est une *lvalue*.

| | | | | |
|---|---|---|---|---|
| `sizeof` | Opérateur de dimension | `sizeof(e)` | 14 | droite-gauche |

### L'opérateur de conversion `(type)`

Il permet de convertir explicitement le type d'une donnée en un autre type. Le compilateur `C` procède à des conversion implicites que nous verrons dans la section qui suit. Dans le jargon de `C`, l'opérateur de conversion de type s'appelle un *cast*.

| | | | | |
|---|---|---|---|---|
| `(type)` | Opérateur de conversion | `(long int)j` | 14 | droite-gauche |

### L'opérateur de parenthésage `()`

Il permet de définir l'ordre d'évaluation d'une expression (vous l'utilisez déjà usuellement). Il sert également à encapsuler les paramètres des fonctions.

| | | | | |
|---|---|---|---|---|
| `()` | Opérateur de parenthésage | `()` | 15 | gauche-droite |

### Les opérateurs de sélection `.`, `[]` et `->`

Ils permettent de sélectionner des champs de données structurées. Nous les verrons plus loin dans le cours.

| | | | | |
|---|---|---|---|---|
| `.` | Opérateur de sélection | `x.info` | 15 | gauche-droite |
| `->` | Opérateur de sélection | `x->info` | 15 | gauche-droite |
| `[]` | Opérateur de sélection | `x[2]` | 15 | gauche-droite |

### L'opérateur séquentiel , (virgule)

Cet opérateur permet de regrouper plusieurs expressions en une seule : l'évaluation de l'expression `e1, e2` consiste en l'évaluation successive (dans l'ordre) des expressions `e1` puis `e2`. L'expression `e1, e2` possède le type et la valeur de `e2`.

## 2.3 Récapitulatif

| Opérateur | Nom | Priorité | Ordre |
|---|---|---|---|
| `[]` | Elément de tableau | 15 | gauche-droite |
| `()` | Parenthésage | 15 | gauche-droite |
| `.` | Sélection | 15 | gauche-droite |
| `->` | Sélection | 15 | gauche-droite |
| `&` | Adressage | 15 | gauche-droite |
| `sizeof` | Dimension | 14 | droite-gauche |
| `(type)` | Conversion | 14 | droite-gauche |
| `!` unaire | NON | 14 | droite-gauche |
| `~` unaire | NON bit à bit | 14 | droite-gauche |
| `++` | Incrément | 14 | droite-gauche |
| `--` | Décrément | 14 | droite-gauche |
| `*` | Multiplication | 13 | gauche-droite |
| `/` | Division | 13 | gauche-droite |
| `%` | Modulo | 13 | gauche-droite |
| `+` | Addition | 12 | gauche-droite |
| `-` | Soustraction | 12 | gauche-droite |
| `>>` | Décalage à droite | 11 | droite-gauche |
| `<<` | Décalage à gauche | 11 | droite-gauche |
| `<=` | Test inférieur ou égal | 10 | gauche-droite |
| `>=` | Test supérieur ou égal | 10 | gauche-droite |
| `<` | Test inférieur strict | 10 | gauche-droite |
| `>` | Test supérieur strict | 10 | gauche-droite |
| `==` | Test égalité | 9 | gauche-droite |
| `!=` | Test non-égalité | 9 | gauche-droite |
| `&` | ET bit à bit | 8 | gauche-droite |
| `^` | OU exclusif bit à bit | 7 | gauche-droite |
| `\|` | OU bit à bit | 6 | gauche-droite |
| `&&` | ET | 5 | gauche-droite |
| `\|\|` | OU | 4 | gauche-droite |
| `? :` ternaire | Conditionnel | 3 | droite-gauche |
| `=` | Affectation | 2 | droite-gauche |
| `+=` | Affectation | 2 | droite-gauche |
| `-=` | Affectation | 2 | droite-gauche |
| `*=` | Affectation | 2 | droite-gauche |
| `/=` | Affectation | 2 | droite-gauche |
| `%=` | Affectation | 2 | droite-gauche |
| `>>=` | Affectation | 2 | droite-gauche |
| `<<=` | Affectation | 2 | droite-gauche |
| `&=` | Affectation | 2 | droite-gauche |
| `\|=` | Affectation | 2 | droite-gauche |
| `^=` | Affectation | 2 | droite-gauche |
| `,` | Séquentiel | 1 | droite-gauche |

## 2.4 Conversion de types

Le problème de la conversion de types se pose quand une expression est composée de données de natures différentes. Par exemple, quel sens donner à une addition d'un entier et d'un flottant ? Que se passe-t-il quand on affecte une variable de type entier à une variable de type caractère ou flottant ? Le langage `C` définit de façon précise les types de données compatibles et les conversions qui sont effectuées.

### 2.4.1 Les conversions implicites

Les conversions implicites sont celles qui sont faites automatiquement par le compilateur `C` lors de l'évaluation d'une expression (et donc aussi d'une affectation). On verra plus loin qu'il peut y avoir aussi conversion implicite lors des appels de fonctions.

Dans un programme, lorsqu'on attend une valeur d'un certain type, il faut théoriquement fournir une valeur de ce type (par exemple, dans une affectation, si la partie gauche est de type flottant, la valeur fournie par la partie droite devra être également de ce type). Néanmoins, on n'applique pas cette règle de façon trop stricte. Si le type attendu et le type fourni sont trop différents, il est normal que le compilateur considère qu'il s'agit d'une erreur (par exemple, si l'on cherche à affecter une valeur de type structure à une variable de type entier). Par contre, si le type attendu et le type fourni sont *proches* (par exemple dans le cas d'un flottant et d'un entier), le compilateur fait lui-même la conversion (c'est une facilité souhaitable). Il existe d'autres cas où les conversions de type sont utiles : le cas de l'évaluation des expressions. Il faut savoir que, physiquement, les opérations arithmétiques sont réalisées par des instructions différentes selon qu'elles portent sur des entiers ou des flottants. Le compilateur fait donc les conversions de types nécessaires sur les opérandes pour avoir un même type.

**Règles de conversion implicite :**[2]

- Convertir les éléments de la partie droite d'une expression dans le type de la variable la plus riche.
- Faire les calculs dans ce type.
- Puis convertir le résultat dans le type de la variable affectée.

La notion de richesse d'un type est précisée dans la norme ANSI. Le type dans lequel un calcul d'une expression à deux opérandes doit se faire est donné par les règles suivantes :

- si l'un des deux opérandes est du type `long double` alors le calcul doit être fait dans le type `long double` ;
- sinon, si l'un des deux opérandes est du type `double` alors le calcul doit être fait dans le type `double` ;

[2] tiré du *Support de Cours de Langage C* de Christian Bac, http ://www-inf.int-evry.fr/COURS/COURSC/index.html

- sinon, si l'un des deux opérandes est du type `float` alors le calcul doit être fait dans le type `float` ;
- sinon, appliquer la règle de promotion en entier, puis :
  - si l'un des deux opérandes est du type `unsigned long int` alors le calcul doit être fait dans ce type ;
  - si l'un des deux opérandes est du type `long int` alors le calcul doit être fait dans le type `long int` ;
  - si l'un des deux opérandes est du type `unsigned int` alors le calcul doit être fait dans le type `unsigned int` ;
  - si l'un des deux opérandes est du type `int` alors le calcul doit être fait dans le type `int`.

La *règle de promotion en entier* précise que lorsqu'on utilise des variables ou des constantes des types suivants dans une expression, alors les valeurs de ces variables ou constantes sont transformées en leur équivalent en entier avant de faire les calculs. Ceci permet d'utiliser des caractères, des entiers courts, des champs de bits et des énumérations de la même façon que des entiers. On peut utiliser à la place d'un entier :

- des caractères et des entiers courts signés ou non signés ;
- des champs de bits et des énumérations signés ou non signés.

**Exemples :**

```
float f; double d; int i; long li;
li = f + i ;
```

`i` est transformé en `float` puis additionné à `f`, le résultat est transformé en `long` et rangé dans `li`.

```
d = li + i ;
```

`i` est transformé en `long` puis additionné à `li`, le résultat est transformé en `double` et rangé dans `d`.

```
i= f + d ;
```

f est transformé en `double`, additionné à `d`, le résultat est transformé en `int` et rangé dans `i`.

### 2.4.2 Les conversions explicites

Il est possible de forcer la conversion d'une variable (ou d'une expression) dans un autre type avant de l'utiliser par une conversion implicite. On utilise pour cela l'opérateur de conversion de type.

Par exemple : `i = (int) f + (int) d ;` . Les variables `f` et `d` sont converties en `int`, puis additionnées. Le résultat entier est rangé dans `i`. Attention, il peut y avoir une différence entre `i = f + d ;` et `i = (int) f + (int) d ;` du fait de la perte des parties fractionnaires.

# Chapitre 3

# Les Structures de contrôle

On peut dire, en résumant, qu'un programme `C` est constitué de définitions et déclarations de variables et de fonctions. Les fonctions sont construites à l'aide d'instructions, combinées entre elles par des **structures de contrôle**.

## 3.1 Instructions

Une instruction est une **instruction simple** ou bien un **bloc d'instructions**.

Une instruction simple est
- une **instruction de contrôle**,
- ou une expression suivie de “;” (le point virgule marque la fin d'une instruction)

Un bloc d'instructions est composé de :

une accolade ouvrante ”{”

une liste de définitions locales au bloc (optionnelle)

une suite d'instructions

une accolade fermante ”}” (sans “;”).

**Exemples :**

```
x=0;
i++;
printf("Coucou\n");

if (x==0) {
  x=1;
  i++;
  printf("Dans le bloc !\n");
}
```

## 3.2 L'instruction conditionnelle `if`

Cette instruction conditionnelle permet d'exécuter des instructions de façon sélective en fonction du résultat d'un test. La syntaxe est la suivante :

`if (`*expression*`)` *instruction1*

`if (`*expression*`)` *instruction1* `else` *instruction2*

Si l'*expression* est *vraie*, l'*instruction1* est exécutée. Sinon c'est l'*instruction2* qui est exécutée. Notez que la partie `else` est facultative.

## 3.3 L'étude de cas avec `switch`

L'instruction `switch` permet de faire une étude de cas de la façon suivante : si la valeur d'une expression est égale à l'une des **constantes** précisées (par la partie `case`), les instructions qui suivent sont toutes exécutées. La partie `default` permet de spécifier les instructions à exécuter si l'expression n'est égale à aucune des constantes. L'exemple ci-dessous devrait suffire pour bien comprendre le mécanisme de l'instruction `switch`. Notez que l'instruction `break` permet de sortir du `switch`.

```
#include <stdio.h>

int main(){
  int i;
  printf("Entre un chiffre : ");
  scanf("%d",&i);
  switch (i) {
    case 0 : printf("ton chiffre est zero !\n");
             break;
    case 6 :
    case 8 : printf("ton chiffre est plus grand que 5\n");
    case 4 :
    case 2 : printf("ton chiffre est pair\n");
             break;
    case 9 :
    case 7 : printf("ton chiffre est plus grand que 5\n");
    case 5 :
    case 3 :
    case 1 : printf("ton chiffre est impair\n");
             break;
    default: printf("ce n'est pas un chiffre !\n");
  }
}
```

## 3.4 Les instructions itératives : `while`, `do...while` et `for`

Ce sont les fameuses instructions de **boucle**. Elles permettent d'exécuter une même séquence d'instructions un certain nombre de fois.

### 3.4.1 `while`

La structure de contrôle `while` évalue une condition et exécute l'instruction tant que cette condition reste vraie.

```
while (condition)
  instruction
```

**Exemple :**

```
int i=10;
while (i>=0) {
       printf(i);
       i=i-1;
}
```

### 3.4.2 `do...while`

L'instruction `do...while` est une variante de la précédente. Une itération est toujours exécutée. Il faut la traduire en français par *Faire... tant que*. Attention de ne pas confondre avec la structure *répéter...jusqu'à ce que*!

```
do
  instruction
while (condition)
```

**Exemple :**

```
int i=-1;
do {
       System.out.println(i);
       i=i-1;
} while (i>=0);
```

### 3.4.3 `for`

L'instruction `for` qui comporte une initialisation, une condition de poursuite, et une ou des instructions de fin de boucle :

```
for (instruction1;condition_de_poursuite;instruction2) instruction3
```

est équivalente à :

```
instruction1;
while (condition_de_poursuite) {
       instruction3
       instruction2
}
```

La virgule (,) est utilisée pour combiner plusieurs initialisations et plusieurs instructions de fin de boucle.

## 3.5 Autres instructions `break, continue, return, goto`

On a déjà rencontré l'instruction `break` dans le seul cas où son usage est vraiment justifié (le `switch`). D'une façon générale, `break` vous permet de sortir d'une instruction `switch`, `for`, `do ... while` ou `while`. Pour les boucles, il est vivement conseillé (pour nous ce sera **obligatoire !**) d'écrire correctement les tests d'arrêt, on peut toujours se passer de l'usage de `break` !

L'instruction `continue`, utilisée dans les boucles `for`, `do ... while` ou `while` permet d'abandonner l'itération courante pour passer au début de l'itération suivante (peu utilisée, à éviter...).

**Exemple :** Supposons que l'on demande à l'utilisateur de fournir 10 entiers, et chaque fois que l'entier fourni est positif on procède à un certain traitement (par exemple lister tous les diviseurs). Vous remarquerez que l'on peut très bien utiliser une instruction `if ... else` pour faire cela.

```
for (i = 0; i < 10; i = i + 1)
   {
   printf("entrer votre %deme entier :",i);
   scanf("%d",&n);
   if (n < 0 ) continue;      /*   on passe au i suivant dans le for   */
   ...                        /*   traitement de l'entier fourni       */
   }
```

Nous verrons en détail l'instruction `return` lorsque nous aborderons le chapitre sur les fonctions. En deux mots, cette instruction permet d'abandonner une fonction pour retourner à la fonction appelante (en fournissant éventuellement une valeur).

Enfin, on se contentera d'évoquer l'instruction `goto` que l'on vous demande de ne pas utiliser (elle est contraire aux principes de la programmation structurée !). Vous pouvez faire précéder toute instruction par un identificateur (appelé étiquette) suivi de " :". Une instruction `goto identificateur` a pour effet de se brancher à l'instruction étiquetée par `identificateur` qui est alors exécutée.

# Chapitre 4

# Fonctions

## 4.1 Introduction

Un programme C[1] est constitué d'un ensemble de fonctions toutes accessibles à partir du *programme principal*, la fonction `main()`. Le découpage en différentes fonctions permet de concevoir un programme à partir de briques de base, plus faciles à concevoir, tester et modifier.

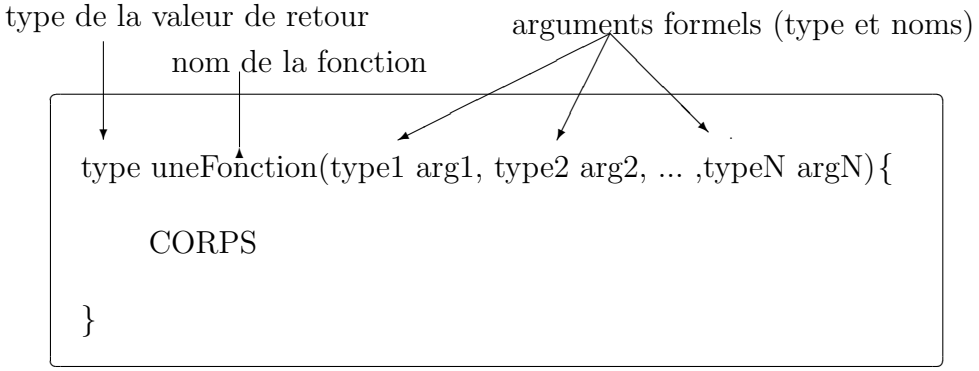



FIG. 4.1 – Structure d'une fonction

**Exemple :**

```
int pgcd(int a, int b) { // calcul du pgcd de a et b
   int c;
   if (a<b) { // echanger a et b
      c=a;   a=b;   b=c;
   }
   do {
      c=a%b; a=b;  b=c;
   } while (c!=0);
   return a;
}
```

[1] ce chapitre est spécifique au `C ANSI`, qui a "nettoyé" les premières versions du langage

```
int main(){
   int x,y;
   printf("donner deux entiers :");
   scanf("%d %d",&x,&y);
   printf("\n le pgcd de %d et %d est %d\n",x,y,pgcd(x,y));
   printf("\n le pgcd de 50 et 30 est %d\n",pgcd(50,30));
}
```

**Arguments formels :** les variables `a` et `b` sont les paramètres ou arguments formels de la fonction `pgcd`. On les qualifie de *formels* car cette fonction s'applique quelles que soient les valeurs de `a` et `b`. Ces valeurs ne sont connues qu'au moment de l'appel de la fonction. Lorqu'une fonction n'a pas d'arguments formels, les parenthèses sont quand même nécessaires, et l'on peut noter `void` pour signifier l'absence d'arguments.
**Arguments effectifs :** les variables `x` et `y` ainsi que les constantes 50 et 30 sont les paramètres ou arguments effectifs de la fonction `pgcd`.
**Type d'une fonction :** une fonction a toujours un type de retour (`int` pour la fonction `pgcd`). Une fonction qui ne retourne rien (pas de résultat) est appelée procédure et son type de retour est `void`.
**Corps d'une fonction :** il implémente la fonction, autrement dit, il définit son comportement.
**Déclaration d'une fonction :** on appelle déclaration d'une fonction la donnée de son type de retour, de son nom et de la liste de ses arguments formels. Le corps de la fonction est défini ailleurs (plus loin dans le même module, ou bien dans un module différent).

## 4.2 Appels de fonctions et gestion de la pile

Une *pile* est une structure de données ayant le comportement suivant (comme une pile de linge) : on dépose (*empile*) le dernier élément sur le sommet de la pile, et le premier élément reiré (*dépilé*) est le dernier empilé. On parle de pile ou encore de liste LIFO (*Last In First Out*).

Les arguments formels et les variables locales d'une fonction sont stockés dans une zone particulière de la mémoire, applée *segment de pile* ou encore *stack*. Les principes de gestion de la pile doivent être bien assimilés pour pouvoir comprendre les notions de passage de paramètres et la récursivité.

### 4.2.1 Le passage des paramètres par valeurs

Lors de l'appel d'une fonction `C`, le système alloue une portion de la pile dédiée au programme. En simplifiant (beaucoup !), on peut considérer que cette zone mémoire est utilisée pour stocker les arguments, les variables locales automatiques (voir plus loin) et l'adresse de retour de la fonction. A l'appel de la fonction, ses arguments effectifs sont donc recopiés à l'emplacement prévu. Notez que les modifications sur les arguments formels n'affectent donc pas les arguments effectifs.

Voyons les états successifs de la pile lorsque l'on exécute le programme suivant :

```
main() {
   int x=0 y=15;
   z=pgcd(x,y);
}
```

**Etat 0 :** juste avant l'appel de la fonction `pgcd`

| 10 | 15 | 0 |
|---|---|---|
| x | y | z |

**Etat 1 :** passage de paramètres à l'appel de la fonction `pgcd`, recopie des paramètres effectifs et allocation de la variable locale :

| 10 | 15 | 0 | @ | 10 | 15 | 0 |
|---|---|---|---|---|---|---|
| x | y | z | @retour | a | b | c |

**Etat 2 :** calcul du pgcd

– échange des valeurs de `a` et `b`

| 10 | 15 | 0 | @ | 15 | 10 | 0 |
|---|---|---|---|---|---|---|
| x | y | z | @retour | a | b | c |

– premier passage dans la boucle

| 10 | 15 | 0 | @ | 10 | 5 | 5 |
|---|---|---|---|---|---|---|
| x | y | z | @retour | a | b | c |

– deuxième passage dans la boucle

| 10 | 15 | 0 | @ | 5 | 0 | 0 |
|---|---|---|---|---|---|---|
| x | y | z | @retour | a | b | c |

**Etat 3 :**retour de la valeur calculée, restitution de l'espace alloué

| 10 | 15 | 5 |
|---|---|---|
| x | y | z |

Ce mécanisme de passage de paramètres (par recopie des valeurs des arguments effectifs) est appelé *passage par valeurs*. Notez que lors d'un passage par valeur, les arguments effectifs peuvent être des expressions quelconques, il n'est pas nécessaire que ce soit des *lvalue*.

### 4.2.2 Le passage par adresse

Il est parfois utile de conserver les valeurs calculées dans les paramètres formels. C'est le cas lorsqu'une fonction calcule plusieurs valeurs que l'on veut récupérer. Le mécanisme de base du langage `C` est le passage de paramètres par valeurs. Pour avoir un *passage par adresse*, on passe par valeur *l'adresse* d'une *lvalue*. En cons'équence, la fonction ne travaille plus sur une copie de la valeur de l'argument effectif mais directement sur celui-ci. L'exemple ci-dessous reprend la fonction pgcd et la modifie de façon que la valeur du premier paramètre effectif soit modifiée :
**Exemple :**

```
int pgcd(int * a, int b) {
              // noter l'operateur * indiquant le passage par adresse
   int c;
   if (*a<b) {// noter l'opérateur * a chaque occurrence de a
      c=*a;   *a=b;   b=c;
   }
   do {
      c=*a%b; *a=b;  b=c;
   } while (c!=0);
   return *a;
}
int main(){
   int x,y;
   printf("donner deux entiers :");
   scanf("%d %d",&x,&y);
   printf("\n le pgcd de %d et %d est %d\n",x,y,pgcd(&x,y));
             //noter operateur &
   printf("\n le pgcd de 50 et 30 est %d\n",pgcd(50,30)); //INTERDIT!!!
}
```

Pour l'instant contentez vous d'utiliser les opérateurs `*`(contenu de l'adresse) et `&`(adresse) comme indiqué dans l'exemple. Nous les reverrons bienôt dans le chapitre sur les pointeurs.

Voyons les états successifs de la pile lorsque l'on exécute le programme suivant :

```
main() {
    int x=15,y=10;
    z=pgcd(&x,y);
}
```

**Etat 0 :** juste avant l'appel de la fonction `pgcd`

| | 15 | 10 | 0 | |
|---|---|---|---|---|
| | x | y | z | |

**Etat 1 :** passage de paramètres à l'appel de la fonction `pgcd`, recopie de l'adresse de `x` pour `a` (passage par adresse) et recopie du contenu de `y` pour `b` (passage par valeur), allocation de la variable locale :

| | 15 | 10 | 0 | @ | &x | 10 | 0 | |
|---|---|---|---|---|---|---|---|---|
| | x | y | z | @retour | a | b | c | |

**Etat 2 :** calcul du pgcd

- premier passage dans la boucle

| | 10 | 15 | 0 | @ | &x | 5 | 5 | |
|---|---|---|---|---|---|---|---|---|
| | x | y | z | @retour | a | b | c | |

- deuxième passage dans la boucle

| | 5 | 15 | 0 | @ | &x | 0 | 0 | |
|---|---|---|---|---|---|---|---|---|
| | x | y | z | @retour | a | b | c | |

**Etat 3 :**retour de la valeur calculée, restitution de l'espace alloué

| | 5 | 15 | 5 | |
|---|---|---|---|---|
| | x | y | z | |

Au retour de la fonction, la valeur de `y` reste inchangée, par contre `x` a été modifiée.

L'exemple le plus parlant pour le passage de paramètres par adresses est certainement celui de la fonction `switch` dont le rôle est d'échanger les valeurs de ses deux paramètres :

```
void switch(int *a, int *b){
   int c;
   c=*a;
   *a=*b;
   *b=c;
}
```

### 4.2.3 Précautions à prendre

Une erreur commune consiste à "oublier" de retourner une valeur pour une fonction qui est censée en retourner. Le compilateur ne détecte pas ce genre d'erreur. Voici un exemple :

```
int foo(int n) {
  printf("n dans foo %d  ",n); // pas d'instruction return
}

int foo2(int n) {
  printf("n dans foo2 %d  ",n);
  if (n%2==0) return 1; else return 0;
}

int foo3(int n) {
  printf("n dans foo3 %d  ",n);
  if (n%2==0) return 1; // si n est impair pas d'instruction return
}
main(){
  int i;
  for (i=0;i<4;i++)
    printf("foo : %d  foo2 : %d  foo3 :%d\n",foo(i),foo2(i),foo3(i));
}
```

Voici le résultat de l'exécution de ce petit programme :

```
n dans foo3 0  n dans foo2 0  n dans foo 0  foo : 14  foo2 : 1  foo3 :1
n dans foo3 1  n dans foo2 1  n dans foo 1  foo : 14  foo2 : 0  foo3 :1
n dans foo3 2  n dans foo2 2  n dans foo 2  foo : 14  foo2 : 1  foo3 :1
n dans foo3 3  n dans foo2 3  n dans foo 3  foo : 14  foo2 : 0  foo3 :3
```

## 4.3 Classification des données

Jusqu'à présent, on s'est contenté d'associer aux variables un `type`. Ce n'est pas le seul attribut qui caractérise une variable. Il y a aussi sa *classification* qui détermine l'emplacement mémoire qui lui est alloué (lequel détermine sa durée de vie, sa portée).

### 4.3.1 Variables globales et variables locales

Selon l'emplacement de la définition d'une variable, celle-ci est une **variable globale** ou une **variable locale** :

1. toute variable définie à l'extérieur des fonctions est une **variable globale** : elle est connue et utilisable partout dans le fichier où elle est définie, en particulier dans n'importe quelle fonction de ce fichier (sauf si elle est masquée par une variable locale). Sa portée peut même s'étendre à d'autres modules du programme, si celui-ci est constitué de plusieurs fichiers.
2. toute variable définie à l'intérieur d'une fonction (au début d'un bloc) est une **variable locale**. Elle n'est connue qu'à l'intérieur du bloc dans lequel elle est définie.

**Exemple :**

```
int g;           // g variable globale
main() {
   int i;        // i variable locale
   for (i=0;i<100;i++) {
      int j;     // j variable locale
      for (j=1;j<100;j++) {
         g=pgcd(i,j);
         printf("le pgcd de %d et %d est %d\n",i,j,g);
         ...
      }          // fin portee j
      ...
   } // fin portée i
}
```

Attention, il faut être prudent, car on peut très bien définir (mais ce n'est pas conseillé !) une variable locale et une variable globale ayant le même nom. Le compilateur ne génère pas d'erreur, et dans la zone de portée de la variable locale, c'est cette dernière qui est choisie, sinon c'est la variable globale.
**Exemple :** le programme suivant affiche successivement 120 et 100

```
#include<stdio.h>
int g=100;        // g variable globale
main(){
   if (g>0) {
     int g=120;
     printf("g=%d\n",g);  // il s'agit de g locale
   }
```

```
   printf("g=%d\n",g);  // il s'agit de g globale
}
```

Ce qu'il faut retenir :

1. les variables globales sont *statiques* (`static`) c'est-à-dire qu'elles sont permanentes. Le système leur alloue l'espace mémoire nécessaire au démarrage du programme et cet espace est restitué à la fin de l'exécution du programme.
2. les variables locales et les arguments formels des fonctions sont *automatiques*. L'espace mémoire nécessaire est alloué lors de l'activation de la fonction ou du bloc d'instructions correspondants. Il est restitué à la fin du bloc ou de la fonction. Nous le verrons plus loin, les qualificatifs `static` et `register` permettent de modifier l'allocation et la durée de vie de ces variables.

Les arguments formels des fonctions se comportent comme des variables locales avec la particularité d'être automatiquement initialisés par les arguments effectifs lors de l'activation de la fonction.

### 4.3.2 Les variables locales statiques

Le qualificatif `static` placé devant la définition d'une variable locale la rend statique, c'est-à-dire permanente, **mais sa visibilité reste locale**. Elle n'est donc accessible que dans le bloc où elle est définie. L'intérêt des variables locales statiques est d'interdire les modifications inconsidérées (comme cela peut arriver avec des variables globales) et de n'autoriser ces modifications qu'à l'intérieur du bloc où elles sont définies.

Il faut utiliser le qualificatif `static` avec beaucoup de précautions. A éviter avec les fonctions récursives.

**Exemple :** le programme ci-dessous affiche successivement,

```
s vaut : 1
s vaut : 3
s vaut : 7
s vaut : 15
s vaut : 31
```

```
#include <stdio.h>
void compte(int i) {
   static int s=0;
   s=s+i;
   printf("s vaut %d\n",s);
}
main(){
   int i;
   for(i=1;i<20;i=i+i) compte(i);
}
```

### 4.3.3 Les variables locales critiques (`register`)

Le qualificatif `register` indique que la variable joue un rôle critique pour le programme, on indique donc au compilateur d'optimiser son utilisation (allocation d'une ressource à accès rapide). Les variables qualifiées de `register` doivent être locales et d'un type élémentaire. Elles sont automatiquement initialisées à 0. L'optimisation se fait différemment selon la plateforme, dans la plupart des cas, le stockage se fait dans des registres de la machine, pas en mémoire centrale (donc pas d'adresse et impossible de leur appliquer l'opérateur &).

Ce qualificatif peut et doit être évité avec des compilateurs récents, qui sont assez sophistiqués pour déterminer les variables critiques et procéder aux optimisations adéquates.

### 4.3.4 Les constantes (`const`)

Le qualificatif `const` placé devant la définition d'une variable précise au compilateur que la variable ou l'argument formel ne changera pas de valeur dans la portée de sa déclaration. Cela permet au compilateur d'effectuer des traitements plus efficaces, mais aussi au programmeur d'écrire du code plus sûr. Il est conseillé d'utiliser ce qualificatif chaque fois que possible.

**Exemple**

```
int main(){
   const int i=3;  // i initialise a 3
   i=4;  //INTERDIT !!!
}
```

Cet exemple montre que l'initialisation n'est pas une affectation.

# Chapitre 5

# Types structurés

Nous avons vu jusqu'ici les types élémentaires (on dit aussi primitifs) du langage `C`. A partir de ces types de base, le langage permet de définir des structures plus riches appelées types de données structurées. Les deux plus courants sont les *tableaux* et les *structures*.

## 5.1 Les tableaux

Ce sont des suites d'éléments homogènes (**de même type**) rangés de façon consécutive en mémoire. Le nombre d'éléments est fixe et est appelé la *taille* du tableau. Les tableaux sont de type quelconque (mais homogène) : type élémentaire, pointeur, tableau, structure. Lorsque l'élément d'un tableau n'est pas lui-même un tableau il s'agit d'un tableau unidimensionnel. Dans le cas contraire, on parle de tableau multidimensionnel.

### 5.1.1 Les tableaux à une dimension

Un tableau unidimensionnel est composé d'éléments d'un type donné. La donnée de ce type et de la taille du tableau permet de représenter le tableau en mémoire. Par exemple, la définition *type* `tab[5] ;` alloue en mémoire une zone qui peut contenir 5 éléments du type *type*, c'est-à-dire `5*sizeof(`*type*`)`. Les éléments du tableau sont désignés par `tab[0], tab[1]`...`tab[4]`. **Notez que le premier élément se trouve à l'indice 0.**

La taille d'un tableau (5 dans l'exemple qui précède) peut être spécifiée par une expression à condition qu'à la compilation on puisse connaître sa valeur. Il y a deux cas où la taille du tableau peut être omise :

- lors d'une déclaration et non d'une définition,
- lorsqu'un tableau est déclaré comme argument d'une fonction.

Remarques importantes :

- **Le nom d'un tableau est équivalent à une constante**, il peut donc figurer dans n'importe quelle expression où une constante est autorisée. Il joue le même rôle que l'adresse du premier élément du tableau. Ainsi les deux expressions

suivantes sont identiques sauf au moment de la déclaration et quand il est argument de `sizeof` : `tab` et `&tab[0]`.
Ce qui précède est important, car cela montre bien la philosophie adoptée par le langage `C` pour ce qui concerne les tableaux. Excepté au moment de la déclaration et du calcul de la taille, `C` ignore l'espace mémoire occupé par un tableau : **il appartient donc au programmeur de s'assurer qu'il n'y a pas de** *débordement*.
Dans l'exemple du tableau `tab` qui a 5 éléments, rien n'interdit d'accéder à `tab[10]`. Voilà qui est dangereux et qui, de fait, constitue une source de bon nombre d'erreurs.

– le nom d'un tableau doit être considéré comme une constante (et donc pas une *lvalue*). Il est donc interdit d'affecter un tableau à un autre (`tab1=tab2` produira une erreur de compilation, car `tab1` n'est pas modifiable). Si l'on veut affecter le contenu d'un tableau à un autre, il faut le faire élément par élément.

**Initialisation :** Il y a plusieurs façons d'initialiser un tableau :

1. écrire une boucle d'initialisation :

```
int tab[10];
int i;
...
for (i=0;i<10;i++) tab[i]=...;
```

2. initialiser tous les éléments au moment de la définition (avec des constantes)

```
int tab[10]={0,1,2,3,4,5,6,7,8,9};
```

3. initialiser les premiers éléments au moment de la définition (avec des constantes)

```
int tab[10]={0,1};
```

4. initialiser tous les éléments au moment de la définition (avec des constantes), la taille est automatiquement calculée :

```
int tab[]={0,1,2,3,4,5,6,7,8,9};
```

### 5.1.2 Passage d'arguments de type tableau

On l'a vu lors de la présentation des fonctions, un passage par valeur consiste à recopier le paramètre effectif et à travailler sur cette copie. Lorsque ce paramètre est un tableau, c'est l'adresse du premier élément qui est passée en argument. Dans la déclaration d'un argument formel `tab` de type tableau,

– le nombre d'éléments est sans intérêt,
– les expressions *type* `tab[]` et *type* `*tab` sont équivalentes,
– la fonction peut recevoir soit un tableau soit un pointeur comme argument effectif.

**Exemple :** trois versions d'une même fonction (longueur d'une chaîne de caractères) :

```
int strlen(char chaine[]){
   int i=0;
   while (chaine[i]!=0) i++;
   return i;
}

int strlen(char *chaine){
   int i=0;
   while (chaine[i]!=0) i++;
   return i;
}

int strlen(char *chaine){
   int i=0;
   while (*(chaine+i)!=0) i++;
   return i;
}
```

**Il n'est pas possible de passer un tableau par valeur, de façon simple.**

De même, lorsque le retour d'une fonction est un tableau, ce qui est transmis est l'adresse du premier élément du tableau. Il faut donc s'assurer que le tableau transmis n'est pas un tableau local qui disparaît à la fin de la fonction.

**Exemple :**

```
tab * fonc(void) {
   int tab[10];
   ...
   return tab; // ce tableau n'existera plus à la fin de la fonction
}
```

**Une fonction de doit jamais retourner un tableau défini localement.**

### 5.1.3 Les tableaux multidimensionnels

Ce sont des tableaux dont les éléments sont eux mêmes des tableaux (par exemple des matrices). La déclaration d'un tableau multidimensionnel se fait comme suit :

```
type tab[index1][index2]...[indexN];
type tab[index1][index2]...[indexN]={...};
```

où `index1` représente la première dimension (le nombre d'éléments), etc...

Par exemple une matrice de dimension 3*4 est un tableau de 3 tableaux (les lignes) de 4 éléments, on la définit comme suit :

```
int matrice[3][4];
```

A une telle matrice est associée une zone mémoire de `3*4*sizeof(int)` octets consécutifs. L'élément d'indices (`i,j`) de la matrice est noté `matrice[i][j]` où `i` est (par convention) le numéro de ligne et `j` le numéro de colonne :

```
int matrice[3][4]={0,1,2,3,4,5,6,7,8,9,10,11}
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

en mémoire,

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

## 5.2 Les chaînes de caractères

En `C` les chaînes de caractères sont considérées comme des tableaux de caractères. La fin de la chaîne est repérée par le caractère '\0'. De même que les tableaux, les chaînes peuvent être initialisées lors de leur définition. Enfin, on peut utiliser les pointeurs pour représenter les chaînes de caractères (**Note :** dans la norme ANSI, il est interdit de modifier les constantes de type chaîne (voir l'option `-fwritable-strings` du compilateur)) :

```
char chaine1[100]={'b','o','n','j','o','u','r'};
// initialisation des 6 1ers caracteres
char chaine1[100]="bonjour";
// raccourci equivalent
char chaine2[]={'b','o','n','j','o','u','r'};
// chaine de longueur 7+1
char chaine2[]="bonjour";
// raccourci equivalent
char * chaine3="bonjour";
// attention pointeur vers une constante de type chaine
```

La bibliothèque standard offre diverses fonctions de manipulation des chaînes de caractères. Vous devez inclure le fichier d'entête `<string.h>` pour récupérer les prototypes de ces fonctions. Voici les plus utilisées :

- `char *strcat(char *s1, const char *s2) ;`
  concatène deux chaînes (le contenu de `s2` est rajouté à la fin de `s1`). Les chaînes ne doivent pas se chevaucher, et la chaîne destinataire (`s1`) doit être assez grande pour contenir le résultat.
- `int strcmp(const char *s1, const char *s2) ;`
  compare deux chaînes, selon l'ordre lexicographique (retourne une valeur négative si `s1` est inférieure à `s2`, nulle si les deux chaînes sont égales, positive si `s1` est supérieure à `s2`).
- `char *strcpy(char *s1, const char *s2) ;`
  copie la chaîne pointée par `s2` (y compris le caractère final) dans la chaîne `s1`, et retourne celle-ci. Les deux chaînes ne doivent pas se chevaucher et `s1` doit être assez grande pour recevoir la copie.

- `size_t strlen(const char *s) ;`
  retourne la longueur de la chaîne `s` sans compter le caractère de fin de chaîne.
- `char *strchr(const char *s, int c) ;`
  retourne un pointeur sur la première occurence du caraère `c` dans la chaîne `s`.

Voici maintenant une petite illustration de ces fonctions :

```
#include <string.h>
int main(){
  char s1[100]="bonjour";  char s2[]=" toi";
  char *s3, *s4;
  s3=strcat(s1,s2);
  printf("s3 = %s\t",s3);   printf("s1 = %s\n",s1);
  printf("s1 comparée à s2 : %d\n",strcmp(s1,s2));
  printf("s2 comparée à s1 : %d\n",strcmp(s2,s1));
  printf("s2 comparée à s2 : %d\n",strcmp(s2,s2));
  s4=strcat(s1,s2);
  printf("s4 = %s\n",s4);  printf("s1 = %s\n",s1);
  printf("longueur de s2 : %d\n",strlen(s2));
}
```

Et le résultat :

```
s3 = bonjour toi s1 = bonjour toi
s1 comparée à s2 : 66
s2 comparée à s1 : -66
s2 comparée à s2 : 0
s4 = bonjour toi toi
s1 = bonjour toi toi
longueur de s2 : 4
```

## 5.3 Les structures

Ce sont des éléments composés de plusieurs champs de types différents. Il est courant d'avoir à construire des types de données de ce genre. Par exemple, imaginons que l'on veuille écrire une application de gestion des étudiants, il faudrait se préoccuper de la représentation des données. On pourrait décider de définir un type `etudiant` composé des champs `nom` et `prenom` (chaînes de caractères) et `age` (entier). Autre exemple classique, un point du plan est caractérisé par une abscisse et une ordonnée...

On définit un type structure de la façon suivante :

```
struct nom {
    type1 champ1;
    type2 champ2;
    ...
    typeN champN;
};
```

Par la suite, une variable pourra être déclarré de ce type : `struct nom x ;`. Voici l'exemple du point du plan :

```
struct point {
   float x;
   float y;
};
struct point A,B;
```

ou bien,

```
struct point {
   float x;
   float y;
} A,B ;
```

On peut aussi omettre de nommer la structure si l'on sait que l'on ne s'en reservira pas :

```
struct {
   float x;
   float y;
} A,B ;
```

**Accès aux champs d'une structure :** c'est l'opérateur de sélection "`.`" qui permet l'accès aux champs d'une structure. Dans l'exemple de la variable `A` déclaré comme un `point`, son champ `x` est `A.x`.

**Initialisation :** comme pour les tableaux, une structure peut être initialisée lors de sa déclaration :

```
struct point {
    float x;
    float y;
} A={1.0,2.3}, origine={0,0};
```

**Manipulations globales :** on peut affecter une variable d'un type *structure* à une autre de même type, cette affectation se fait alors pour chacun des champs :

```
struct point {
    float x;
    float y;
} A={1.0,2.3}, origine={0,0},centre;
centre=origine;
```

**Stockage en mémoire :** `C` garantit que les champs d'une structure sont disposés de façon contigüe en mémoire, dans l'ordre où ils sont déclarés dans la structure. Mais attention, la taille obtenue par l'opérateur `sizeof` peut paraître incohérente. En effet, dans le stockage des structures, il y a un "offset" (par rapport au début de la structure) affecté par le compilateur à chaque champ de la structure, ceci pourdes contraintes d'alignement dont nous ne parlerons pas ici. Il peut ainsi avoir des "zones invisibles" la représentation d'une structure :

```
struct {
    int d1;
    char c;
    int d2;
} exemple;
```

La taille de cette structure (sur ma machine, un Pentium II) est de 12 (résultat de `sizeof(exemple)`), alors que la somme de la taille de ses champs est 9 (4 + 1+ 4).

**Passage d'argument par valeur :** contrairement aux tableaux, les structures sont recopiées champ par champ lors d'un passage d'argument par valeur. Le retour d'une fonction peut très bien être une structure. Une astuce pour transmettre un tableau par valeur consiste à l'encapsuler dans une structure :

```
struct encapsule {int tab[10];};
void modifier(struct encapsule s){
   s.tab[1]=-999;
}
void main(){
   struct encapsule c;
   c.tab[1]=0;
   modifier(c);
   printf("c.tab[1]=%d\n",c.tab[1]); // la valeur affichee est 0
}
```

**Passage d'argument par adresse :** on le reverra dans le chapitre sur les pointeurs. L'argument effectif est l'adresse de la structure. Pour atteindre le contenu de chaque champ, on utilise l'opérateur de sélection `->` :

```
struct point {
    float x;
    float y;
} ;
void translater(struct point* s,float t1,float t2){
  s->x=(s->x)+t1;
  s->y=(s->y)+t2;
}
void main(){
  struct point A={0,0};
  translater(&A,10,10);
   printf("A est (%f,%f)\n",A.x,A.y); // affiche (10.000000,10.000000)
}
```

## 5.4 Unions

Les champs des structures se suivent sans chevauchement, alors que les champs des unions se chevauchent. Ainsi, une variable peut avoir plusieurs types différents. Une définition d'union a la même syntaxe que celle d'une structure, le mot clé `struct` étant remplacé par `union`.

```
union nombre {
   int i;
   float f;
}
```

Une variable de type union n'a, à un instant donné, qu'un seul membre ayant une valeur. Ainsi, si l'on déclare `union nombre n ;`, la variable `n`aura soit une valeur entière, soit une valeur flottante, mais pas les deux à la fois.

**Accès aux membres d'une union :** on utilise le même opérateur (`.`) que dans le cas des structures. Dans l'exemple qui précède, si l'on veut affecter une valeur entière à `n` on écrira `n.i=10 ;`, si on veut lui affecter une valeur flottante, `n.f=3.14159`.

**Utilisation des unions :** comme il n'y a pas moyen de savoir quel est le champ valide d'une union, en pratique on l'encapsule avec un indicateur dans une structure :

```
enum type{Entier,Flottant};
struct nombre{
   enum type typeNb;
   union {
     int i;
     float f;
   } u;
};
...
struct nombre a1,a2;
...
a1.typ_val=Entier;
a1.u.i=10;
a2.typ_val=Flottant;
a2.u.f=3.14159;
}
```

## 5.5 Types énumérés

Ils permettent une définition pratique d'un ensemble fini de valeurs. Ce n'est pas un type structuré. En fait il s'agit simplement de désigner des entiers par des noms symboliques :

```
enum jour {lundi,mardi, mercredi, jeudi, vendredi, samedi, dimanche} x;
enum jour y,z;
```

Les noms symboliques donnés dans la définition d'une énumération sont en correspondance avec les entiers, ainsi, dans l'exemple qui précède, `lundi` correspond à 0, `mardi` à 1,... On peut altérer cette correspondance de la façon suivante :

```
enum jour {lundi=2,mardi,mercredi=6,jeudi, vendredi,samedi,dimanche} x;
```

`lundi` correspond alors à 2, `mardi` à 3,`mercredi` à 6, `jeudi` à 7...

## 5.6 Synonymes de types et typedef

Le langage offre la possibilité de donner un nom à un type. Pour cela, on utilise le mot clé `typedef` suivi d'une construction ayant la même syntaxe qu'une déclaration de variable. L'identificateur utilisé est le nom du type :

```
type def int TableauEntiers[10];
```

déclare `TableauEntiers` comme étant le type tableau de 10 entiers ;

```
type struct {
     char nom[20];
     int num_ss;
} Individu;
```

déclare `Individu` comme étant un type structure à deux champs : un tableau de 20 caractères et un entier. Ces noms de type sont ensuite utilisables dans les déclarations de variables, comme les types de base :

```
TableauEntiers t1,t2;
individu p1,p2;
```

On utilisera beaucoup la définition de types pour les structures. En voici une autre utilité classique :

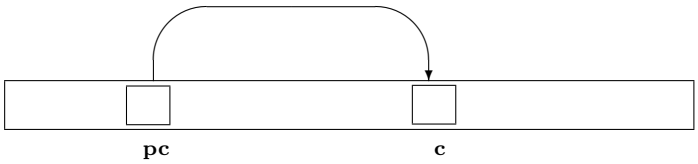
```
typedef enum {FAUX,VRAI} Boolean ;
Boolean b1;
int main(){
  int x;
  printf("entrer un entier :");
  scanf("%d",&x);
  b1=(x>10);
  if (b1==VRAI) printf("x est supérieur à 10\n");
  else printf("x est inférieur ou égal à 10\n");
}
```

# Chapitre 6

# Les pointeurs

## 6.1 Introduction

On peut dire, en simplifiant beaucoup, qu'un programme en cours d'exécution possède un tableau de cases mémoire (octets) consécutives que l'on manipule individuellement ou par groupe. C'est ce tableau qui constitue la mémoire alouée au programme. Une variable de type pointeur est destinée à contenir l'adresse d'une autre variable. Par exemple, si `c` est une variable de type caractère et `pc` un pointeur vers `c`, on peut représenter cette situation par la figure ci-dessous :




L'opérateur unaire `&`, appelé opérateur d'adressage, donne l'adresse d'une variable. Ainsi, on écrira, `pc = &c`, pour affecter l'adresse de `c` à `pc` On dit que `pc` *pointe* sur (ou vers) `c`. Attention, l'opérateur `&` ne s'applique que sur des objets en mémoire, il ne peut s'appliquer à des constantes ou à des variables déclarées comme `register`.

L'opérateur unaire `*`, appelé opérateur d'indirection, (ou de déréférence), s'applique à un pointeur et donne accès à l'objet pointé par ce pointeur (contenu de l'adresse pointée).

```
int x=1, y=2, z[10] ;
int *pi ;              pi pointeur sur un int
pi = &x ;              pi pointe sur x
y = *pi ;              y est affecté à la valeur de l'objet pointé par pi (x)
*pi=0 ;                x vaut 0
pi = &z[0] ;           pi pointe vers le premier élément du tableau z
```

La définition du pointeur `int *pi` spécifie que `*pi` est un `int`. Un pointeur adresse nécessairement un objet de type bien défini. Il y a néanmoins une exception avec les pointeurs `void *` que l'on verra plus loin.

Si `pi` pointe sur une variable `x`, alors `*pi` et `x` désignent la même chose, et l'on peut utiliser l'une ou l'autre des notations :

`*pi = *pi +3 ;` équivaut à `x=x+3`

Les pointeurs sont essentiels pour les passages de paramètres par adresse, mais aussi pour les allocations dynamiques décrites plus loin dans ce chapitre.

Notez que l'on peut affecter à un pointeur de n'importe quel type la valeur 0, qui indique que le pointeur n'est pas encore dirigé vers un objet particulier. La valeur 0 dans ce cas est souvent appelée `NULL` (défini dans `stdio.h` : comme équivalent de `((void *)0)` et de `(0)`).

## 6.2 Pointeurs et structures

L'exemple ci-dessous décrit l'utilisation des pointeurs sur les structures.

```
typedef struct maStruct {
                        int a;
                        int * b;
            } maStruct;
maStruct X, *Y;
X.a = 55;
X.b = new int;          // pointeur sur var simple
*(X.b) = 77;            // equivalent mais plus lsible que * X.b = 77;
Y = new rec;            // pointeur sur une struct
Y -> a = 55;            // equivalent mais plus lisible que (*Y).a = 55;
Y -> b = (int*) malloc (sizeof(int));  // allocation
*(Y -> b) = 77;       // equivalent à *Y -> b = 77; et à *(*Y).b = 77;
```

## 6.3 Pointeurs et passage d'arguments par adresse

Nous l'avons vu dans la section 4.2.2, pour que les modifications effectuées sur les arguments formels soit conservée sur les arguments effectifs, il faut procéder à un passage par adresse.

**Le passage par adresse d'un argument est simplement le passage par valeur de son adresse.**

## 6.4 Pointeurs et tableaux

Les pointeurs et les tableaux sont très similaires en `C`. A part les tableaux multidimensionnels, toute opération effectuée par indexation sur un tableau peut être aussi réalisée avec des pointeurs. Après les instructions suivantes :

`int tab[10] ; int * pi ;`

`pint = &tab[0] ;` ou bien `pi=tab ;`

`tab` et `pi` pointent tous les deux sur le même élément (le premier). Si `pi` pointe sur le ième, alors `*(pi+1)` et `*(pi-1)` désignent respectivement les (i+1)ème et (i-1)ème éléments du tableau. On écrira indifféremment `tab[i]` ou `*(pi+i)`. De même, `&tab[i]` et `pi+i` sont équivalents.


**Attention,** la correspondance entre pointeurs et tableaux est étroite, mais ils ne sont pas équivalents, notez bien que :
- un tableau alloue de la place en mémoire,
- un tableau n'est pas une *lvalue* et ne peut donc pas figurer à gauche du signe d'affectation.

## 6.5 Arithmétique des pointeurs

**Opérateurs + et -, ajouter ou soustraire un entier à un pointeur :** On l'a vu précédemment, les opérateurs `+` et `-` permettent de réaliser la somme et la soustraction d'un pointeur et d'un entier. Mais une telle opération n'a de sens que si le pointeur repère un élément de tableau. Notez que si `p` est un pointeur vers un tableau dont les éléments sont des `struct` (ou d'autre type), l'écriture `p+1` désigne toujours l'élément suivant, quelle que soit sa taille. Ainsi, la notation `p+2` peut être comprise comme `p+2*sizeof(*p)` en octets.

**Opérateurs ++ et − :** On peut les appliquer à des pointeurs (cela revient à ajouter et soustraire des pointeurs). Cette opération est classique dans le cas de parcours de tableaux. Exemple :

```
int main(){
  char msg[]="Bonjour !";
  char *p;
  for (p=msg; *p!='\0';p++) printf("%c",*p);
}
```

**Opérateurs < et > , comparer des pointeurs :** Cela peut se faire, une fois encore dans le cas de deux pointeurs (`p` et `q`) sur un tableau, alors `p<q` est vrai si `p` pointe vers un élément qui précède l'élément pointé par `q`.

## 6.6 Pointeurs et tableaux multidimensions

Si l'on définit la matrice `M` somme suit :

```
int M[3][5]={{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}};
```

Le nom du tableau `M` est l'adresse du premier élément, et pointe sur le tableau (eh oui !) `M[0]` qui vaut {`0,1,2,3,4`}. L'expression `M+1` est l'adresse du deuxième élément `M[1]`. La question se pose de savoir comment accéder à l'adresse des éléments `M[i][j]` de notre matrice. Rappelons que le stockage est contigü, donc l'élement `M[i][j]` se trouve à la position `i*nb_col +j` (ici, `i*5+j`), son adresse est donc `M+i*5+j`.

Pour travailler avec des pointeurs dans un tableau à deux dimensions, il faut :

- l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau (ici `int`, car `M` est l'adresse d'un tableau),
- la longueur d'une ligne réservée en mémoire (c'est à dire le nombre de colonnes),
- le nombre d'éléments effectivement utilisés dans une ligne (attention aux tableaux où l'on utilise pas tout l'espace réservé !),
- le nombre de lignes effectivement utilisées.

**Exemple :**

```
int main(){
  int M[4][5]={{0,1,2,3},{4,5,6,7},{8,9,10,11}};
  int *p;
  int i,j;
  p=(int *)M; // conversion de type
  for (i=0;i<3;i++) {
   for (j=0;j<4;j++)
     printf("M[%d][%d]=%d\t",i,j,*(p+i*5+j));
                          // attention au nb de col.reserve
   printf("\n");
  }
  printf("--------------\n");
  for (i=0;i<3;i++) {
   for (j=0;j<4;j++)
     printf("M[%d][%d]=%d\t",i,j,*(M[i]+j));
   printf("\n");
  }
  printf("--------------\n");
  for (i=0;i<20;i++) {
   printf("%d ",*(p+i));
 }
 printf("\n");
}
```

ce programme affiche :

```
M[0][0]=0       M[0][1]=1       M[0][2]=2       M[0][3]=3
M[1][0]=4       M[1][1]=5       M[1][2]=6       M[1][3]=7
M[2][0]=8       M[2][1]=9       M[2][2]=10      M[2][3]=11
--------------
M[0][0]=0       M[0][1]=1       M[0][2]=2       M[0][3]=3
M[1][0]=4       M[1][1]=5       M[1][2]=6       M[1][3]=7
```

```
M[2][0]=8       M[2][1]=9       M[2][2]=10      M[2][3]=11
--------------
0 1 2 3 0 4 5 6 7 0 8 9 10 11 0 0 0 0 0 0
```

## 6.7 Le type `void`

Le type `void` se comporte syntaxiquement comme un type fondamental. Mais il n'existe aucun objet de ce type ! Il sert à préciser qu'une fonction ne renvoie aucune valeur. Le type `void *` est le type *pointeur générique*, il est utilisé comme type de base pour des pointeurs sur des objets de type inconnu. Ainsi, un pointeur de n'importe quel type peut être affecté à un pointeur `void *`. Cela est utilisé pour des fonctions d'allocation de mémoire qui rendent un pointeur vers l'objet alloué. L'exemple classique est celui de la fonction `malloc` de la bibliothèque standard dont le prototype est

```
void * malloc(size_t size) ;
```

Nous verrons cette fonction dans la section suivante 6.8.

## 6.8 Allocation Dynamique

Lorsqu'on utilise des tableaux, on a vu que leur taille devait être définie à la compilation, ce qui peut être un inconvénient quand on ne connait pas à priori cette taille. Une solution consiste donc à surdimensionner les tableaux pour être sûr de ne pas avoir de phénomènes de débordement. Evidemment cela n'est pas satisfaisant (occupation inutile de la mémoire).

La fonction `malloc` permet d'allouer dynamiquement (au moment de l'exécution) de l'espace mémoire. Ainsi,

```
int * tab;
tab=(int *)malloc(10*sizeof(int));
```

demande l'allocation de 10 fois le nombre d'octets nécessaires au stockage d'un `int` et `tab` désigne l'adresse du premier élément. Mais **attention**, cela n'est pas équivalent à la définition `int tab2[10]`. Notez les différences fondamentales suivantes :

- l'espace mémoire nécessaire au stockage de `tab2` est alloué automatiquement, alors que la définition de `tab` n'alloue que l'espace nécessaire au pointeur, il appartient au programmeur d'allouer l'espace nécessaire au tableau de 10 entiers,
- `tab2` désigne l'adresse du premier élément du tableau et c'est une constante, alors que `tab` est une variable de type pointeur vers un entier et peut changer de valeur (il désigne aussi le premier élément du tableau, sauf modification utltérieure),
- la durée de vie de `tab2` dépend de son statut, mais l'espace qui lui est alloué peut être restitué à la fin de la fonction qui le définit, alors que l'espace alloué par `malloc` ne peut être restitué que par un appel à la fonction `free`.

# Chapitre 7

# Entrées/sorties

Un fichier est constitué d'une suite de données et est vu par un programme `C` comme un flot, c'est-à-dire une suite d'octets qui, selon le cas, constitue un fichier *texte* ou un fichier *binaire*. On peut voir un fichier `C` comme un tableau d'octets. Un fichier *texte* est un fichier structuré en une suite de lignes et est exploitable par divers programmes (éditeur de texte, logiciel d'impression, ...) alors qu'un fichier *binaire* n'a aucune structure et n'est pas exploitable par des programmes externes qui ignorent la structure réelle du fichier.
Nb : la partie concernant les entrées/sorties formatées est largement inspirée du support de B.Cassagne (http ://www.linux-kheops.com/doc/ansi-c).

## 7.1 Mémoire tampon

Quelque soit l'opération que l'on effectue sur un fichier (lecture ou écriture), les données transitent par une zone de mémoire temporaire appelée *mémoire tampon*, ou *tampon*. Le programmeur n'a pas à se préoccuper de la gestion de ce tampon, c'est le système qui en a la responsabilité. L'utilisation d'un tampon permet d'optimiser le fonctionnement des entrées-sorties car on y accède bien plus rapidement qu'au fichier. Par défaut, à tout fichier est associé un tampon de taille prédéfinie. Le programmeur peut spécifier cette taille à l'aide de la fonction `setvbuf` (s'il spécifie une taille nulle, cela revient à se passer du tampon).

Si l'utilisation d'un tampon augmente les performances, elle produit un décalage dans la chronologie (l'opération effective de lecture ou écriture dans le fichier n'est pas contrôlable). Il s'en suit qu'une interruption brutale d'un programme peut conduire à des pertes d'informations dans des fichiers (sauf si la taille du tampon est nulle !).

## 7.2 Fonctions générales sur les flots

Les flots sont désignés par des variables de type `FILE *`, pointeur sur une structure de type `FILE`. Ce type est défini dans le fichier d'interface `stdio.h`. La définition de ce type est dépendante du système d'exploitation, mais correspond toujours à une structure (on parle de *descripteur*) qui contient les informations suivantes :

– l'adresse de la mémoire tampon,
– position de la tête de lecture/écriture
– type d'accès au fichier (lecture, écriture),
– état d'erreur,
– ...

Pour utiliser un fichier dans un programme `C`, il faut donc inclure le fichier `stdio.h` et définir une variable de type `FILE *` :

```
#include <stdio.h>
FILE * fic1;
```

**Ouvrir un fichier, la fonction `fopen` :** Pour utiliser un fichier, il faut l'ouvrir avec la commande `fopen` dont voici le prototype :

```
FILE * fopen(char *, char *);
```

L'ouverture d'un fichier initialise le descripteur décrit plus haut. On fournit à la fonction `fopen` le nom du fichier (premier paramètre, c'est une chaîne de caractères) et le mode d'accès sur ce fichier. Les opérations que l'on veut réaliser, les plus utilisées sont listées ci-desssous, pour plus de détails voir le manuel UNIX (commande `man fopen`). La fonction retourne un pointeur vers une structure de type `FILE` qui sera utilisée par la suite pour les opérations d'entrées-sorties. C'est ce pointeur qui identifie le fichier que l'on utilise.

| | |
|---|---|
| "r" | ouvrir le fichier en lecture seule<br>erreur si le fichier n'existe pas |
| "w" | ouvrir le fichier en écriture seule<br>création du fichier s'il n'existe pas,<br>sinon, son contenu est écrasé |
| "a" | ouvrir le fichier en ajout (écriture en fin de fichier)<br>création du fichier s'il n'existe pas |

Si l'ouverture s'est avérée impossible, `fopen` rend la valeur `NULL`
Exemple :

```
#include <stdio.h>
FILE *fic;
fic = fopen("donnees","r")); // donnees est le nom d'un fichier
if (fic== NULL)
   printf("Impossible d'ouvrir le fichier donnees\n");
```

**Fermer un fichier, la fonction `fclose` :** Même si le système se charge en principe de fermer tous les fichiers ouverts, à la fin de l'exécution d'un programme, il est vivement conseillé de fermer les fichiers explicitement à la fin de leur utilisation. Cela permet de réduire le nombre de descripteurs ouverts et de se prémunir contre toute interruption brutale du programme. Pour cela on utilise la fonction `fclose` dont voici le prototype :

```
int fclose(FILE *);
```

Le paramètre est un pointeur vers une structure de type `FILE` (celui qui vous a été retourné par la fonction `fopen`). La fonction `fclose` retourne la valeur `EOF` si la fermeture s'est mal passée, 0 sinon.

**Vider le tampon, la fonction `fflush` :** Cette fonction provoque l'écriture immédiate du contenu du tampon, elle retourne EOF en cas d'erreur, 0 dans les autres cas :

```
int fflush(FILE *flot);
```

**Tester la fin de fichier, la fonction `feof` :** Cette fonction teste l'indicateur de fin de fichier du flot spécifié et retourne une valeur non nulle si la fin de fichier est atteinte :

```
int feof(FILE *flot);
```

**D'autres fonctions :** `tmpfile` qui permet la création d'un fichier temporaire, `setvbuf` qui permet de redéfinir la politique de gestion du tampon d'entrées/sorties...

## 7.3 Les unités standards d'entrées/sorties

Par défaut, il existe trois flots prédéfinis qui sont ouverts, ce sont les flots d'entrées/sorties standards appelés `stdin` `stdout`, `stderr` :

- `stdin` l'unité standard d'entrée (le clavier),
- `stdout` l'unité standard de sortie (l'écran),
- `stderr` l'unité standard de sortie erreur (l'écran).

Même si un programme n'utilise que ces trois flots prédéfinis, sous UNIX, il est possible (sans rien modifier dans le programme) de modifier les flots par défaut, en utilisant les redirections sur la ligne de commande du lancement de programme...

## 7.4 Lecture et écriture en mode caractères

**Ecriture, la fonction `fputc` :** Cette fonction transfère le caractère spécifié dans le fichier représenté par le flot passé en paramètre. La valeur de retour est le caractère ou EOF en cas d'erreur :

```
int fputc(int car, FILE *flot);
```

**Lecture, la fonction `fgetc` :** Cette fonction retourne le caractère lu dans le fichier représenté par le paramètre flot. En cas d'erreur ou de fin de fichier, elle retourne la valeur EOF :

```
int fgetc(FILE *flot);
```

**Ecriture sur sortie standard, la fonction `putchar` :** C'est un synomyme de `fputc(c,stdout)` :

```
int putchar(int c);
```

**Lecture sur entrée standard, la fonction `getchar` :** C'est un synomyme de `fgetc(stdin)` :

```
int getchar();
```

Il existe une fonction `getc` qui est identique à `fgetc` (même interface, même sémantique), mais `getc` est implémentée comme une macro, donc à priori plus efficace. Il faut faire attention (et éviter) les effets de bords. De même, il existe la macro `putc`.

## 7.5 Lecture et écriture en mode chaînes

**Ecriture, la fonction `fputs` :** Cette fonction écrit une chaîne de caractères dans le flot spécifié (elle n'écrit pas le caractère '\0'), elle retourne une valeur non négative si aucune erreur ne s'est produite, EOF sinon :

```
int fputs(const char *s, FILE *flot);
```

**Lecture, la fonction `fgets` :** Cette fonction lit une chaîne de caractères dans le flot spécifié et la range dans un tampon défini (et alloué) :

```
char * fgets(char *s, int taille, FILE *flot);
```

Plus précisemment, la fonction lit `taille-1` caractères dans le fichier, ou lit jusqu'au caractère `'\n'` (une ligne tout au plus), et :

- `s` est de type pointeur vers `char` et pointe vers un tableau de caractères (alloué).
- `taille` est la taille en octets du tableau de caractères pointé par `s`
- `flot` est de type pointeur sur `FILE` et pointe vers le fichier à partir duquel se fait la lecture.
- la valeur rendue est le pointeur `s` en cas de lecture sans erreur, ou `NULL` dans le cas de fin de fichier ou d'erreur.

**Ecriture sur sortie standard, la fonction `puts` :** Cette fonction écrit sur le flot `stdout` la chaîne de caractères spécifiée :

```
int puts(char *s);
```

La valeur de retour est non négative si l'écriture se passe sans erreur, et EOF en cas d'erreur. Cette fonction est équivalente à la fonction `fputs` utilisant le flot de sortie standard.

**Lecture sur entrée standard, la fonction `gets` :** Cette fonction lit sur le flot d'entrée `stdin`. Cette fonction est équivalente à `fgets` sur le flot `stdin`. Mais attention, il n'y a pas le paramètre `taille` qui donne la taille du tableau pointé par chaîne. La fonction `gets` ne fait donc aucune vérification de débordement du tableau de caractères. **Pour cette raison l'usage de `gets` est déconseillé**.

## 7.6 Lecture et écriture formatées

Les fonctions `fprintf` et `fscanf` permettent de faire des lectures et écritures formatées. Les fonctions `printf` et `scanf` sont des raccourcis pour les flots d'entrée/sortie standards.

### 7.6.1 Ecriture formatée avec `fprintf`

La fonction `fprintf` admet un nombre variable de paramètres :
`int fprintf (`*flot* ,*format* , *param1* , *param2* , ... ,*paramN* )
avec,
- *flot*, de type pointeur sur `FILE`, le fichier sur lequel on écrit,
- *format*, chaîne de caractères qui spécifie ce qui doit être écrit,
- *param1*, expression dont on veut écrire la valeur,
- ...
- *paramN*, expression dont on veut écrire la valeur.
- la valeur retournée est égale au nombre de caractères écrits, ou à une valeur négative si il y a eu une erreur d'entrée-sortie.

La chaîne *format* contient des caractères ordinaires (c'est à dire différents du caractère %) qui doivent être écrits tels quels, et des séquences de spécifications (repérées par le caractère %), décrivant la manière dont doivent être écrits les paramètres *param1, param2, ... paramN*. Attention, s'il y a moins de paramètres que n'en réclame le format, le comportement n'est pas défini. S'il y en a davantage que n'en nécessite le format, les paramètres en excès sont évalués, mais leur valeur est ignorée.

Pour plus de détails, vous pouvez consulter les pages du manuel UNIX (commande `man fprintf`).

**Les spécifications :** elles commencent toujours par le caractère % et se terminent par un caractère de conversion. Entre ces deux caractères, on peut avoir (dans l'ordre) un certain nombre (éventuellement nul) d'indicateurs :
- des drapeaux :

  \- *param* sera cadré à gauche dans son champ d'impression.

  \+ si *param* est un nombre signé, il sera imprimé précédé du signe + ou -.

  *espace* si *param* est un nombre signé et si son premier caractère n'est pas un signe, on imprimera un blanc devant *param*. Si on a à la fois l'indicateur + et l'indicateur espace, ce dernier sera ignoré.

# demande l'impression de *param* sous une forme non standard. Pour le format o, cet indicateur force la précision à être augmentée de manière à ce que *param* soit imprimé en commençant par un zéro. Pour les formats x et X, cet indicateur fait précéder *param* respectivement de 0x ou 0X, sauf si *param* est nul. Pour les formats e, E, f, g et G, cet indicateur force `param` à être imprimé avec un point décimal, même si il n'y a aucun chiffre après. Pour les formats g et G, cet indicateur évite d'enlever les zéros de la fin. Pour les autres formats, le comportement n'est pas défini.

0 pour les formats d, i, o, u, x, X, e, E, f, g, et G cet indicateur a pour effet de compléter l'écriture de `param` avec des 0 en tête, de manière à ce qu'il remplisse tout son champ d'impression. Si il y a à la fois les deux indicateurs 0 et -, l'indicateur 0 sera ignoré. Pour les formats d, i, o, u, x et X, si il y a une indication de précision, l'indicateur 0 est ignoré. Pour les autres formats, le comportement n'est pas défini.

- Un nombre entier décimal (optionnel) indiquant la **taille** minimum du champ d'impression, exprimée en nombre de caractères. Si *param* s'écrit sur un nombre de caractères inférieur à cette taille, il est complété à droite ou à gauche (selon que l'on aura utilisé ou pas l'indicateur -), avec des blancs ou des 0, comme il a été expliqué plus haut.
- Une indication (optionnelle) de **précision**, qui donne :
  - le nombre minimum de chiffres pour les formats d, i, o, u, x et X.
  - le nombre de chiffres après le point décimal pour les formats e, E et f.
  - le nombre maximum de chiffres significatifs pour les formats g et G
  - le nombre maximum de caractères pour le format s.

  Cette indication de précision prend la forme d'un point (.) suivi d'un nombre décimal optionnel. Si ce nombre est omis, la précision est prise égale à 0.
- Un caractère optionnel (h, l ou L) qui modifie la **largeur** du champ.

Les caractères de **conversion** peuvent prendre les valeurs suivantes :

d, i *param* interprété comme un `int`, et écrit sous forme décimale signée.

u *param* interprété comme un `unsigned int`, et écrit sous forme décimale non signée.

o *param* interprété comme un `int`, et écrit sous forme octale non signée.

x, X *param* interprété comme un `int`, et écrit sous forme hexadécimale non signée. La notation hexadécimale utilisera les lettres abcdef dans le cas du format x, et les lettres ABCDEF dans le cas du format X.

Dans les cas qui précèdent, la précision indique le nombre minimum de chiffres avec lesquels écrire *param*. Si *param* s'écrit avec moins de chiffres, il sera complété avec des zéros. La précision par défaut est 1. Une valeur nulle demandée avec une précision nulle, ne sera pas imprimée.

c *param* interprété comme un `unsigned char`.

s *param* interprété comme l'adresse d'un tableau de caractères (terminé ou non par un '
0'). Les caractères du tableau seront imprimés jusqu'à la fin de `param` (rencontre

de ' '
0') ou de précision caractères (dans le cas où *param* n'est pas terminé par un '0', le format d'impression doit comporter une indication de précision).

p *param* interprété comme un pointeur vers `void`. Le pointeur sera imprimé sous une forme dépendante de l'implémentation.

n *param* interprété comme un pointeur vers un `int` auquel sera affecté le nombre de caractères écrits jusqu'alors par cette invocation de `fprintf`.

e,E *param* interprété comme un `double` et écrit sous la forme : [-]m.ddde±xx (partie entière sur un seul chiffre, partie fractionnaire sur le nombre de chiffres donné par la précision (6 par défaut), exposant sur au moins deux chiffres). Dans le cas du format E, la lettre E est imprimée à la place de e.

f *param* interprété comme un `double` et écrit sous la forme : [-]mmm.ddd (partie entière, partie fractionnaire sur le nombre de chiffres donné par la précision (6 par défaut).

**Exemples d'utilisation des formats** (tirés du support de B.Cassagne)

| source C | resultat |
|---|---|
| `printf("\|%d\|\n",1234);` | `\|1234\|` |
| `printf("\|%d\|\n",-1234);` | `\|-1234\|` |
| `printf("\|%+d\|\n",1234);` | `\|+1234\|` |
| `printf("\|%+d\|\n",-1234);` | `\|-1234\|` |
| `printf("\|% d\|\n",1234);` | `\| 1234\|` |
| `printf("\|% d\|\n",-1234);` | `\|-1234\|` |
| `printf("\|%x\|\n",0x56ab);` | `\|56ab\|` |
| `printf("\|%X\|\n",0x56ab);` | `\|56AB\|` |
| `printf("\|%#x\|\n",0x56ab);` | `\|0x56ab\|` |
| `printf("\|%#X\|\n",0x56ab);` | `\|0X56AB\|` |
| `printf("\|%o\|\n",1234);` | `\|2322\|` |
| `printf("\|%#o\|\n",1234);` | `\|02322\|` |
| `printf("\|%10d\|\n",1234);` | `\|      1234\|` |
| `printf("\|%10.6d\|\n",1234);` | `\|    001234\|` |
| `printf("\|%.6d\|\n",1234);` | `\|001234\|` |
| `printf("\|%*.6d\|\n",10,1234);` | `\|    001234\|` |
| `printf("\|%*.*d\|\n",10,6,1234);` | `\|    001234\|` |
| `printf("\|%f\|\n",1.23456789012345678e5);` | `\|123456.789012\|` |
| `printf("\|%.4f\|\n",1.23456789012345678e5);` | `\|123456.7890\|` |
| `printf("\|%.20f\|\n",1.23456789012345678e5);` | `\|123456.78901234567456413060\|` |
| `printf("\|%20.4f\|\n",1.23456789012345678e5);` | `\|         123456.7890\|` |
| `printf("\|%e\|\n",1.23456789012345678e5);` | `\|1.234568e+05\|` |
| `printf("\|%.4e\|\n",1.23456789012345678e5);` | `\|1.2346e+05\|` |
| `printf("\|%.20e\|\n",1.23456789012345678e5);` | `\|1.23456789012345674564e+05\|` |

```
printf("|%20.4e|\n",1.234567890123456789e5);     |          1.2346e+05|

printf("|%.4g|\n",1.23456789012345678e-5);      |1.235e-05|
printf("|%.4g|\n",1.23456789012345678e5);       |1.235e+05|
printf("|%.4g|\n",1.23456789012345678e-3);      |0.001235|
printf("|%.8g|\n",1.23456789012345678e5);       |123456.79|
```

### 7.6.2 Lecture formatée

La fonction `fscanf` admet un nombre variable de paramètres :
`int fscanf` (*flot* ,*format* , *param1* , *param2* , ... ,*paramN* )
avec,

- *flot*, de type pointeur sur `FILE`, le fichier sur lequel on lit,
- *format*, chaîne de caractères qui spécifie la forme de ce qui doit être lu,
- *param1...paramN*, pointeurs sur des variables dans lesquelles `fscanf` range les valeurs lues dans le flot, après les avoir converties en binaire,
- la valeur retournée est égale au nombre de paramètres affectés, si aucun paramètre n'a été affecté (cas de fin de fichier ou d'erreur avant toute affectation), la valeur retournéee est EOF.

La fonction `fscanf` lit une suite de caractères du fichier spécifié par *flot*, en vérifiant que cette suite est conforme à la description donnée dans *format*. Cette vérification s'accompagne d'un effet de bord qui est l'affectation des valeurs aux variables pointées par *param1*,...*paramN*.

Les **caractères séparateurs** sont les suivants : *espace*, *tabulation*, *line feed*, *new line* (\n), *vertical tab* et *form feed*. Notez que la gestion des *espaces* n'est pas triviale.

Un *format* est composé de spécifications, un peu comme pour la fonction `fprintf`. Nous ne décrivons ici que les spécifications de conversion les plus utilisées, pour plus de détails, consultez le support de B.Cassagne, ou encore le manuel UNIX.

| | |
|---|---|
| d | pour un nombre décimal, éventuellement précédé d'un signe, |
| x | pour un nombre hexadécimal, éventuellement précédé d'un signe, |
| c | pour un caractère, |
| s | pour une suite de caractères (non blancs), |
| e,f,g | pour un flottant, |
| lf | pour un double (l est ici un modificateur de type, il en existe 3, h,l et L). |

# Annexe A

# Le préprocesseur

Ce chapitre a été rédigé pour l'essentiel par Touraivane

## A.1 La directive include

On a déjà rencontré la directive `#include`. La directive `#include <stdio.h>` demande au préprocesseur d'inclure le fichier `stdio.h` dans le fichier source avant de le compiler. Ce fichier `stdio.h` contient toutes les déclarations nécessaires pour l'utilisation des fonctions entrée-sorties.

Comme vous l'avez sûrement déjà noté, les fichiers `include` ont traditionellement l'extension `.h` : ce sont les fichiers d'entête ou *header files*. Le fichier `stdio.h` et bien d'autres encore sont livrés avec le compilateur `C`. Vous trouverez généralement tous ces fichiers dans le répertoire `/usr/include`. Ces fichiers sont éditables et il est très vivement déconseillé de les modifier.

En dehors des fichiers fournis avec le compilateur, un programmeur peut créer ses propres fichiers d'entête (on dit aussi interface) pour les programmes qu'il réalise. Supposons que l'on veuille réaliser un programme qui utilise des piles de caractères. On décompose ce programme en deux modules (ou plusieurs), l'un qui implémente les piles de caractères et l'autre qui les uitlise. Le programmeur qui réalise le module d'implémentation des piles founit au(x) programmeur(s) qui réalisent le reste du programme deux fichiers : le premier contenant l'implémentation effective des piles et la deuxième une entête pour ce module.

```
/* Entête du module PILE de caracteres (pile.h) */
typedef char * pile;
pile creer_pile(int);
void empiler(char, pile);
char depiler(pile);
void detruire_pile(pile);
/* Fin de l'entête */
```

Les utilisateurs de ce module ignorent parfaitement les détails d'implémentation de cette pile. En particulier, s'agit-il d'un tableau ou d'une liste chainée ? Seul le

développeur de ce module le sait. L'utilisation de ce module par d'autres se fait alors de la manière suivante :

```
/* Module XXX utilisant une pile de caractères */
#include "pile.h"
void une_fonction(void) {
    short c1, c2;
    pile P;
    P = creer_pile(100);
    empiler('a', P);
    empiler('b', P);
    ...
    c1 = depiler(P);
    if (c1 == EOF) erreur("la pile est vide");
    c2 = depiler(P);
    if (c2 == EOF) erreur("la pile est vide");
    ...
}
/* Fin du module XXX */
```

Le préprocesseur, selon que la directive est de la forme `#include <fichier>` ou de la forme `#include "fichier"` ira chercher ce fichier dans le répertoire `/usr/include` ou dans le répertoire spécifié par l'utilisateur.

## A.2 Les macros et constantes symboliques

En dehors des variables et fonctions qui possèdent un nom symbolique, il est possible d'associer des noms symboliques à des constantes et des portions de code `C`. Ces noms symboliques n'ont d'intérêt que pour le programmeur : ils sont censés rendre le programme plus lisible et plus facile à maintenir.

### A.2.1 Les constantes symboliques

Les constantes symboliques se définissent à l'aide de la directive `#define Nom constante`.

Le `Nom` devient alors un nom symbolique qui peut être utilisé partout où l'on veut faire figurer la constante. Le préprocesseur se contente de substituer syntaxiquement le nom symbolique par la constante avant de compiler.

Par exemple, lorsqu'on définit un tableau de 20 éléments par exemple, on pourrait avoir le code suivant :

```
int tab[20];
main() {
    ...
    for (i=0; i<20; i++)
```

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

On notera que la valeur 20 se trouve éparpillée dans tout le code du programme. Une manière élégante de regrouper cette constante en un seul endroit consiste à définir une constante symbolique `MAX_TAB` et utiliser celle-ci dans tout le programme.

```
#define MAX_TAB 20
int tab[MAX_TAB];
main() {
    ...
    for (i=0; i < MAX_TAB; i++)
        printf("tab[%i]=%d'', tab[i]);
    ...
}
```

Ainsi, si l'on veut modifier la taille du tableau, on se contentera de modifier la seule définition de la constante symbolique `MAX_TAB` et le reste programme reste inchangé.

### A.2.2 Les constantes symboliques sans valeur

On peut également définir des constantes symboliques sans valeur comme `#define UNIX`. Le texte à remplacer étant vide, le préprocesseur se contentera de remplacer toutes les occurences de `UNIX` par la chaîne vide. Quoiqu'étant remplacé par "rien", cette constante sybolique est considérée définie par le préprocesseur. Ce type de constantes est utilisé avec les directives conditionnelles, comme nous le verrons plus loin.

### A.2.3 Supprimer la définition d'une constante symbolique

Il est possible de supprimer la définition d'un nom symbolique par la directive `#undef UNIX`.

### A.2.4 Les macros sans paramètre

Comme les constantes symboliques, les macros se définissent à l'aide de la directive `#define Nom <texte à remplacer>`.

Le `Nom` devient alors un nom symbolique qui peut être utilisé partout où l'on veut faire figurer la texte à remplacer. Le préprocesseur se contente de substituer syntaxiquement le nom symbolique par le texte à remplacer avant de compiler.

```
#define BONJOUR printf("Bonjour, tout le monde !!!");
```

### A.2.5 Les macros avec plusieurs instructions

Puisque la directive `define` ne sert qu'à substituer un nom par le texte défini, rien ne s'oppose à donner un nom symbolique à une suite d'instructions ou toute partie de programme valide.

```
#define ERR_MALLOC  {printf("La fonction malloc a produit une d'erreur"); exit(0); }
```

Comme vous l'avez remarqué, tout le texte à remplacer se trouve sur une même ligne. Et c'est là une restriction (désagréable, j'en conviens) de la directive `define`. Cette restriction étant inappliquable, on contourne celle-ci avec les fameux caractère comme nous l'avions fait pour les chaînes de caractères.

```
#define ERREUR_MALLOC  {                                          \
    printf("La fonction malloc a produit une code d'erreur"); \
    exit(0);                                                  \
}
```

### A.2.6 Les macros avec paramètres

Comme les fonctions, les macros peuvent avoir des paramètres :
`#define Nom(arg1, ..., argn) texte_à_remplacer`
Attention, il ne s'agit pas de fonction, ce ne sont que des macros.

```
#define ERREUR(x)  {                                        \
    printf("La fonction x a produit une code d'erreur"); \
    exit(0);                                               \
}
```

### A.2.7 Concaténation de mots (token)

Il existe un opérateur qui permet de concaténer des mots : il s'agit de ##.

```
#define coller(x, y)  x##y
main() {
    int coller(toto, 5); //équivaut à la déclaration int toto5;
}
```

## A.3 Compilation conditionnelle

Il existe des directives permettant, selon un critère que l'on définira, de compiler de manière conditionnelle un programme. Supposons que l'on veuille réaliser un programme qui doit fonctionner sur un PC, une station SUN ou une machine ALPHA de DIGITAL. Il est vraisemblable que des parties du programme devront être réécrites suivant l'architecture de la machine choisie. Il est tout aussi vraisemblable, que des grands bouts de ce programme (écrit en C) seront communes aux diverses machines.

La compilation conditionnelle permet de réaliser un seul code contenant l'ensemble des codes nécessaires à toutes les machines et en spécifiant au compilateur, selon la machine choisie, quelle partie du code il faudra utiliser.

Pour ce faire, on dispose des directives

```
#if
#elif
#else
#endif
#ifdef
#indef
```

Voici des bouts de programmes trouvés dans le programme `xcoral` :

```
...
#ifdef SYSV
#include
extern void bzero();
#else
#include
#endif
...
#ifdef  USE_NANOSLEEP
  struct timespec rqtp;
  ...
#elif   USE_POLL
  struct pollfd unused;
  ...
#else  /* default to select */
  struct timeval delay;
  ...
#endif /* USE_NANOSLEEP */
```

Contrairement à la directive `#ifdef` qui est toujours associée à une seule constante symbolique, la directive `#if` peut être combinée avec une expression à condition qu'elle soit constante entière.

Voici quelques exemples trouvés dans les sources de Linux.

```
#if EVERY_ACCESS || ERRORS_ONLY || DEBUG_ABORT
   ...
#endif
#if ((~0UL) == 0xffffffff)
   ...
#else
   ...
#endif
```

```
#if (((DEBUG & PHASE_ETC) == PHASE_ETC) || (DEBUG & PRINT_COMMAND) )
   ...
#enif
```

### A.3.1 L'opérateur `defined`

La directive `#if` peut être combinée avec l'opérateur `defined`. Le code

```
#if defined(CONSTANTE)
    ...
#endif
```

est équivalent au code

```
#ifdef (CONSTANTE)
    ...
#endif
```

L'intérêt de cet opérateur que l'utilisation de la directive `#if` (au lieu de `#ifdef`) permet d'utiliser des expressions complexes.

```
#if defined(MODULE) && !defined(GFP_DMA)
    ...
#endif
```

### A.3.2 Numérotation des lignes

La directive `#line` initialise des lignes du fichier source. C'est ainsi que le compilateur peut afficher les numéros de ligne d'une erreur ; n'oublions pas que le fichier passé au compilateur est différent du fichier source de l'utilisateur à cause de la phase du préprocesseur. C'est d'ailleurs ce préprocesseur qui ajoute automatiquement les directives `#line` pour que le compilateur puisse afficher correctement les messages d'erreurs.

```
#line n "fichier"
```

# Annexe B

# Notes sur la compilation séparée

Ce document est probablement imparfait, mais il devrait déjà donner une introduction raisonnable au sujet traité...

## B.1 Introduction

La compilation séparée est indispensable pour qui veut développer des applications importantes (en taille !). Elle suppose un découpage de l'application en modules compilables séparément. On ne discutera pas ici des avantages de la programmation modulaire (clarté, robustesse, maintenance, réutilisabilité,...). Ce qui suit est valable en C, C++, nous utilisons ici le compilateur GNU `gcc`.

Un **module** est composé d'un ensemble de procédures et fonctions regroupées dans un fichier avec les déclarations nécessaires et sans programme principal (`main`). Pour chaque module on écrira deux fichiers :

- le fichier *source* (suffixe **.c**) qui contient les définitions,
- le fichier *en-tête* (suffixe **.h**) qui contient les déclarations des variables et fonctions exportées, c'est-à-dire susceptibles d'être utilisées par d'autres modules.

Un module qui utilise des entités (fonctions ou variables) d'un autre module doit "inclure" le fichier en-tête correspondant (cf. directive #include).

Les modules peuvent être compilés séparément, et l'exécutalbe est obtenu par l'édition de liens qui assemble tous les fichiers objets.

## B.2 Produire un exécutable

Il faut d'abord bien comprendre qu'il y a plusieurs étapes à franchir pour produire un exécutable. Pour suivre les étapes de la compilation, on prendra pour exemple le petit programme C ci-dessous (`bonjour.c`) :

```
#include<stdio.h>
int i=2;
main(){
 printf("Bonjour !\n");
```

```
}
```

Voici les étapes enchaînées par le compilateur gcc (elles sont détaillées plus loin) :

| **Avant** (extension) | **Traitement** | **Après** (extension) |
|---|---|---|
| Source C **.c** | `cpp` *C PreProcessor* | Source C *pur* **.i** |
| Source C *pur* **.i** | `pcc` *Pur C Compiler* | Source assembleur **.s** |
| Source assembleur **.s** | `as` *ASsembler* | Fichier objet **.o** |
| Fichiers objets et bibliothèques **.o** et**.a** | `ld` *LoaDer* éditeur de liens | exécutable |

Pour mémoire un tableau (non exhaustif) des extensions de fichiers que vous pourrez rencontrer, et leur signification :

| | |
|---|---|
| **.c** | source C |
| **.cxx** | source C++ |
| **.cc** | source C++ |
| **.h** | fichier d'entête |
| **.i** | source C prtraité |
| **.ii** | prétraié C++ |
| **.s** | source assembleur |
| **.o** | fichier objet |

### B.2.1 Le préprocesseur

Le préprocesseur (`cpp`) opère un pré-traitement, qui se résume essentiellement à du traitement de texte, en mettant en oeuvre les directives comme `#include, #define, ...` (inclusion d'entêtes, développement de macros, compilation conditionnelle). L'option **-E** du compilateur `gcc` indique au compilateur de s'arrêter après la phase de pré-traitement qui se contente d'exécuter toutes les directives qui s'adressent au préprocesseur, le résultat est envoyé sur la sortie standard (ou, si vous le spécifiez, un fichier avec traditionnellement une extension **.i**) Attention, il y est assez volumineux !

### B.2.2 Le compilateur "pur"

Après la phase de pré-traitement, le compilateur passe à une phase de vérification syntaxique et traduit le source C en langage assembleur (`pcc`), qui dépend de la plateforme sur laquelle vous travaillez. On peut demander au compilateur de s'arrêter à la fin de cette étape, en utilisant l'option **-S**, on a alors un fichier source en assembleur (extension **.s**). Par exemple, la commande `cc -S bonjour.c` génère le fichier `bonjour.s` suivant (sur une machine `Debian`) :

```
.file "bonjour.c"
.version "01.01"
gcc2_compiled.:
.globl i
.data
.align 4
```

```
.type  i,@object
.size  i,4
i:
.long 2
.section .rodata
.LC0:
.string "Bonjour !\n"
.text
.align 16
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
pushl $.LC0
call printf
addl $4,%esp
.L1:
movl %ebp,%esp
popl %ebp
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.7.2.3"
```

### B.2.3 L'assembleur

L'assembleur (`as`) transforme le fichier source assembleur en un fichier binaire dit "fichier objet" (code en langage machine, non exécutable car incomplet), il s'agit d'un fichier avec l'extension **.o**. Ce ficher contient des instructions et des données en langage machine, et chaque donné ou chacune d'entre elles a un nom symbolique par lequel elle est rérencée (on parle de lien à résoudre quand on fait référence à un objet défini à l'extérieur du module et lien utilisable quand il s'agit d'un objet pouvant être référencé dans un module différent de celui dans lequel il est défini. La commande `cc -c bonjour.c` génère le fichier objet `bonjour.o`. Ce n'est pas un fichier texte. Pour le visualiser, vous pouvez utiliser le programme `od` (pour *Octal Dump*). Voici l'affiachage obtenu suite à la commande `od -c bonjour.o` :

```
0000000 177   E   L   F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 001  \0 003  \0 001  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040 330  \0  \0  \0  \0  \0  \0  \0   4  \0  \0  \0  \0  \0   (  \0
0000060  \v  \0  \b  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100   U 211 345   h  \0  \0  \0  \0 350 374 377 377 377 203 304 004
0000120 211 354   ] 303 002  \0  \0  \0  \b  \0  \0  \0  \0  \0  \0  \0
0000140 001  \0  \0  \0   0   1   .   0   1  \0  \0  \0   B   o   n   j
0000160   o   u   r       !  \n  \0  \0   G   C   C   :       (   G   N
0000200   U   )       2   .   7   .   2   .   3  \0  \0   .   s   y   m
0000220   t   a   b  \0   .   s   t   r   t   a   b  \0   .   s   h   s
0000240   t   r   t   a   b  \0   .   t   e   x   t  \0   .   r   e   l
0000260   .   t   e   x   t  \0   .   d   a   t   a  \0   .   b   s   s
0000300  \0   .   n   o   t   e  \0   .   r   o   d   a   t   a  \0   .
0000320   c   o   m   m   e   n   t  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

```
0000340  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000400 033  \0  \0  \0 001  \0  \0  \0 006  \0  \0  \0  \0  \0  \0  \0
0000420   @  \0  \0  \0 024  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000440 020  \0  \0  \0  \0  \0  \0  \0   !  \0  \0  \0  \t  \0  \0  \0
0000460  \0  \0  \0  \0  \0  \0  \0  \0   x 003  \0  \0 020  \0  \0  \0
0000500  \t  \0  \0  \0 001  \0  \0  \0 004  \0  \0  \0  \b  \0  \0  \0
0000520   +  \0  \0  \0 001  \0  \0  \0 003  \0  \0  \0  \0  \0  \0  \0
0000540   T  \0  \0  \0 004  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000560 004  \0  \0  \0  \0  \0  \0  \0   1  \0  \0  \0  \b  \0  \0  \0
0000600 003  \0  \0  \0  \0  \0  \0  \0   X  \0  \0  \0  \0  \0  \0  \0
0000620  \0  \0  \0  \0  \0  \0  \0  \0 004  \0  \0  \0  \0  \0  \0  \0
0000640   6  \0  \0  \0  \a  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000660   X  \0  \0  \0 024  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000700 001  \0  \0  \0  \0  \0  \0  \0   <  \0  \0  \0 001  \0  \0  \0
0000720 002  \0  \0  \0  \0  \0  \0  \0   l  \0  \0  \0  \v  \0  \0  \0
0000740  \0  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0
0000760   D  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0001000   w  \0  \0  \0 024  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0001020 001  \0  \0  \0  \0  \0  \0  \0 021  \0  \0  \0 003  \0  \0  \0
0001040  \0  \0  \0  \0  \0  \0  \0  \0 213  \0  \0  \0   M  \0  \0  \0
0001060  \0  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0
0001100 001  \0  \0  \0 002  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0001120 220 002  \0  \0 300  \0  \0  \0  \n  \0  \0  \0  \t  \0  \0  \0
0001140 004  \0  \0  \0 020  \0  \0  \0  \t  \0  \0  \0 003  \0  \0  \0
0001160  \0  \0  \0  \0  \0  \0  \0  \0   P 003  \0  \0   (  \0  \0  \0
0001200  \0  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0
0001220  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0001240 001  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 004  \0 361 377
0001260  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 001  \0
0001300  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 003  \0
0001320  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 004  \0
0001340  \v  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 001  \0
0001360  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 006  \0
0001400  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0 005  \0
0001420  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 003  \0  \a  \0
0001440 032  \0  \0  \0  \0  \0  \0  \0 004  \0  \0  \0 021  \0 003  \0
0001460 034  \0  \0  \0  \0  \0  \0  \0 024  \0  \0  \0 022  \0 001  \0
0001500   !  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 020  \0  \0  \0
0001520  \0   b   o   n   j   o   u   r   .   c  \0   g   c   c   2   _
0001540   c   o   m   p   i   l   e   d   .  \0   i  \0   m   a   i   n
0001560  \0   p   r   i   n   t   f  \0 004  \0  \0  \0 001 006  \0  \0
0001600  \t  \0  \0  \0 002  \v  \0  \0
0001610
```

Un fichier objet contient la suite d'octets qui représente le programme, mais aussi une table des étiquettes (ou références) définies ou requises par le programme. On peut visualiser cette table des symboles en utilisant le programme `nm`. Voici l'affichage obtenu suite à la commande `nm` :

```
00000000 D i
00000000 T main
```

```
U printf
```

La deuxième colonne affichée par `nm` donne le type du symbole et la dernière colonne donne le nom du symbole (pour plus de détails, consulter le manuel de `nm`.

- **T** ou **t** pour une définition globale,
- **U** pour une référence externe,
- **d** pour une définition de donnée locale.

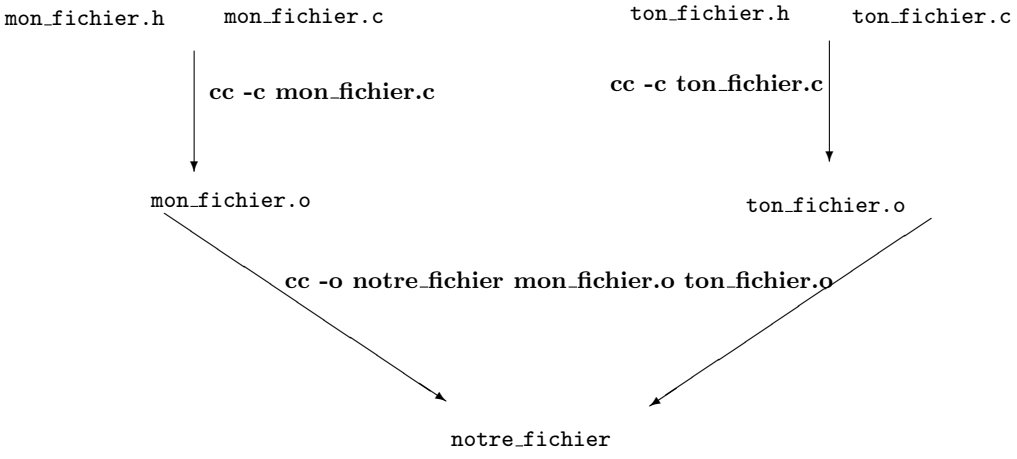
Ainsi, un fichier objet contient ce que l'on appelle des liens à résoudre (références externes à des objets non définis dans le module) et des liens utilisables, c'est-à-dire pouvant être référencé dans un autre module que celui-ci.

## B.3 L'éditeur de lien (ce paragraphe est à completer...)

L'éditeur de lien (`ld`), assemble les codes objets, et résoud les références (c'est-à-dire affecte aux liens à résoudre des liens utilisables). C'est lui qui produit le fichier exécutable.

## B.4 Exemple

Dans l'exemple qui suit (que l'on suppose être en C), on a deux modules (`mon_fichier` et `ton_fichier`) et un programme principal qui utilise ces deux modules (`notre_fichier`). Voici la chaîne suivie pour produire l'exécutable :




Cette chaîne illustre ce que l'on appelle les dépendances entre les fichiers et que l'on peut obtenir par la commande `gcc -MM *.c` qui vous retournera toutes les dépendances entre les fichiers d'extension **.c** du répertoire courant.

Vous devriez vous demander à ce stade s'il n'y a pas moyen d'automatiser tout cela. En effet, si le fichier `ton_fichier.h` change, alors il faut recompiler `ton_fichier.c` et `notre_fichier.h` mais pas `mon_fichier.c`. Imaginez qu'au lieu de 5 fichiers, nous en ayons 50 ! Il existe un utilitaire pour cela, il s'agit de **make**.

## B.5 L'utilitaire Make

L'utilitaire `make` sert à coordonner la compilation et l'édition de liens d'un ensemble de sources qui composent une application. L'objectif et de ne recompiler que ce qui doit l'être. Pour cela `make` gère les dépendances entre fichiers qui sont décrites dans un fichier sp{ecifique appelé `makefile`. Typiquement, les fichiers *objet* dépendent des fichiers *source*, ainsi, lorsqu'un *source* est modifié, le fichier *objet* correspondant doit être reconstruit. L'utilitaire `make` doit disposer de la description des dépendances et des règles de création de s fichiers cibles (tout cela est décrit dans le fichier `makefile`) mais il doit également disposer des fichiers eux-mêmes (pour vérifier leur date de dernière modification).

Dans le fichier `makefile`, on écrit trois type d'instructions :

- description des dépendances : `nom_fichier_cible : nom(s)_source(s)`
- règle de production : `<tab> liste_de_commandes`
- définition de *macros* : pour faciliter l'écriture des dépendances et des règles de production, on peut définir ce que l'on appelle des *macros*. Il s'agit de donner un nom à une chaî de caractères, c'est ensuite simplement du remplacement de texte. Par exemple, si un fichier dépend des fichiers `fic1.o`, `fic2.o`,`fic3.o`,`fic4.o`, on écrira `OBJ = fic1.o fic2.o fic3.o fic4.o` pour définir la macro, puis on écrira `$(OBJ)` pour y faire référence.
  Il existe des macros spéciales :
  - `$@` est le nom de la cible à reconstruire
  - `$*` est le nom de la cible sans suffixe
  - `$<` est le nom de la dépendance à partir de laquelle on reconstruit la cible
  - `$?` est la liste des fichiers dont dépend la cible et qui sont plus récents que la cible.
- enfin, pour les commentaires dans le `makefile`, utilisez le `#`.

### B.5.1 Petits exemples de *Makefile*

1. ```
   notre_fichier : mon_fichier.o ton_fichier.o
         cc mon_fichier.o ton_fichier.o -o notre_fichier
   mon_fichier.o: mon_fichier.c mon_fichier.h
         cc -c mon_fichier.c
   ton_fichier.o: ton_fichier.c ton_fichier.h
         cc -c ton_fichier.c
   nettoie:
         rm -rf *.o notre_fichier
   ```
2. ```
   OBJ = fic1.o fic2.o fic3.o
   programme : $(OBJ)
   $(OBJ) -o $@
   fic1.o : fic1.c;
   cc -c $*.c
   ```