— LozZone —

Introduction au langage C

Version du 8 septembre 2008
Dernière version sur :
http://lozzone.free.fr

Vincent LOZANO

Sommaire

1 Introduction

1.1 Ce manuel

La rédaction de ce document a débuté pendant l'année scolaire 2002-2003 à destination des auditeurs du Cnam de Saint-Étienne de la filière informatique. Il a pour but de livrer une introduction au langage C ; l'auteur a choisi de faire découvrir le langage en considérant que les notions élémentaires d'algorithmique sont connues et en tentant de présenter les concepts par ordre de difficulté. Une lecture linéaire de ce document s'impose donc, même si le lecteur pourra revenir dans un deuxième temps sur des parties isolées. Cet ouvrage n'est pas une présentation de la librairie standard du langage C, et ne détaille pas l'utilisation de tout le langage. L'accent est cependant mis sur les mécanismes particuliers du langage C et ceci par le biais d'exemples et d'illustrations.

1.2 Historique

L'histoire du langage C remonte au début des années soixante dix, où Dennis Ritchie rejoint Ken Thompson pour développer le système Unix. Le C a pour but de créer un langage permettant de faire évoluer Unix en conservant sa portabilité. Mais au delà du développement du système, Kernighan et Ritchie notent dans la préface de la première édition du livre « Le langage C » (1978) que ce langage, ni de bas niveau, ni de très haut niveau, est un langage adapté à des projets très variés. Ils citent en exemple le système Unix lui-même, le compilateur C, la plupart des utilitaires système et le programme qui a permis de mettre en œuvre leur livre.

Le C a été normalisé en 1988 par l'Ansi. [1] L'apport majeur de cette norme a été de standardiser une bibliothèque de fonctions, et ainsi d'assurer la portabilité du langage sur de nombreux systèmes.

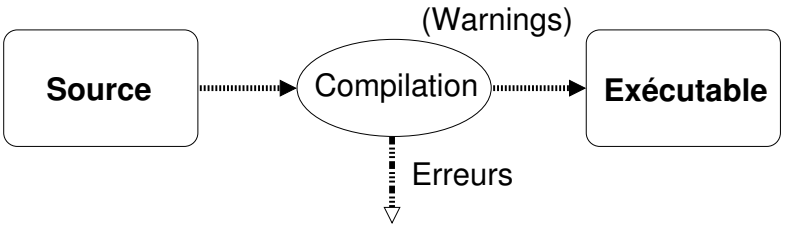
Notons enfin que si aujourd'hui le C est considéré plutôt comme un langage de bas niveau par rapport à d'autres langages actuels, il est malgré tout encore très utilisé. Il présente par ailleurs un atout majeur sur le plan de l'apprentissage car son aspect bas niveau permet (et impose) de comprendre et de maîtriser les mécanismes d'accès à la mémoire et de communication avec le système d'exploitation.

1.3 C : un langage compilé

Pour créer un exécutable à partir d'un programme écrit en C, il faudra passer nécessairement par les étapes suivantes :

❶ édition du *programme source* à l'aide d'un éditeur de texte. Cette phase créera donc un fichier de type texte, c'est-à-dire ne contenant que les codes Ascii [2] des caractères qui le composent ;

❷ compilation [3] à l'aide d'un programme particulier appelé *compilateur*. Ce programme après avoir vérifié la syntaxe, la grammaire et la sémantique du source, va créer un programme qui pourra être exécuté par le système d'exploitation sur un processeur donné.



La compilation peut générer des erreurs au cours des analyses effectuées lors de la deuxième phase, dans ce cas aucun exécutable ne sera créé. La particularité du langage C est que le compilateur émet également des *warnings* ou avertissements. Ces avertissements n'empêchent pas la création de l'exécutable mais attirent l'attention du programmeur sur certaines parties du code source qui peuvent générer des erreurs à l'exécution. Il est donc toujours préférable de supprimer tous les avertissements en corrigeant le code source, à moins d'être sûr de ce que l'on fait. Le lecteur se reportera à la section 10 page 45 où il trouvera des informations précises sur la production d'un exécutable à partir d'un ou plusieurs sources écrits en C.

1.4 Bonjour, monde...

Il est courant de découvrir un langage de programmation en étudiant de manière très sommaire le célèbre programme « hello, world » pour se faire un première idée de la syntaxe du langage :

1. American national standard institute.
2. American standard code for information interchange.
3. On verra plus loin que ce terme n'est pas utilisé ici tout à fait à bon escient, puisqu'il faudrait dire « compilation et édition de liens ». Lire à ce sujet le paragraphe 10 page 45.

```
1 /* premier programme en C */
2 #include <stdio.h>
3
4 int main()
5 {
6   printf("Bonjour, monde...\n");
7   return 0;
8 }
```

La première ligne du source est un commentaire, c'est-à-dire du texte qui sera ignoré par le compilateur. Les commentaires peuvent s'étendre sur plusieurs lignes et commencent par `/*` et finissent par `*/`. On notera également que ces commentaires ne peuvent être imbriqués.

Sans entrer dans les détails pour l'instant, [4] la ligne 2 indique que l'on va inclure des fonctions de la bibliothèque standard d'entrée/sortie (*standard input/output library*).

La partie du source **int** `main`(){...} est une *fonction* particulière du source, appelée fonction *main* ou programme principal. Lorsqu'on exécutera le programme, on exécutera les instructions de la fonction *main*.

Les lignes 5 et 8 définissent un *bloc* d'instructions qu'on peut rapprocher du *begin* et du *end* du Pascal.

À la ligne 6 est appelée la fonction `printf` qui permet d'afficher à l'écran la chaîne de caractères « bonjour, monde... » ; on notera l'utilisation de la séquence `\n` pour ajouter un saut de ligne après l'affichage. La bibliothèque d'entrée sortie sera étudié au paragraphe 9 page 36.

Le mot clef **return** de la fonction `main` permet de faire *renvoyer* une valeur à la fonction, et donc dans ce cas précis au programme lui-même. Dans ce cas le système d'exploitation et plus précisément l'interpréteur de commandes pourra utiliser ce code le cas échéant.

On notera enfin que la ligne 6 constitue une *instruction* et que les instructions du langage C sont terminées par le caractère '`;`'.

2 Variables et types

Le langage C est un langage faiblement typé, c'est-à-dire que le compilateur ne rechignera généralement pas à compiler des instructions affectant un flottant à un entier et inversement. Il vous *avertira* lorsqu'il rencontrera des affectations de pointeurs à des entiers, ou de pointeurs de types différents.

2.1 Types prédéfinis

Les types reconnus par le langage C sont les suivants :

Les entiers : signés ou non signés :

4. D'autres informations à ce sujet au paragraphe 10 page 45

types	taille	portée
short	2 octets	$[-32768, 32767]$
unsigned short		$[0, 65535]$
int	4 octets	$[-2147483648, 2147483647]$
unsigned int		$[0, 4294967295]$
long	4 octets	$[2147483648, 2147483647]$
unsigned long		$[0, 4294967295]$

sur les machines 32 bits d'aujourd'hui il n'y a pas de différence entre le type **long** et le type **int**. [5]

Les flottants : permettent de manipuler des nombres à virgule flottante ayant une précision définie :

Type	taille	précision
float	4 octets	environ 10^{-7}
double	8 octets	environ 10^{-15}

Caractère : le type **char** permet de mémoriser un caractère. Le langage C ne fait pas de distinction entre le caractère lui-même (par exemple 'A') et son code Ascii. On peut donc se représenter le type **char** comme un entier codé sur un caractère :

types	taille	portée
char	1 octet	$[-128, 127]$
unsigned char		$[0, 255]$

Il faut noter qu'il n'y a pas, en C, de type *chaîne de caractères*, même si le langage permet de le construire à partir d'un tableau de caractères (cf. § 8 page 32) ;

Le type void : qui permet :

❶ de ne faire renvoyer aucune valeur à une fonction (cf. § 5 page 14) ;

❷ de définir des pointeurs non typés (cf. § 5.1 page 14).

Le type pointeur : ce type permet de stocker une adresse mémoire dans une variable (cf. § 5.1 page 14).

On pourra noter que le langage C :

➟ propose deux types structurés que nous présenterons au paragraphe 6 : le tableau et l'enregistrement ;

➟ ne propose pas de type *booléen* (vrai/faux) ;

➟ la fonction **sizeof** renvoie la taille en octets occupée par un type. Ainsi, sur une système 32 bits, **sizeof**(**int**) renverra la valeur 4.

2.2 Variables

Les variables permettent de réserver un espace mémoire et d'affecter un symbole à cette zone. Lequel symbole représente le *nom* ou son *identificateur*. D'autre part, une variable est toujours d'un *type* particulier, prédéfini dans le langage C ou défini par le programmeur.

5. La taille de type **int** n'est pas fixée par la norme du langage C. Elle peut être de 2 ou 4 octets en fonction de l'architecture. Même si aujourd'hui ce sera toujours 4 octets en pratique, on ne présumera pas de la taille du type **int**.

2.2.1 Déclaration

Pour déclarer une variable on procédera comme suit :

⟨*type*⟩ ⟨*nom de la variable*⟩ ;

Par exemple :

```
1 int entier;
2 double x,y,z;
3 char car1, car2;
```

2.2.2 Affectation

On peut ensuite affecter des valeurs à ces variables, comme suit :

```
1 entier=4;
2 x=y=2.7;
3 z=1.2345e16;
4 car1='a';/* car1 reçoit le code de 'a' */
5 car2=90; /* on affecte ici le code du caractère */
```

En C, il est possible de procéder à la déclaration et à l'initialisation en une seule instruction, par exemple :

```
int entier=4;
double x=2.7,y=2.7,z=1.2345e16;
char car1='a',car2=90;
```

2.2.3 Constantes

On notera par le biais de ces exemples qu'il est possible de manipuler des *constantes* de type entier ou de type flottant :

constante entière : peuvent être de la forme :

- 12 ou -90 ou ...
- 023 correspondant à 23_8 en base 8 (soit 19 en base 10) ;
- 0x40 correspondant à 40_{16} en base 16 (soit 64 en base 10) ;

constante caractère : peuvent être de la forme :

- le code Ascii du caractère ;
- 'Z', 'a' : le caractère lui-même entre apostrophe.

constante flottante : peuvent être exprimées sous la forme :

- -1.4 ou 2.345 ou .434 ...
- 23.4e-10 ou 1.234e45 ou ...

2.2.4 Typage

Le C est un langage *faiblement typé* ; c'est-à-dire que le compilateur n'émet pas d'erreur et très peu d'avertissement lorsqu'on affecte à une variable d'un type donné une valeur d'un autre type. Ainsi, par exemple :

```
int i=80000;
char c;
double d=4.76;

c=i;
printf("le caractère : %d\n",c);
i=d;
printf("l'entier     : %d\n",i);
```

Donnera :

```
le caractère : -128
l'entier     : 4
```

La première affectation, d'un entier à un caractère, affecte le premier octet du codage de 800000 ($100111000\boxed{10000000}_2$) au caractère, on retrouve donc la valeur -128. Dans la deuxième, l'entier se voit affecter la valeur de 4.76 tronquée, soit 4. Notez bien que dans ces deux cas le compilateur ne vous avertira pas.

2.2.5 Portée et durée de vie

On appelle *portée* d'une variable les zones du texte du source dans lesquelles on peut y faire référence. Accéder à une variable en dehors de sa portée générera une erreur à la compilation.

On appelle *durée de vie* d'une variable le temps pendant lequel la zone mémoire associée est effectivement allouée. Cette durée ne peut dépasser la durée d'exécution du programme.

Variables globales Toutes les variables déclarées en dehors du bloc de la procédure `main` et de tout autre bloc, sont des variables *globales* :

- leur durée de vie est le temps d'exécution du programme ;
- leur portée est l'intégralité du source *après* la déclaration ;

```
int Globale=20;

int main()
{
  Globale=10;
}
```

Ici la variable `Globale` peut être utilisée dans tout le source après sa déclaration à la ligne 1.

Variables locales Les variables déclarées localement à un bloc (par exemple le bloc de la fonction `main`) sont des variables *locales* :

- leur durée de vie est le temps d'exécution du bloc ;
- leur portée est le bloc lui-même et les blocs inclus.

```
int main()
{
  int locale;
  if (Globale=10)
    {
      int llocale=2;
    }
}
```

Ici, la variable `locale` n'est visible que dans la procédure `main`, la variable `llocale` n'est visible que dans le bloc du **if**.[6]

2.3 Affichage du contenu des variables

On utilise la fonction `printf` pour afficher le contenu des variables. La syntaxe de cette fonction est la suivante :

`printf(`⟨*chaine de format*⟩, ⟨var_1⟩, ⟨var_2⟩,...`);`

La chaîne de formattage permet de spécifier la manière d'afficher. Les arguments suivants sont les variables que l'on veut afficher. Ainsi :

```
int i=20;
double d=0.176471;

printf("la valeur de i est : %d\n",i);
printf("d vaut %f ou encore (%1.2f)\n",d,d);
printf("i=%04d et d=%1.3f\n",i,d);
```

Ce programme donne à l'écran :

```
la valeur de i est : 20
d vaut 0.176471 ou encore (0.18)
i=0020 et d=0.176
```

On notera donc que le caractère `%` permet de spécifier le contenu d'une variable (variable qui doit apparaître dans la liste des arguments qui suivent), et que :

- `%d` permet d'afficher un entier ;
- `%04d` permet d'afficher un entier sur 4 caractères en comblant avec des 0 sur la gauche si nécessaire ;
- `%f` permet d'afficher un nombre à virgule flottante ;
- `%1.3f` un nombre à virgule flottante avec 1 chiffre avant la virgule et 3 après ;

La bibliothèque d'entrée/sortie du C dispose de bien d'autres fonctionnalités dont nous vous faisons grâce ici...

2.4 Allocation mémoire

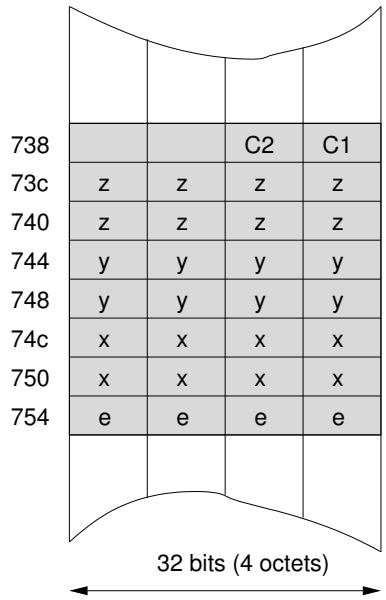
Il est possible d'obtenir l'adresse des zones mémoires des variables que l'on déclare, en utilisant l'opérateur & :

6. Voir § 4 page 11 sur les structures de contrôle.

```
int entier;
double x,y,z;
char car1, car2;

printf("adresse de entier\t: %p\n", &entier);
printf("adresse de x\t\t: %p\n", &x);
printf("adresse de y\t\t: %p\n", &y);
printf("adresse de z\t\t: %p\n", &z);
printf("adresse de car1\t\t: %p\n", &car1);
printf("adresse de car2\t\t: %p\n", &car2);
```

Ce programme permet d'afficher l'adresse des variables déclarées. [7] À l'écran, sur un système Unix, on a :

```
adresse de entier       : 0xbffff754
adresse de x            : 0xbffff74c
adresse de y            : 0xbffff744
adresse de z            : 0xbffff73c
adresse de car1         : 0xbffff73b
adresse de car2         : 0xbffff73a
```



Le fait que les variables soient stockées en mémoire dans l'ordre inverse où elles sont déclarées illustre le fait que les variables locales sont stockées dans une zone mémoire particulière appelée la *pile*. Il s'agit d'une zone dans laquelle on « empile » les données à partir d'une adresse donnée.

2.5 Opérations arithmétiques

On peut utiliser naturellement les opérateurs $+$, $-$, $*$ et $/$ pour l'addition, la soustraction, la multiplication et la division.

Il est parfois utile d'utiliser l'opérateur % qui renvoie le modulo ou le reste de la division entière, par exemple 5%3 renvoie 2.

2.5.1 Conversion des types

Il est important de comprendre que lors de l'*évaluation* d'une opération arithmétique, l'opération sera effectuée dans le plus grand des deux types mis en jeu, ainsi :

7. On notera le format `%p` qui permet d'afficher une adresse en hexadécimal et le caractère `\t` pour insérer une tabulation.

```
int i=2;
float f=4.3;

f=f*i;
```

la multiplication `f*i` est effectuée en **float**. **Attention,** l'opérateur de division peut agir comme division entière ou division sur les nombres à virgule flottante. À ce sujet :

```
float f=2/3;
```

affecte la valeur 0 à `f` car puisque 2 et 3 sont des constantes de types entiers, la division est une division entière ! On écrira donc :

```
float f=2.0/3;
```

pour affecter $2/3 = 0.6666...$ à `f`.

2.5.2 Bibliothèque mathématique

La bibliothèque mathématique propose un grand nombre de fonctions mathématiques (logarithmes, fonctions puissance et racines, trigonométrie, arrondis, ...). Pour les utiliser dans un programme, on écrira dans l'entête :

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

et selon l'environnement de développement il pourra être nécessaire d'inclure la bibliothèque à l'édition de liens (cf. § 10 page 45). Les fonctions disponibles sont, en vrac et de manière non exhaustive :

- **double** `exp`(**double** `x`); renvoie l'exponentielle de `x` ;
- **double** `log`(**double** `x`); renvoie le logarithme népérien de `x` ;
- **double** `pow`(**double** `x`, **double** `y`); renvoie `x` à la puissance `y` ;
- **double** `sqrt`(**double** `x`); **double** `cbrt`(**double** `x`); renvoient la racine carrée et cubique de `x`, respectivement ;
- **double** `sin`(**double** `x`); **double** `cos`(**double** `x`); **double** `tan`(**double** `x`); renvoient respectivement le sinus, le cosinus et la tangente de `x` (`x` étant exprimé en radians) ;
- **int** `abs`(**int** `x`); **double** `fabs`(**double** `x`); renvoie la valeur absolue de `x` ;
- **double** `ceil`(**double** `x`); **double** `floor`(**double** `x`); **double** `rint`(**double** `x`); effectuent toutes trois des arrondis, `ceil` par excès, `floor` par défaut et `rint` vers l'entier le plus proche.

3 Instructions

On notera que :

- les instructions du langage C doivent être terminées par un point virgule : ';'
- elles peuvent être regroupées dans un bloc {...}

3.1 Valeurs renvoyées par les instructions

Il est important de comprendre que toutes les instructions renvoient une valeur sauf s'il s'agit d'une procédure de type **void**. Par exemple, la fonction `printf`, qu'on utilise généralement comme une procédure au sens du Pascal, renvoie une valeur qui est le nombre de caractères affichés :

```
int nbcar;

nbcar=printf("bonjour\n");
printf("nombre de caractères affichés : %d\n",nbcar);
```

affichera :

```
bonjour
nombre de caractères affichés : 8
```

Il faut également noter qu'une opération d'affectation renvoie une valeur : la valeur affectée. On verra que ceci peut d'une part être exploité dans les tests des structures de contrôle, et que d'autre part cela permet l'écriture :

```
int x,y;
x=y=4;
```

l'expression `y=4` renvoie la valeur 4 qui est affectée à `x`.

3.2 Instructions d'incrémentation

Le C permet d'écrire de manière concise les instructions d'incrémentation, ainsi les instructions :

```
x=x+4;
y=y-5;
```

peuvent être écrites :

```
x+=4;
y-=5;
```

dans les deux cas x est incrémenté de 4, et y décrémenté de 5. D'autres opérateurs analogues existent comme $*=$, $/=$, etc. En fait, tous les opérateurs binaires (prenant deux opérandes) que nous rencontrerons peuvent se décliner sous cette forme.

Il existe un opérateur particulier : l'opérateur `++` qui ajoute la valeur 1 à son opérande. Il peut s'utiliser comme opérateur de « post-incrémentation » :

```
int a,b=4;

a=b++;
```

Dans ce cas, `b++` renvoie la valeur de `b` (4) puis incrémente `b`. Donc ici `a` recevra la valeur 4. Dans le cas de la « pré-incrémentation » :

```
int a,b=4;

a=++b;
```

Dans ce cas, ++b incrémente b puis renvoie la valeur (5) ; donc ici a vaudra 5. Il faut noter que dans les deux cas, pré ou post-incrémentation, b vaut 5 après l'opération. On peut naturellement utiliser ces opérateurs sans exploiter la valeur renvoyée :

```
int a,b=4;

b++; /* ou ++b; */
a=b;
```

4 Structures de contrôle

Les structures de contrôle du langage C permettent d'utiliser des constructions du type Si...Alors...Sinon, Faire...Tant que, etc. Ces structures *itératives* et *conditionnelles* reposent sur des tests et donc sur l'*évaluation d'expressions booléennes.* Il faut également noter que toutes les structures de contrôle présentées ci-après peuvent être imbriquées les unes dans les autres.

4.1 Expressions booléennes

Il n'y a pas de type booléen en langage C (contrairement au Pascal, et au C++ par exemple). En C :

❶ la valeur 0 équivaut à *faux* ;

❷ la valeur non-zéro équivaut à *vrai*

Les opérateurs « classiques » pour effectuer des tests sont :

< (<=)	inférieur (ou égal)
> (>=)	supérieur (ou égal)
==	égal
!=	différent
||	ou logique
&&	et logique
!	négation

Avant de commencer l'étude des structures de contrôle on pourra constater que :

```
int a=4,b=5;
printf("%d\n",a>b);
```

affiche 0 à l'écran...

4.2 Structures alternatives

Il s'agit des structures de contrôle permettant d'exécuter des instructions en fonction de choix définis ; ces structures ne sont pas itératives, c.-à-d. qu'elles ne permettent pas de répéter des instructions.

4.2.1 Alternative simple : Si ... Alors ... Sinon

La syntaxe pour construire une structure de ce type est la suivante :

```
if ( <expression_booléenne> )
  {
    /* instruction(s) lorsque la condition est vraie */
  }
else
  {
    /* instruction(s) lorsqu'elle est fausse */
  }
```

Il faut noter que :

- la clause **else** est facultative
- si on veut exécuter *plusieurs* instructions pour l'une des clauses, il faut les regrouper dans un bloc {...}.

Voici un exemple :

```
int x=4,y=12;
if (x>y)
  {
    printf("x est supérieur à y\n");
    x=x+y;
  }
else
   printf("x est inférieur à y\n");
```

Étant donné qu'il n'y ici a qu'une instruction à exécuter pour la clause **else**, on pourra s'abstenir d'utiliser un bloc {...}.

4.2.2 Alternative multiple : Dans le cas ou... Faire

L'alternative multiple permet d'exécuter des instructions en fonction de l'examen de la valeur d'une variable qui fait office de *sélecteur*. Voyons sa syntaxe sur un exemple :

```
int choix;
...
switch(choix)
  {
  case 1: case 2: case 3:
    /* instructions dans le cas 1, 2 ou 3 */

    break;
  case 4:
    /* instructions dans le cas 4 */
    ...
    break;
  default:
    /* instructions par défaut */
    ...
  }
```

Dans cet exemple, on construit une alternative multiple en distinguant les trois situations suivantes :

❶ le cas où la variable `choix` vaut 1, 2 ou 3

❷ le cas où elle vaut 4

❸ tous les autres cas

Il faut noter d'une part que le mot clef **break** est nécessaire pour quitter la structure de contrôle une fois les instructions exécutées pour un cas donné. D'autre part, il n'est pas nécessaire dans cette structure de grouper les instructions dans un bloc {...}.

4.3 Structures itératives

Le langage C possède trois structures itératives :

❶ une structure post-testée « Faire .. TantQue ... » ;

❷ une structure pré-testée « TantQue ... Faire ... » ;

❸ une boucle « Pour » qui est en réalité une généralisation de la précédente.

4.3.1 Boucle « faire ... tant que ... »

Voyons son utilisation sur un exemple :

```
int compteur=1;

do
  {
    printf("3 x %2d = %2d\n",compteur,3*compteur);
    compteur++;
  }
while (compteur<=10);
```

Ce petit programme affiche la table de multiplication par 3 :

```
3 x  1 =  3
3 x  2 =  6
3 x  3 =  9
3 x  4 = 12
3 x  5 = 15
3 x  6 = 18
3 x  7 = 21
3 x  8 = 24
3 x  9 = 27
3 x 10 = 30
```

On notera que cette boucle est dite « post-testée » dans le sens où le test est effectué après avoir exécuté les instructions, par conséquent celles-ci sont exécutées au moins une fois.

4.3.2 Boucle « Tantque ... Faire ... »

Pour afficher la même table de multiplication on pourrait écrire :

```
int compteur=1;
while (compteur<=10)
  {
    printf("3 x %2d = %2d\n",compteur,3*compteur);
    compteur++;
  }
```

Ici le test précède les instructions (la boucle est dite « pré-testée ») par conséquent les instructions peuvent ne jamais être exécutées.

4.3.3 Boucle « pour »

La boucle « pour » du langage C n'est pas véritablement une boucle « pour » à l'instar de celle du Pascal par exemple. Celle du C peut être vue comme une boucle **while** (...){...} avec des clauses d'initialisation et d'incrémentation intégrées. Ainsi on pourra simplement écrire l'affichage de notre table de multiplication avec une boucle « for » :

```
int compteur;

for(compteur=1;compteur<=10;compteur++)
   printf("3 x %2d = %2d\n",compteur,3*compteur);
```

On peut observer que cette structure de contrôle :

`for (` ⟨*initialisation*⟩ `;` ⟨*test*⟩ `;` ⟨*incrémentation*⟩ `)`

contient trois clause séparées par des ';' :

1. une clause d'*initialisation* ⟨*initialisation*⟩ (on peut initialiser plusieurs variables en séparant les affectations par des virgules) ;
2. une clause de *test* ⟨*test*⟩ ;
3. une clause d'*incrémentation* ⟨*incrémentation*⟩ (on peut également utiliser plusieurs instructions d'incrémentation séparées par des virgules).

L'algorithme correspondant à cette boucle est le suivant :

⟨*initialisation*⟩
TantQue ⟨*test*⟩ **Faire**
  `... instruction(s) de la boucle « pour » ...`
  ⟨*incrémentation*⟩
FinTantQue

Notez bien que la clause ⟨*incr*⟩ est constituée d'une ou plusieurs instructions qui ne sont pas nécessairement des incrémentations.

5 Fonctions

5.1 Prérequis : les pointeurs

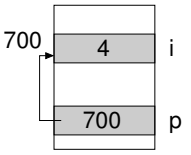
5.1.1 Déclaration

En C on peut déclarer une variable de type pointeur à l'aide de la syntaxe suivante :

```
int* p; /* déclare une variable p pouvant
           stocker l'adresse d'un entier */
```

5.1.2 Affectation

On peut ensuite initialiser ce pointeur en lui affectant l'adresse d'une autre variable :



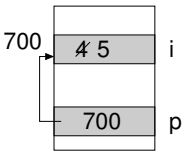

```
int i=4;
int *p;
p=&i; /* p reçoit l'adresse de i*/
```

On utilise ici l'opérateur esperluette (&) qui est l'opérateur « adresse de ». On notera également que pour affecter à `p` l'adresse de `i`, il est nécessaire que `p` soit de type « adresse d'un entier », c'est-à-dire **int**∗.

5.1.3 Déréférencement

Enfin il est possible d'accéder à la zone pointée par `p` : cette opération s'appelle le *déréférencement* :

```
*p=5;
```



ceci met 5 dans la zone pointée par `p` c'est-à-dire `i`. Il est important de comprendre ici que c'est parce que le pointeur est de type **int**∗ que la zone vers laquelle il pointe est considérée comme étant de type entier.

Notez bien que le déréférencement ne sera valide que si l'adresse contenu dans la variable de type pointeur correspond à une zone mémoire à laquelle le programmeur peut accéder, c'est-à-dire :

❶ l'adresse d'une variable ;

❷ l'adresse d'un tableau (cf. § 6.1 page 24) ;

❸ l'adresse d'une zone mémoire allouée dynamiquement (cf. § 7 page 28).

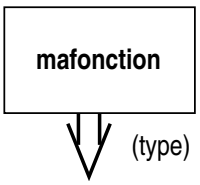
c'est pourquoi, le code :

```
int *p;

p=4;
```

compilera correctement, mais créera un exécutable qui « plantera » très probablement, car il tentera d'accéder à une zone mémoire (dont l'adresse aléatoire est contenue dans `p`) qui ne lui est pas réservée. Par souci de sécurité, le système d'exploitation arrêtera donc l'exécution du programme.

5.2 Valeur renvoyée par une fonction

Pour une fonction renvoyant une valeur, on utilisera l'en-tête suivant :



⟨*type*⟩ `mafonction(...` ⟨*arguments*⟩ `...);`

Dans la définition, on fera apparaître le mot clef **return** :

```
⟨type⟩ mafonction( ... ⟨arguments⟩ ... )
{
  ...
  return ⟨expression⟩;
}
```

Ainsi, par exemple pour créer une fonction renvoyant la valeur 4, on écrira, l'en-tête suivant :

```
int mafonction(); /* fonction renvoyant un entier */
```

et comme définition :

```
int mafonction()
{
  return 4; /* rend le contrôle au bloc appelant
               en renvoyant la valeur 4 */
}
```

Cette fonction pourra être par exemple utilisée comme suit dans un bloc appelant :

```
int i;

i=mafonction();
...
printf("%d\n",mafonction());
```

On notera que si la fonction ne renvoie aucune valeur, il s'agit d'une procédure selon notre langage algorithmique, on utilisera alors le type **void** comme type de la valeur renvoyée :

```
void mafonction(...)
{
  ...
  return ; /* on rend le contrôle sans valeur renvoyée */
}
```

5.3 Arguments

Nous avons vu qu'il y a, d'un point de vue algorithmique, trois natures d'arguments :

❶ donnée

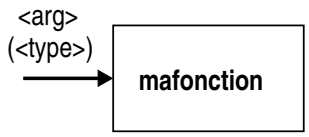
❷ résultat

❸ transformée

Au niveau du langage C, on dispose de deux *modes de passage* des arguments : le passage *par valeur* et le passage `par adresse`. Nous allons voir dans ce qui suit qu'à chaque nature d'arguments correspond un mode de passage.

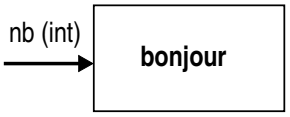
5.3.1 Arguments de nature donnée

Pour une fonction prenant un argument de nature donnée, on utilisera le *passage par valeur*, et on écrira l'en-tête suivant :




```
... mafonction(⟨type⟩ ⟨arg⟩);
```

Dans la définition, on manipulera l'agument formel en utilisant l'identificateur de l'en-tête ; Par exemple pour écrire une fonction qui affiche *nb* fois `bonjour` à l'écran, on écrira l'en-tête :




```
void bonjour(int nb);
```

et la définition :

```
void bonjour(int nb)
{
  ❷
  int i;  /* variable locale à la fonction bonjour */
  ❸

  for(i=0;i<nb;i++)
  printf("bonjour\n");
  return;
}
```

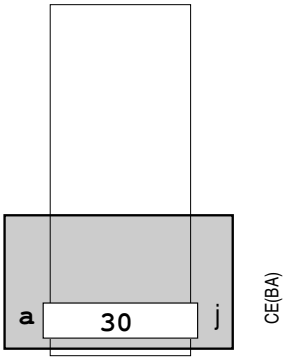
On pourra utiliser cette fonction, dans un bloc appelant comme suit :

```
{
  ...
  bonjour(4); /* affichera 4 fois "bonjour" */
  ...
}
```

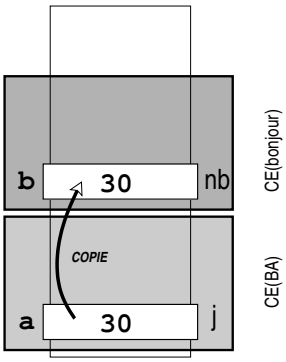
ou

```
{
  int j=30;
  ... ❶
  bonjour(j); /* affichera j fois "bonjour" */
  ... ❹
}
```

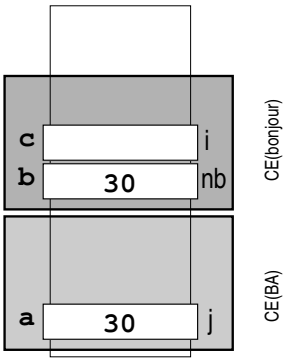
Il est important ici de détailler, l'occupation de la mémoire, et en particulier de la *pile*, avant, pendant, et après l'appel à une telle fonction.



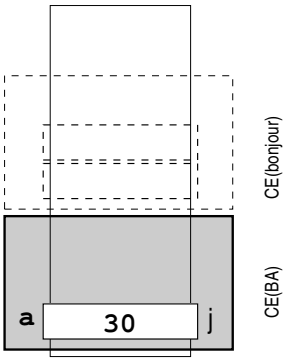
À l'étape ❶, nous sommes dans le contexte d'éxécution (CE) du bloc appelant (BA). La variable locale `j` du bloc appelant est déposée sur la pile à l'adresse `a`.



À l'étape ❷, un nouveau contexte d'exécution est installé sur la pile, pour l'exécution de la fonction `bonjour`. Dans ce contexte est créée une zone pour l'argument `nb`, dans lequel est *copié* la valeur de l'argument effectif.



À l'étape ❸, on crée sur la pile, dans le contexte d'exécution de la fonction `bonjour`, une zone mémoire pour stocker la variable locale `i`.

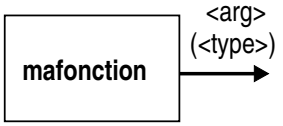


À l'étape ❹, le bloc appelant reprend le contrôle, le contexte d'exécution de la fonction `bonjour` est supprimé ainsi que tout ce qu'il contenait.

Attention : lorsqu'on utilise le passage par valeur, on ne peut modifier l'argument effectif par le biais de l'argument formel dans la fonction. Ainsi, dans la fonction `bonjour` de notre exemple, on ne peut modifier l'argument effectif `j` par le biais de l'argument formel `nb`, puisque ce dernier n'est qu'une copie de `j`.

5.3.2 Arguments de nature résultat

Pour une fonction prenant un argument de nature résultat, on utilisera le *passage par adresse*,[8] et on écrira l'en-tête suivant :



... `mafonction(` ⟨*type*⟩`*` ⟨*arg*⟩ `);`

Dans la définition on utilisera le *déréférencement* pour pouvoir stocker le résultat à l'adresse stockée dans l'argument formel.Par exemple pour écrire une fonction qui prend deux arguments `x` et `y` en données et dont on veut renvoyer la somme à l'aide d'un argument de nature résultat `s`, on utilisera l'en-tête suivant :

8. Qui est en fait un passage par valeur, pour lequel la valeur passée est une adresse.


```
void somme(int x, int y, int* s);
```

et la définition :

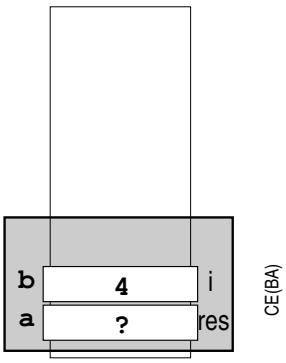
```
void somme(int x, int y, int *s)
{
  ❷
  *s=x+y;
  ❸
  return;
}
```

Pour utiliser cette fonction dans un bloc appelant il faudra passer, pour l'argument de nature résultat, l'adresse de la zone mémoire dans laquelle on veut déposer le résultat. Par exemple :

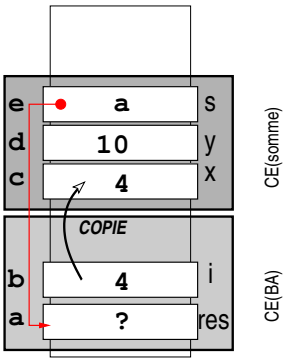
```
int res;
int i=4;

... ❶
somme(i,10,&res);
... ❹
```

On indique donc ici, lors de l'appel à `somme` que la fonction peut stocker le résultat de son calcul dans la variable `res` en passant l'adresse de cette dernière.

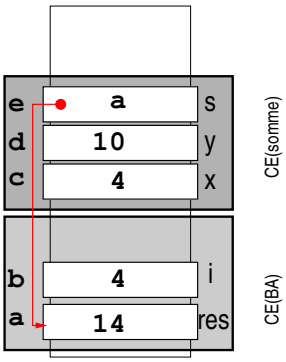
Examinons les états successifs de la pile avant, pendant et après l'appel à `somme`.



À l'étape ❶, les variables locales au bloc appelant, `res` et `i` sont empilées dans le contexte d'exécution.



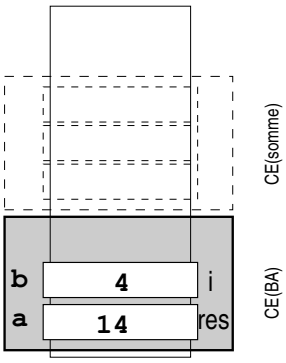
À l'étape ❷ un contexte d'exécution est créé pour la fonction `somme` et des zones mémoires y sont allouées pour chacun des arguments `x`, `y` et `s`. Lors de cette phase `x` et `y` contiennent respectivement les valeurs 4 et 10, et `s` contient l'adresse de `res`.



À l'étape ❸, la ligne :

```
*s=x+y;
```

utilise le déréférencement pour stocker la somme de `x` et `y` à l'adresse passée à la fonction : ici, il s'agit de l'adresse de la variable `res`.

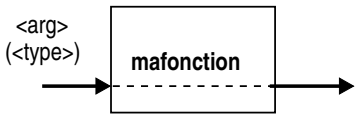


À l'étape ❹, au retour de la fonction, on revient au contexte d'éxécution du bloc appelant ; le contexte d'exécution de `somme` est quant à lui retiré de la pile. Si on examine la mémoire, on constate que l'on a bien le résultat de notre somme dans la variable `res`.

On notera donc que le passage par adresse permet de modifier un argument effectif par le biais d'un argument formel dans une fonction. Ceci est possible puisque l'on passe à la fonction, l'adresse de la zone mémoire (ici c'est une variable) que l'on veut modifier.

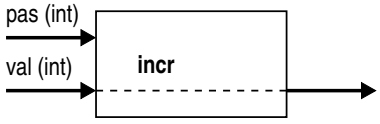
5.3.3 Arguments de nature transformée

Pour une fonction prenant un argument de nature transformée, on utilisera également le *passage par adresse*, on écrira donc, comme précédemment l'en-tête suivant :



... `mafonction(`⟨*type*⟩`*` ⟨*arg*⟩`);`

Dans la définition on utilisera le *déréférencement* pour lire le contenu de l'argument effectif et pour modifier son contenu. Par exemple pour écrire une fonction ayant pour but d'incrémenter une valeur d'un pas donnée, on pourra utiliser la fonction dont le schéma fonctionnel est indiqué ci-contre. Dans ce cas l'en-tête sera :




```
void incr(int* val, int pas)
```

et la définition :

```
void incr(int* val, int pas)
{
  *val=*val+pas;
  return;
}
```

On pourra par exemple utiliser cette fonction comme suit :

```
int entier=10;

...
incr(&entier,2); /* rajoute 2 à la variable entier */
```

Les schémas d'état de la pile sont tout à fait analogues à ceux du cas de l'argument de nature résultat, puisqu'ici aussi on fait un passage par adresse.

5.4 Arguments et types structurés

5.4.1 Rappel sur les tableaux

En langage C, on déclare un tableau, de la manière suivante :

⟨*type*⟩ ⟨*identificateur*⟩ [⟨*nb elts*⟩]

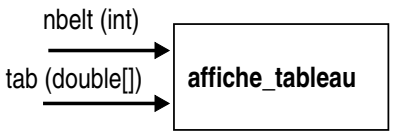
Par exemple :

```
double tab[20];
```

déclare une variable tableau de 20 flottants en double précision. On se rappellera également que l'idenficateur du tableau, lorsqu'il est utilisé sans les opérateurs crochets, renvoie l'adresse du premier élément du tableau. On utilisera cette propriété du langage C pour écrire des fonctions prenant des tableaux en arguments.

5.4.2 Arguments et tableaux

Pour écrire une fonction prenant en argument un tableau, on utilisera le passage par adresse. Par exemple pour écrire une fonction ayant pour but d'afficher le contenu d'un tableau de **double**, on pourra utiliser la fonction dont le schéma fonctionnel est indiqué ci-contre. Dans ce cas l'en-tête sera :



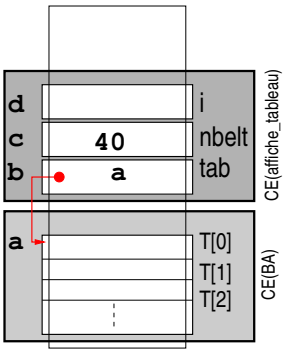

```
void affiche_tableau(const double* tab, int nbelt)
```

et la définition :

```
void affiche_tableau(const double* tab, int nbelt)
{
  int i;
  for(i=0;i<nbelt;i++)
  print("%f\n",tab[i]);
  return;
}
```

On notera l'utilisation du mot clef **const** pour indiquer au compilateur que l'on n'a pas l'intention de modifier la zone mémoire dont l'adresse est `tab` (c'est-à-dire le contenu du tableau). On pourra utiliser cette fonction comme suit :

```
double T[40];

...
affiche_tableau(T,40); /* on passe à la fonction
                          l'adresse du début du tableau */
```



Pendant l'exécution de la fonction `affiche_tableau`, le contexte d'exécution contient donc l'adresse du début du tableau, passée à la fonction par le biais de l'argument `tab`. À l'aide de l'opérateur crochet on peut donc accéder au données de cette zone.

De la même manière, pour écrire une fonction qui par exemple, mettrait à zéro tous les éléments d'un tableau, on pourrait écrire la fonction suivante :

```
void zero_tableau(double* tab, int nb)
{
  int i;
  for(i=0;i<nb;i++)
    tab[i]=0;
}
```

On notera qu'ici, on n'utilise pas le mot clef `const` puisque la fonction va modifier les données du tableau dont on passe l'adresse. Un exemple de bloc appelant serait donc :

```
double T[40];
...
zero_tableau(T,40);
```

5.4.3 Arguments et enregistrements

Pour les enregistrements il sera préférable d'utiliser exclusivement le passage par adresse, on évitera ainsi la surcharge de traitement induite par la copie des données en mémoire lors de l'appel avec un passage par valeur. On aura donc également recours au mot clef **const** si nécessaire. Voici un exemple, utilisant le type suivant :

```
typedef struct {
  double x,y;
} Tpoint;
```

Supposons que l'on souhaite écrire une fonction calculant le milieu d'un segment donné par deux points. On écrira donc le prototype suivant :

```
void mileu(const Tpoint* A, const Tpoint* B, Tpoint* M);
```

La fonction quant à elle pourra s'écrire :

```
void mileu(const Tpoint* A, const Tpoint* B, Tpoint* M)
{
  M->x=( A->x + B->x )/2;
  M->y=( A->y + B->y )/2;
}
```

On notera un particularité du langage C : on peut accéder aux champs d'un enregistrement dont on connaît l'adresse grâce à l'opérateur `->`. Ainsi :

```
A->x
```

est équivalent à :

```
(*A).x
```

En guise d'exemple d'appel à la fonction définie ci-dessus :

```
Tpoint U={0,0}, V={10,10}, W;

...
milieu(&U,&V,&W) /* on stocke le milieu de [UV] dans W */
```

5.5 Structure d'un programme C

Pour l'écriture d'un source en langage C contenant des fonctions, on utilisera la structure suivante :

```
#include <...> /* directives pour le préprocesseur */

/* définitions des types */
typedef ... ;

/* déclarations des fonctions (en-tête) */
int f(...);
double g(...);
void h(...);

/* définitions des fonctions */
int f(...)
{
  return ... ;
}
double g(...)
{
  return ... ;
}
void h(...)
{
  return ;
}

/* programme principal */
int main()
{
  ...
  return 0;
}
```

6 Types structurés

Outre les types prédéfinis du langage vus au paragraphe 2.1, le C offre la possibilité de créer deux types dits *structurés* : le tableau et l'enregistrement :

Tableau : c'est une collection de données homogènes, c.-à-d. un ensemble de données de même type ; on accède à ces données en temps constant grâce à un indice ;

Enregistrement : c'est un ensemble de données hétérogènes, donc de données de types qui peuvent être différents ; on accède à ces données appelées des *champs* par l'intermédiaire d'un identificateur.

Ces deux types structurés peuvent être imbriqués (on peut faire un tableau d'enregistrement, un enregistrement contenant des tableaux, etc.). Cette imbrication est à la base de la création de types pouvant atteindre un haut niveau de complexité, et permet la conception d'objets ayant un degré élevé d'abstraction.

6.1 Type tableau

6.1.1 Définition

Pour créer un tableau en langage C on utilise la déclaration suivante :

```
int tableau[10];
double tabflottant[20];
...
```

On accède à un élément du tableau à l'aide d'un indice, les tableaux sont indicés à partir de 0, ainsi :

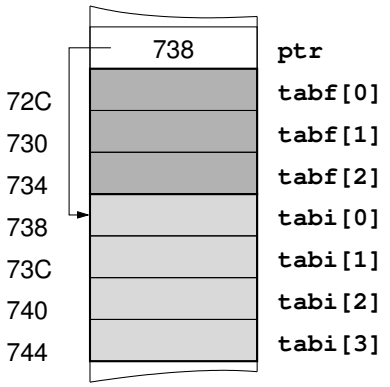
```
tableau[4]=23;
tabflottant[2]=4.123745;
```

permet de mettre deux valeurs dans la 5^{e} case du tableau d'entiers et dans la 3^{e} case du tableau de flottants.

6.1.2 Tableaux et pointeurs

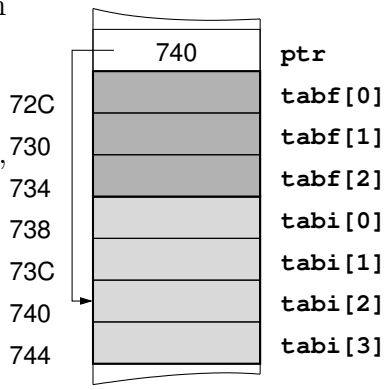
En C, le symbole identifiant le tableau, lorsqu'il est utilisé seul, renvoie l'adresse du premier élément ; ainsi :

```
1 int tabi[4];
2 double tabf[3];
3 int *ptr;
4
5 ptr=tabi;
6 /* ptr pointe sur la première case
7 du tableau d'entier */
```



Il est également possible d'accéder au tableau via un pointeur sur la première case :

```
1    ptr[2]=4;
```



ou même en faisant pointer `ptr` sur une autre case, grâce à l'*arithmétique des pointeurs* :

```
1    ptr+=2; /* on ajoute « 2 »
2                à ptr */
3    ptr[1]=8; /* accès à la 4e case
4                 de tabi */
```

La ligne 9 (`ptr+=2`) ajoute la valeur 2 à `ptr`. Puisque le pointeur est typé (**int**∗), l'adresse contenue dans `ptr` sera augmentée de la valeur 2∗**sizeof**(**int**). La variable `ptr` pointe donc sur la troisième case du tableau `tabi` (cf. figure ci-contre) ; et par conséquent `ptr`[1] fait référence à la quatrième case de ce tableau.

> **Attention :** S'il est possible d'affecter à un pointeur l'adresse d'un tableau, il n'est pas possible d'affecter à un tableau une adresse particulière, ainsi le code :
>
> ```
> int tab1[10], tab2[10];
> int* ptr=tab2 ;
>
> tab1=ptr;
> ```
>
> génère une erreur de compilation à la ligne 4.

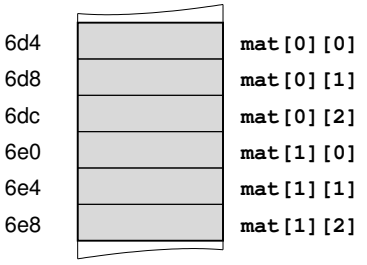
6.1.3 Tableaux multidimensionnels

On peut bien entendu manipuler des variables de type tableau ayant plus d'une dimension. Nous étudierons ici le type tableau à deux dimensions en gardant à l'esprit qu'on pourra généraliser aux dimensions supérieures. Pour déclarer un tableau à deux dimensions, on écrira simplement :

```
1 int mat[2][3];
```

Par curiosité, affichons les adresses de chacune des cases du tableau :

```
int i,j;

for(i=0;i<2;i++)
  for(j=0;j<3;j++)
    printf("adresse de [%d][%d] %p\n",
           i,j,&mat[i][j]);
```



donne :

```
adresse de [0][0] 0xbffff6d4
adresse de [0][1] 0xbffff6d8
adresse de [0][2] 0xbffff6dc
adresse de [1][0] 0xbffff6e0
adresse de [1][1] 0xbffff6e4
adresse de [1][2] 0xbffff6e8
```

On notera donc que, comme dans le cas du tableau mono-dimensionnel :

- les données sont contiguës en mémoire ;
- le symbole `mat` renvoie l'adresse du tableau, ainsi :

```
1  printf("adresse du tableau %p\n",mat);
```

affichera en plus de la liste d'adresse ci-dessus :

```
adresse du tableau : 0xbffff6d4
```

6.1.4 Tableaux et fonctions

Pour passer un tableau en argument à une fonction, il suffira de passer l'adresse de la première case. Par contre, il faut noter qu'une variable de type tableau ne contient pas d'information sur le nombre d'éléments ; il faudra donc en général passer ce nombre en argument à la fonction.

```
void affiche_tableau(double * tab, int taille)
{
  int i;
  for (i=0;i<taille;i++)
    printf("#d : %f\n",i,tab[i]);
}
int main()
{
  double t[20];

  affiche_tableau(t,20);
}
```

Ici aussi c'est l'arithmétique sur les pointeurs qui est utilisée pour évaluer `tab[i]` de la ligne 5. Cette expression est transformée en :

```
tab + i*sizeof(double)
```

puisque `tab` est de de type **double***.

Attention, pour passer un tableau à deux dimensions à une fonction, on doit nécessairement passer un type particulier qui précise les dimensions du tableau. Par exemple, dans le cas d'un tableau à deux dimensions, il faudra écrire :

```
void affiche_matrice(int mat[2][3])
{
  int i,j;
  for(i=0;i<2;i++)
  {
    for(j=0;j<3;j++)
       printf("(%d,%d)=%d ",i,j,mat[i][j]);
    printf("\n");
  }
}
```

ou encore :

```
void affiche_matrice(int mat[][3], int hauteur)
{
```

```
    int i,j;
    for(i=0;i<hauteur;i++)
    {
      for(j=0;j<3;j++)
        printf("(%d,%d)=%d ",i,j,mat[i][j]);
      printf("\n");
    }
  }
```

Le programmeur est contraint de passer au moins la deuxième dimension du tableau de manière à ce que le compilateur puisse évaluer correctement l'expression `mat[i][j]`. En effet, lors de l'évaluation de cette expression, on commence par `mat[i]`, c'est-à-dire :

```
mat + i * sizeof(int[3])
```

on comprend donc que l'adresse de la $\texttt{i}^{\text{e}}$ ligne se situe à l'adresse `mat` décalée de `i` fois la taille de trois **int**. En l'absence de cette information, l'accès à `mat[i][j]` ne pourra être effectué ; c'est pourquoi le code :

```
void affiche_matrice(int mat[][3], int hauteur)
{
  ...
}
```

donnera le message :

```
In function 'affiche_matrice':
 arithmetic on pointer to an incomplete type
```

à ligne où apparaît `mat[i][j]`. Pour pouvoir créer des fonctions permettant de s'affranchir de cette contrainte de dimensions, il est nécessaire de passer à l'allocation dynamique (voir le paragraphe 7 page suivante).

6.2 Enregistrement

Un enregistrement est un type structuré permettant de regrouper plusieurs donnée de types pouvant être différents. Le terme d'enregistrement est emprunté au vocabulaire des bases de données où les données sont organisées en ensembles d'enregistrements (*record*) contenant eux-mêmes un certain nombre de champs (*field*). Par exemple une table contenant des informations sur des personnes regroupe plusieurs enregistrements contenant des champs tels que nom, prénom, âge, sexe, etc.

6.2.1 Définition

Pour créer un enregistrement, le plus simple est de créer un nouveau type de données et d'énumérer la liste des champs qu'il contient :

```
typedef struct {
  char sexe;
  double taille;
  unsigned short age;
} info;
```

Cette structure ou enregistrement contient donc trois champs, chacun d'entre eux étant caractérisé par un identificateur (par exemple `sexe`) et un type (ici **char** pour le champ `sexe`). Une fois le nouveau type défini, on pourra l'utiliser comme un type prédéfini du langage, c'est-à-dire :

- déclarer une variable de ce type ;
- passer un argument de ce type à une procédure ;
- ...

6.2.2 Accès aux champs

Pour accéder aux champs d'une structure on dispose tout d'abord de l'opérateur '.' :

```
info I;

I.sexe='F';
I.age=30;
I.taille=1.75;
```

On peut également affecter directement des valeurs à la structure dans son ensemble en utilisant la notation suivante :

```
info J={'M',1.70,20};
```

Il faudra alors respecter l'ordre de déclaration des champs. Enfin, si l'on dispose de l'*adresse* d'un enregistrement, c'est-à-dire une variable de type pointeur sur cet enregistrement, on pourra utiliser l'opérateur « flèche » pour accéder aux champs :

```
info* pI;

I->sexe='F';
I->age=30;
I->taille=1.75;
```

ici, on suppose bien sûr que la zone mémoire désignée par `pI` est une zone allouée.

6.2.3 Affectation

L'affectation d'une variable enregistrement à une autre, est possible en C, et s'écrit naturellement :

```
int I,J={'M',1.70,20};

I=J;
```

Lors de l'affectation d'une structure à une autre, tous les champs de la structure sont copiés les uns après les autres.

7 Allocation dynamique

Jusqu'ici, les variables que nous avons définies l'ont été *statiquement*. C'est-à-dire que les zones mémoires allouées ne peuvent être explicitement agrandies ou supprimées pendant l'*exécution* du programme. Nous allons voir dans ce paragraphe

comment on peut, *pendant l'exécution* d'un programme, réserver une zone de mémoire pour y déposer des données puis libérer cette zone une fois que les traitements ne nécessitent plus l'accès à cette zone. On dit dans ce cas que l'on utilise l'*allocation dynamique*.

7.1 Principe général

Lorsqu'un langage de programmation offre au programmeur la possibilité d'allouer dynamiquement de la mémoire, le principe est souvent celui-ci :

❶ déclarer une variable de type pointeur ;

❷ demander au système d'exploitation via une fonction particulière l'allocation d'une zone d'une taille donnée ;

❸ si une zone de cette taille est disponible, le système renvoie une adresse que l'on stocke dans la variable de type pointeur ;

❹ utilisation proprement dite de la zone ;

❺ libération de la zone, c'est-à-dire notification au système d'exploitation que la zone en question n'est plus utilisée.

7.2 Allocation dynamique en C

En C, pour réserver dynamiquement une zone de 40 entiers, on commencera donc par déclarer une variable de type pointeur :

```
1 int *zone;
```

Ensuite on pourra effectuer la requête au système d'exploitation pour obtenir une zone de 40 entiers :

```
1 zone=malloc(40*sizeof(int));
```

La fonction `malloc` (nécessitant l'inclusion de la librairie standard, #**include** <stdlib.h>) qui demande au système une zone dont la taille doit être exprimée en octets. Cette fonction renvoie l'adresse de la zone disponible, et la valeur prédéfinie `NULL` sinon. On pourra par exemple arrêter le programme si aucune zone mémoire n'est disponible :

```
1 if (zone==NULL)
2 {
3   printf("Pas de mémoire disponible\n");
4   exit(2); /* arrêt du programme */
5 }
```

Une fois l'adresse de la zone obtenue on peut utiliser cette zone, en y accédant comme avec un tableau, par exemple :

```
1 zone[0]=4;
2 zone[10]=20;
```

Lorsque le programme n'a plus besoin d'utiliser cette zone, il faut procéder à la « libération », à l'aide de la routine `free` :

```
free(zone);
```

7.3 Pointeurs : le retour

Si on examine le prototype de la fonction `malloc` tiré du fichier `stdlib.h`, on trouvera quelque chose du genre :

```
void* malloc(unsigned int taille)
```

On comprend donc que cette fonction attend un entier en argument : le nombre d'octets à allouer ; elle renvoie l'adresse de la zone sous la forme d'un *pointeur non typé*, ou *pointeur générique* : de type **void***. Ceci permet aux concepteurs de la librairie du C de n'écrire qu'une fonction d'allocation pour une zone d'entiers, de flottants, etc.

Le type **void** définit un ensemble vide de valeur. Par conséquent déréférencer un pointeur de type **void*** n'a pas de sens, puisque cela reviendrait à accéder à une zone vide, ainsi dans le code :

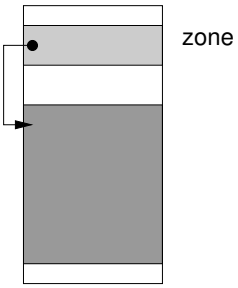
```
void *p=malloc(10);

*p=4;
p[2]=10;
```

les lignes 3 et 4 généreront une erreur à la compilation. En effet, dans ces deux expressions, on tente de déréférencer le pointeur `p`, et d'affecter une valeur de type **int** (4 et 10) à une valeur de type **void** qui par définition ne peut contenir de données.

Un pointeur de type **void*** permet donc de créer des fonctions *génériques*, par exemple la fonction `malloc` alloue une zone mémoire sans connaître le type des données de cette zone. On pourra toujours affecter un pointeur **void*** à un pointeur d'un autre type sans avertissement du compilateur. C'est pour cette raison que l'on peut écrire :

```
zone=malloc(40*sizeof(int));
```

7.4 Propriétés de l'allocation dynamique



Le schéma correspondant à l'état de la mémoire après l'appel à la fonction `malloc` est celui de la figure ci-contre. On notera donc que contrairement à l'allocation statique où les zones sont référencées par des identificateurs, ici on ne dispose que de l'adresse de la zone. [9]

Il faut également noter que :

❶ la zone, une fois libérée par la fonction `free` est « marquée » comme libre par le système d'exploitation, mais n'est pas modifiée ; et que :

❷ après l'appel à `free`, la variable de type pointeur (ici `zone`) contient toujours l'adresse de la zone. On peut par conséquent toujours accéder à la mémoire à cette adresse avec les risques que cela comporte.

Par conséquent, il est préférable :

❶ de toujours initialiser les pointeurs à `NULL` avant l'appel à `malloc` ;

❷ de toujours affecter la valeur `NULL` aux pointeurs après l'appel à `free`.

9. On trouve parfois le terme de variable anonyme.

7.5 Portée et durée de vie des zones allouées

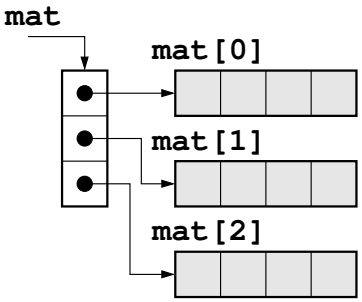
Pour faire un parallèle avec les variables locales et globales et leur durée de vie, on peut noter que :

❶ la durée de vie d'une zone allouée est entièrement à la discrétion du programmeur : c'est lui qui décide du début de l'allocation (appel à `malloc`) et de la fin de la vie de la zone en mémoire (appel à `free`) ;

❷ la portée de la zone allouée dans le source est une question qui n'a pas lieu d'être. En effet, les variables (locales ou globales) sont accessibles via un identificateur, auquel est associé une portée. Dans le cas de l'allocation dynamique, la zone n'est accessible que via son adresse. Par conséquent cette zone sera accessible de n'importe quel endroit du source, à condition de connaître son son adresse, qui sera la plupart du temps stockée dans une variable de type pointeur.

7.6 Tableaux multi-dimensionnels dynamiques

Nous allons étudier dans ce paragraphe, comment on peut utiliser l'allocation dynamique pour mémoriser des tableaux multi-dimensionnels ; pour l'exemple nous nous intéresserons au tableau à deux dimensions, tout en sachant qu'on pourra extrapoler aux dimensions supérieures. Le principe d'un tableau dynamique de $N \times M$ **int** est le suivant (illustré ci-contre avec $N = 3$ et $M = 4$) :



- ➟ on alloue dynamiquement une zone pour N **int**∗, c'est-à-dire un tableau stockant les adresses de N zones destinées recevoir des entiers ;
- ➟ on alloue N zones contenant M entiers et on stocke l'adresse de ces zones dans le tableau précédemment alloué.

Tout ceci se traduit en langage C par le code suivant :

```
1 int **mat;
```

`mat` est l'adresse d'une zone contenant des adresses de zones contenant des entiers,

```
1 mat=malloc(N*sizeof(int*));
```

on alloue une zone susceptible de recevoir N adresses d'entiers,

```
1 for(i=0;i<N;i++)
2     mat[i]=malloc(M*sizeof(int));
```

on affecte à ces adresses, les adresses de zones allouées dynamiquement, et contenant M entiers. Les instructions pour libérer les zones allouées après leur utilisation est laissée en exercice `;-)`. On pourra ensuite écrire une procédure d'affichage quelque peu différente de celle vu au paragraphe 6.1.4 page 26 :

```
void affiche_matrice(int** mat, int hauteur, largeur)
{
  int i,j;
  for(i=0;i<hauteur;i++)
  {
```

```
        for(j=0;j<largeur;j++)
            printf("(%d,%d)=%d ",i,j,mat[i][j]);
        printf("\n");
    }
}
```

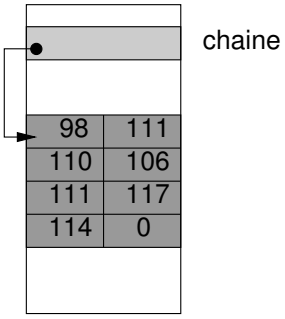
8 Chaînes de caractères

Il n'y a pas de type chaîne de caractères en C, mais le langage permet de manipuler ce type d'objet par le biais de tableaux de caractères et de l'allocation dynamique.

8.1 Définition

Une chaîne de caractères en C est un tableau de caractères (pouvant être alloué statiquement ou dynamiquement selon le contexte). Ce tableau a ceci de particulier que le dernier caractère a pour code 0, on désigne souvent ce caractère par par \0 et on le nomme le *caractère de fin de chaîne*.
En C, on pourra écrire :

```
char* chaine;

chaine="bonjour";
```



La variable chaîne qui n'est autre qu'un pointeur sur une zone contenant des caractères, ou encore un tableau de caractère, contiendra l'adresse de la zone où seront stockés les codes des caractères composant la chaîne "bonjour" plus le caractère de code 0.

On verra que l'on peut également utiliser des tableaux de caractères pour manipuler les chaînes :

```
1 char chaine[256];
```

déclare un tableau de 256 caractères que l'on pourra utiliser pour y stocker une chaîne. Par contre l'instruction :

```
1 chaine="bonjour";
```

générera une erreur de compilation, puisque dans ce cas on cherche à modifier l'adresse de la variable tableau, ce qui est impossible.

8.2 Opérateurs sur les chaînes

Les opérateurs sur les chaînes sont accessibles via des fonctions et nécessitent l'ajout de :

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

en tête de programme.

8.2.1 Longueur d'une chaîne

On obtient la longueur d'une chaîne grâce à la fonction `strlen` :

```
char* chaine="bonjour";

printf("La longueur de la chaîne 'bonjour' est %d.\n",
        chaine,
        strlen(chaine));
```

Ce programme affichera :

```
La longueur de la chaîne 'bonjour' est 7.
```

On notera que pour faire afficher une chaîne de caractères à la fonction `printf` on utiliser le format `%s` (pour *string*).

8.2.2 Comparaison de deux chaînes

La fonction permettant de comparer deux chaînes c_1 et c_2 est `strcmp`. Elle renvoie :

- 0 si les deux chaînes sont identiques ;
- un nombre positif si c_1 est supérieure à c_2 ;
- un nombre négatif si c_1 est inférieure à c_1.

L'ordre utilisé ici est l'ordre lexicographique :

- `"a"` < `"b"`
- `"abc"` > `"ab"`
- `"a"` > `"A"`
- ...

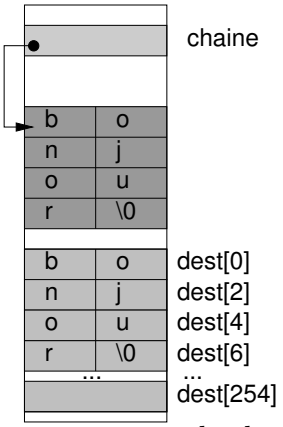
En supposant que l'on ait déclaré deux chaînes `c1` et `c2`, un test d'égalité entre ces deux chaînes en C, serait :

```
if (strcmp(c1,c2)==0)
{
   printf("%s et %s sont identiques\n",c1,c2);
}
```

8.2.3 Copie

La fonction `strcpy` permet de copier une chaîne dans une autre :

```
char* chaine="bonjour";
char dest[256];

strcpy(dest,chaine);
```



Il est très important de comprendre que la fonction `strcpy` suppose que la chaîne destination est une zone mémoire allouée. Ainsi le code :

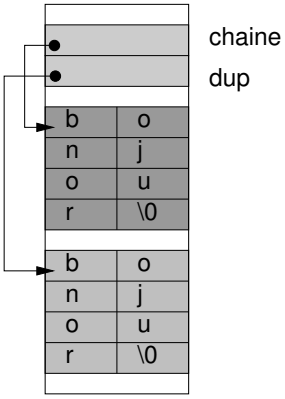
```
char* chaine="bonjour";
char* destination;

strcpy(destination,chaine);
```

compilera correctement, mais générera une erreur à l'exécution du programme, car la zone mémoire correspondant à `destination` n'a pas été allouée. Enfin, il est aussi important de noter que la fonction `strcpy` suppose que la chaîne source (ici `chaine`) est bien terminée par le caractère de fin de chaîne.

8.2.4 Duplication

On peut dupliquer une chaîne grâce à la fonction `strdup`.

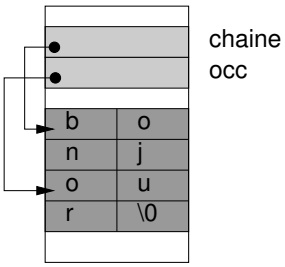
```
char* chaine="bonjour"; char* dup;

dup=strdup(chaine);
```



Cette fonction fait un appel à la routine `malloc` pour allouer suffisamment de mémoire pour contenir la chaîne "bonjour" et renvoie l'adresse de la zone allouée. Par conséquent il faudra libérer cette zone avec la fonction `free`.

8.3 Recherche

Il existe plusieurs fonctions standard permettant de rechercher l'occurrence d'un caractère ou d'une sous-chaîne dans une chaîne. C'est le cas de la fonction `index` cherche la première occurrence d'un caractère dans une chaîne donnée :




```
1 char* occ;
2 char* chaine="bonjour";
3
4 occ=index(chaine,'o');
```

Après l'exécution de la fonction `index`, la variable `occ` reçoit l'adresse du premier caractère 'n' dans la chaîne "bonjour". `occ` reçoit donc ici : `chaine`+1. On pourra relancer cette fonction pour chercher l'occurrence suivante :

```
1 occ=index(occ+1,'o');
```

en cherchant à partir du caractère suivant ; le pointeur `occ` contiendra alors l'adresse de la zone mémoire contenant le deuxième caractère 'o' (cf. figure ci-dessus). À la troisième recherche, on ne trouvera pas de caractère 'o', `index` renverra `NULL`.

Pour information, la fonction `strstr` effectue le même type de recherche que `index`, mais le motif recherché est une chaîne et non plus un caractère.

8.4 Conversion vers un autre type

8.4.1 Conversion vers un nombre

On peut convertir une chaîne de caractères en type flottant à l'aide des fonction `atoi` et `atof` qui permettent de transformer la chaîne de caractère respectivement

en un nombre de type entier (**int**) ou flottant (**double**). Par exemple, on pourra écrire :

```
char c1="12340", c2="2.456e45";
int entier;
double flottant;

entier=atoi(c1);
flottant=atof(c2);
```

Ces deux fonctions ne détectent pas les erreurs de conversion (par exemple si la chaîne à convertir ne contient pas un nombre valide) ; pour pouvoir détecter les éventuelles erreurs il faudra utiliser la fonction `strtol` dont on ne donnera pas le fonctionnement ici. Notez que pour utiliser ces fonctions il faudra insérer en tête de programme :

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

8.4.2 Conversion depuis un nombre

On veut souvent pouvoir convertir un nombre (entier ou flottant) en une chaîne de caractères. Le plus simple dans ce cas est de faire appel à la fonction de la librairie standard d'entrée sortie `sprintf`. On peut utiliser cette fonction de la manière suivante :

```
char str[10];
int i=4;

sprintf(str,"%03d",i);
printf("contenu␣de␣la␣chaîne␣:␣%s\n",str);
```

Ce programme permet donc d'écrire la valeur de `i` (ici 4) dans la chaîne `str`. On utilisera la même syntaxe que la fonction `printf`. Ce programme affichera donc :

```
contenu de la chaine : 004
```

Notez que pour utiliser cette fonctione il faudra insérer en tête de programme :

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

8.5 Les arguments de la ligne de commandes

Dans un système comme Unix, il est très fréquent d'exécuter et un programme dans un interpréteur de commande. Dans ce cas, l'utilisateur peut passer des informations au programme par le biais de la ligne de commande. Ainsi, lorsqu'on écrit :

```
mkdir temp
```

on exécute le programme `mkdir` qui crée un répertoire dont le nom (ici `temp`) est donné par l'utilisateur. On conçoit donc qu'il est nécessaire au concepteur du programme `mkdir` d'avoir accès à l'argument de la ligne de commande, c'est-à-dire le nom du répertoire que l'utilisateur veut créer.

Dans la procédure `main` d'un programme écrit en C, on peut accéder :

❶ au nombre d'arguments de la ligne de commande ; on notera que :

- ➟ `cp -i truc.txt /tmp` en comporte 4 ;
- ➟ `mkdir temp` en comporte 2 ;
- ➟ `./test 40 bidule.dat 2` en compote 4.

❷ à chacun des arguments sous la forme d'une chaîne de caractères.

Ainsi on pourra écrire :

```
#include <stdio.h>
int main(int nbargs, char** args)
{
  int i;
  for(i=0;i<nbargs;i++)
  {
    printf("L'argument %d est : [%s]\n",i,args[i]);
  }
}
```

Si l'on nomme l'exécutable correspondant à ce source, `test`, et qu'on lance la commande :

```
./test 50 truc.txt
```

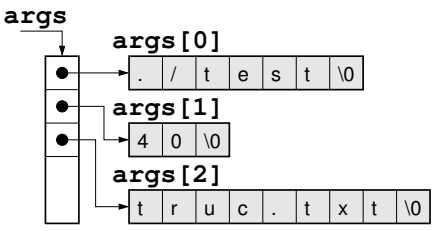
le programme affichera à l'écran :

```
L'argument 0 est : [./test]
L'argument 1 est : [50]
L'argument 2 est : [truc.txt]
```

On peut donc comprendre par le truchement de cet exemple :

- ➟ que la procédure `main` peut exploiter deux arguments dont les types sont prédéfinis :
  - ❶ le premier de type **int** contenant le nombre d'arguments de la ligne de commande ;
  - ❷ le deuxième de type **char**$**$ contenant un tableau de chaîne de caractères.
- ➟ ces deux arguments sont nommés à la discrétion du programmeur ; on trouve très souvent dans les manuels les noms de `argc` pour le premier (*argument counter*) et `argv` pour le deuxième (*argument values*).

Pour comprendre ce qu'est un tableau de chaîne de caractères examinons la figure ci-contre. Si `args` est de type **char**$**$, on peut se le représenter comme l'adresse d'une zone contenant des **char**$*$, ou encore comme un tableau contenant des **char**$*$. **char**$*$ étant bien entendu le type pour stocker les chaînes de caractères.



9 Bibliothèque d'entrée/sortie

Les fonctions de la bibliothèque d'entrée/sortie standard sont accessibles à condition d'inclure en tête de programme le fichier `stdio.h`, en utilisant :

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

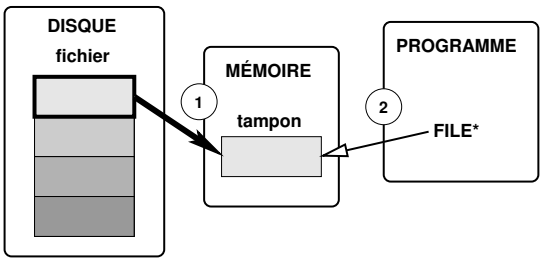
Ces fonctions permettent de manipuler des fichiers en lecture et écriture ; on verra également que l'affichage d'information à l'écran, et la lecture depuis le clavier s'apparente, en langage C et *a fortiori* sur le système Unix, à une manipulation de fichier.

9.1 Principe général sur les fichiers

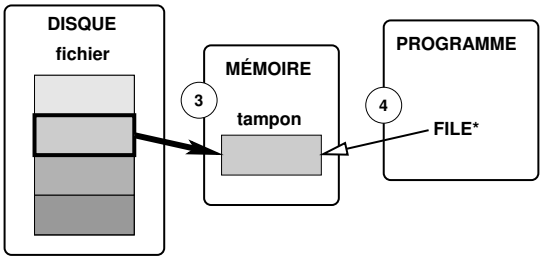
En langage C, on manipule les fichiers par l'intermédiaire d'une structure de données particulière : `FILE`. Cette structure permet au programmeur d'accéder au système de fichier du système d'exploitation. La bibliothèque standard d'entrée/-sortie (`stdio.h`), propose de manipuler les fichiers par l'intermédiaire d'un *tampon mémoire* (*buffer*). Ce tampon suit le principe de la *mémoire cache* et permet donc d'accélérer l'accès aux données stockées sur les disques. [10]

9.1.1 Lecture « bufferisée »

Pour lire un fichier à l'aide d'un tampon, les routines de lecture de la bibliothèque vont charger une partie des données du fichier stocké sur le disque, dans la zone tampon. Le programme ira donc lire les données du fichier dans cette zone mémoire.

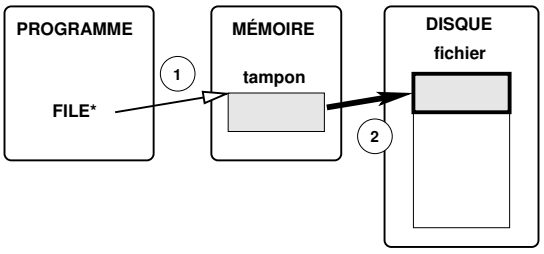


Lorsque que le programme lit la fin du tampon, les fonctions d'accès en lecture de la bibliothèque, vont charger le buffer avec de nouvelles données provenant du fichier. Tout ceci se passe naturellement sans intervention du programmeur.

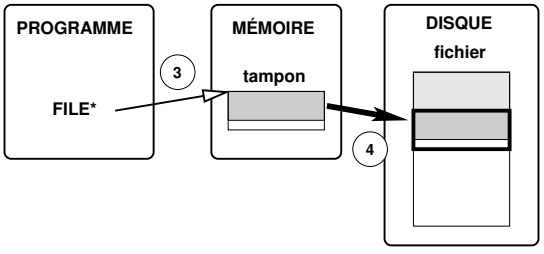


9.1.2 Écriture « bufferisée »

Pour écrire dans un fichier par l'intermédiaire du tampon, le programme va d'abord écrire en mémoire dans le buffer. Lorsque ce buffer est rempli, les données de ce dernier sont transférées dans le fichier sur le disque.



Lorsque le programme a fini d'écrire les données dans le tampon, une opération particulière appelée *vidange* (*flush*), permet de forcer l'écriture du contenu du tampon dans le fichier.



10. Le système d'exploitation propose lui aussi son propre système de cache pour l'accès aux fichiers.

9.1.3 La notion de flux

L'utilisation d'un tampon mémoire est nécessaire puisqu'on ne peut imaginer à chaque ouverture d'un fichier, ni que l'intégralité de son contenu soit transférée en mémoire, ni que chaque opération de lecture nécessite un accès disque. Malgré tout, cette solution implique que l'on accède *séquentiellement* aux données des fichiers. La structure `FILE` que l'on appelle *flux* (*stream* en anglais) permet de mettre en œuvre ces accès séquentiels. Il est assez aisé de faire une analogie entre l'utilisation des flux et l'utilisation d'un dispositif à bande (telle une cassette vidéo), utilisation dans laquelle on est amené à *rembobiner*, *avancer*, etc. pour accéder aux données. On verra que la bibliothèque d'entrée sortie propose ce genre de routines pour accéder aux données d'un fichier.

9.2 Manipulation des fichiers en C

En C, on procédera en quatre étapes :

❶ déclaration d'une variable de type `FILE*` qui permet d'interfacer un programme avec le système de fichier du système d'exploitation ;

❷ instanciation de cette variable à l'aide de la fonction `fopen`. Cette fonction permet d'« ouvrir » un fichier dont on précise le nom dans le système de fichier. On pourra utiliser trois mode : lecture, écriture et ajout.

❸ opération d'écriture ou de lecture à l'aide de `fprintf` et/ou `fscanf` ;

❹ fermeture avec vidange éventuelle du buffer à l'aide de `fclose`.

9.2.1 Mode texte ou mode binaire

Les fonctions `fprintf` et `fscanf` permettent d'écrire ou de lire en mode texte ; en d'autres termes, lorsqu'on dispose d'une variable :

```
int i=12345678;
```

on pourra écrire le contenu de cette variable avec très exactement 9 caractères dans le fichier en question, 8 pour l'entier `i`, et 1 pour le caractère de saut de ligne. Si le système sous-jacent utilise un octet pour coder chacun des caractères, le fichier occupera donc 5 octets sur le disque.

Cependant dans certaines situations, on veut écrire dans un fichier un nombre sous la même forme que sa représentation mémoire ; par exemple 4 octets pour un entiers, 8 octets pour un flottant, etc. Dans ce cas on utilisera le mode d'écriture/-lecture binaire et les fonctions de la bibliothèque standard : `fread` et `fwrite`. Dans ce cas les variables :

```
double f=2.3;
double g=1.2233445566778899e43;
```

une fois écrites en mode binaire dans un fichier, occuperont chacune 8 octets (8 octets étant la taille d'un flottant à double précision).

9.2.2 Écriture en mode texte dans un fichier

Le code suivant écrit les données la table de multiplication par 7 dans un fichier texte :

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  FILE* fichier=fopen("table.dat","w");
  int i;

  for(i=1;i<=10;i++)
     fprintf(fichier,"7 x %2d = %2d\n",i,7*i);
  fclose(fichier);
}
```

En ouvrant le fichier `table.dat`[11] avec un éditeur de texte, on y trouverait :

```
7 x  1 =  7
7 x  2 = 14
7 x  3 = 21
7 x  4 = 28
...
```

On notera donc, que la fonction de la bibliothèque d'entrée/sortie `fprintf` s'utilise comme la fonction `printf`, à ceci près que le premier argument doit être l'adresse d'un *flux* (*stream*) c'est-à-dire un argument de type `FILE*`.

L'appel à la fonction `fclose` est nécessaire pour :

❶ vidanger le flux, et ainsi s'assurer que les données sont bien écrites dans le fichier ;

❷ informer le système d'exploitation que le fichier `table.dat` n'est plus utiliser.

9.2.3 Lecture en mode texte dans un fichier

On peut lire dans un fichier grâce à la fonction `fscanf`, à condition d'ouvrir le fichier en lecture avec `fopen` (argument "r"). Par exemple si on voulait lire les données d'un fichier nommé `freejazz.txt` contenant :

```
Coltrane John 1967
Ayler Albert 1970
Dolphy Eric 1964
```

On pourrait utiliser le code :

```c
#include <stdio.h>
int main()
{
  char nom[100], prenom[100];
  int  date;
  FILE* fichier=fopen("freejazz.txt","r");

  while ( ! feof(fichier) )
  {
    fscanf(fichier,"%s %s %d",nom,prenom,&date);
```

11. le fichier `table.dat` est créé dans le répertoire de travail où est lancé l'exécutable.

```
    printf("%s %s nous a quitté en %d\n",prenom,nom,date);
  }
  fclose(fichier);
}
```

qui donnera à l'écran :

```
John Coltrane nous a quitté en 1967
Albert Ayler nous a quitté en 1970
Eric Dolphy nous a quitté en 1964
```

Ici la fonction `feof` renvoie vrai ou faux, selon que la fin du fichier (*end of file*) a été atteinte.

Il est important de noter que la fonction `fscanf` attend comme `fprintf` une chaîne de format, mais que les arguments suivants doivent être des *adresses*. En l'occurrence, il s'agit des adresses des variables dans lesquelles on veut stocker la valeur lue depuis le fichier. [12]

Enfin, les espaces de la chaîne de format de la fonction `fscanf`, lorsqu'elle comporte plusieurs arguments, correspondront à un ou plusieurs des caractères : espace, tabulation, saut de ligne. Ainsi, si dans le format de l'exemple précédent :

```
"%s %s %d"
```

la fonction `fscanf` cherche deux chaînes suivies d'un entier ; ces trois entités pouvant être séparées par un nombre quelconque d'espaces, de tabulations ou de sauts de ligne.

9.2.4 Écriture en mode binaire dans un fichier

Supposons que l'on veuille écrire le résultat de la table de multiplication par 7, dans un fichier en mode binaire. On va donc écrire dans un fichier la séquence (7, 14, 21, etc.) comme des entiers de type **int**. On pourra faire comme suit, après avoir ouvert le fichier comme précédemment en écriture :

```
int i,res;
FILE* fichier=fopen("table.dat","w");
for(i=1;i<=10;i++)
{
  res=7*i;
  fwrite(&res,sizeof(int),1,fichier);
}
fclose(fichier);
```

la fonction `fwrite` attend les arguments suivants :

- l'adresse de la zone à copier dans le fichier ;
- la taille de chacun des éléments ;
- le nombre de ces éléments ;
- le flux

Par conséquent, si on stocke la table dans un tableau d'entier :

12. Tout ceci n'a rien de bien étonnant puisqu'il est nécessaire que la fonction `fscanf` modifie l'argument qu'elle reçoit (cf. § 5 page 14).

```
int tabmult[10];
```

on peut réaliser l'écriture en une instruction :

```
fwrite(tabmult,sizeof(int),10);
```

9.2.5 Lecture en mode binaire dans un fichier

L'opération de lecture est l'opération inverse et peut être réalisée grâce à la fonction `fread`. Si on voulait lire le fichier précédemment écrit en mode binaire, il faudrait ouvrir le fichier en lecture puis :

```
int table[10];
FILE* fichier=fopen("table.dat","r");

fread(table,sizeof(int),10,fichier);
fclose(fichier);
```

9.3 Écran et clavier

9.3.1 Écran

On a vu précédemment l'utilisation de la fonction `printf` pour afficher des messages à l'écran (voir en particulier le paragraphe 2.3 page 7). Il est intéressant de noter que la fonction :

```
printf(...);
```

est équivalente à :

```
fprintf(stdout,...);
```

par conséquent tous les programmes ont accès à un flux spécial, appelé *sortie standard*, dirigé vers l'écran. Ce flux n'a pas à être déclaré dans un programme.

9.3.2 Clavier

Nous avons superbement ignoré jusqu'ici comment il était possible de demander à l'utilisateur d'entrer des données au programme via le clavier. Réparons cette injustice en étudiant la fonction de la bibliothèque standard, `scanf`. Tout d'abord voici un exemple :

```
#include <stdio.h>

int main()
{
  int i;
  printf("Veuillez entrer un entier svp.\n");
  scanf("%d",&i);
  printf("Vous avez saisi : %d.\n",i);
  /* ... */
```

Ce programme affichera à l'écran :

```
Veuillez entrer un entier svp.
```

Puis l'exécution de la fonction `scanf` à la ligne 7, interrompra momentanément le programme pour que l'utilisateur puisse entrer une valeur (par ex. 24). Le `printf` de la ligne 8 affichera ensuite :

```
Vous avez saisi : 24.
```

Comme pour la fonction `printf`, il est intéressant de noter que la fonction :

```
scanf(...);
```

est équivalente à :

```
fscanf(stdin,...);
```

par conséquent tous les programmes ont accès à un flux spécial, appelé *entrée standard*, dirigé vers le clavier. Ce flux n'a pas à être déclaré dans un programme.

9.4 Gestion des erreurs

9.4.1 Erreur à l'ouverture

Si la fonction `fopen` échoue elle renverra la valeur `NULL`. L'ouverture d'un fichier en écriture (argument "w" de `fopen`) peut échouer si :

❶ le fichier n'est pas modifiable ;

❷ le répertoire on où tente de créer le fichier est en lecture seule.

Ce qui sous-entend qu'on peut écrire dans un fichier qui n'existe pas avant l'appel à `fopen`, dans ce cas précis un nouveau fichier est créé. Il faut également noter que si le fichier existe déjà son contenu est écrasé.

Une fonction très utile de *stdio.h*, est la fonction `perror` ; elle permet d'afficher un message d'erreur concernant le dernier appel système. On peut utiliser cette fonction comme suit :

```
if ( (fichier=fopen("table.dat","w"))==NULL )
{
  perror("Problème␣à␣l'ouverture␣de␣table.dat␣");
  exit(1);
}
```

qui donne à l'exécution sur le système de votre serviteur :

```
Problème à l'ouverture de table.dat : Permission denied
```

9.4.2 Erreurs à la lecture

La fonction `fscanf` peut parfois sembler avoir un comportement étrange, certains d'ailleurs n'hésitent pas à la qualifier de diabolique. Nous étudierons ici deux exemples illustrant le fonctionnement des flux d'entrée. Le premier exemple est celui du paragraphe 9.2.3 page 39. Pour mémoire, la boucle de lecture dans le fichier texte, est la suivante :

```
while ( ! feof(fichier) )
{
  fscanf(fichier,"%s␣%s␣%d",nom,prenom,&date);
  printf("%s␣%s␣nous␣a␣quitté␣en␣%d\n",prenom,nom,date);
}
```

Si le fichier que l'on ouvre contient :

```
Coltrane        John 1967
Ayler           Albert 1970
Dolphy Eric 1964
```

c'est-à-dire trois lignes plus des sauts de ligne à la fin du fichier ; le programme affichera à l'écran :

```
John Coltrane nous a quitté en 1967
Albert Ayler nous a quitté en 1970
Eric Dolphy nous a quitté en 1964
Eric Dolphy nous a quitté en 1964
```

ce qui paraît pour le moins curieux. Il y a bien entendu une explication qui est la suivante : la chaîne de format "%s␣%s␣%d" permet de lire les trois premières lignes du fichier sans encombre. Après la lecture de la troisième ligne la « tête de lecture » du flux se trouve sur le caractère de saut ligne qui suit le '4' de '1964' ; par conséquent le flux n'est pas en « end of file » et donc la fonction `feof` renvoie faux. Une autre itération est alors effectuée ; lors de cette itération la fonction `fscanf` ne réussira pas à lire ce qu'on lui demande (deux chaînes et un entier) et n'affectera pas le contenu de ses arguments (`nom`, `prenom` et `date`). Ces derniers seront donc une nouvelle fois affichés par `printf` avec leur contenu inchangé c'est-à-dire ("Eric", "Dolphy" et 1964).

Pour éviter cette erreur il y a plusieurs solutions. La première est d'exploiter la valeur que renvoie la fonction `fscanf`. Cette valeur correspond au nombre d'élément qui a été réellement instancier à la lecture. En d'autres termes dans notre exemple si `fscanf` réussit a décoder les deux chaînes et l'entier, cette valeur sera 3. On pourra donc modifier le code de lecture comme suit :

```
while ( ! feof(fichier) )
 {
   /* on affiche que si on a bien lu 3 éléments */
   if (fscanf(fichier,"%s␣%s␣%d",nom,prenom,&date)==3)
     printf("%s␣%s␣nous␣a␣quitté␣en␣%d\n",prenom,nom,date);
 }
```

À noter que la fonction `fscanf` peut renvoyer la valeur `EOF` si pendant que la tentative de lecture, la fin de fichier survient.

Un autre solution pourrait être de demander à chaque lecture de ligne, de lire les deux chaînes, l'entier, et autant de saut ligne qu'il faudra. Ceci peut être fait grâce au code suivant :

```
while ( ! feof(fichier) )
{
  fscanf(fichier,"%s␣%s␣%d␣%*[\t\n]",nom,prenom,&date);
  printf("%s␣%s␣nous␣a␣quitté␣en␣%d\n",prenom,nom,date);
}
```

Ici la chaîne de format contient en plus : "%[\t\n]*" qui signifie :

- `%[`⟨*liste de caractère*⟩`]` : on cherche à lire dans le flux une chaîne de caractères sans espace, formée des caractères listés entre crochets. Donc dans notre exemple, on cherche à lire une chaîne composés de caractères saut de ligne (`\n`) ou tabulation (`\t`) ;
- `%*`⟨*format*⟩ : le caractère `*` spécifie que la chaîne lue dans le flux ne sera pas affectée à une variable.

On peut utiliser ce format particulier pour mettre en place une boucle de contrôle de la saisie. Pour mettre en œuvre un l'algorithme de saisie : « répéter la saisie d'un entier tant que l'utilisateur ne saisit pas en entier » on peut écrire le code suivant :

```
int nbval,entier;
do
  {
    printf("Entier␣?\n");
    nbval=scanf("%d",&entier);
    if (nbval==0)
      {
        printf("ce␣n'est␣pas␣un␣entier...\n");
        scanf("%*s");
      }
  }
while (nbval!=1);
```

La ligne 9 est nécessaire ici pour retirer du flux la valeur qui n'est pas un entier.

9.5 Contrôle des flux

9.5.1 Vidange

L'opération de vidange (*flush*) consiste à forcer l'écriture du tampon mémoire dans le fichier. Cette opération n'a donc de sens que pour les flux de sortie : par exemple `stdout` et les fichiers ouverts en écriture. Cette vidange est réalisée dans plusieurs cas :

- lorsqu'on envoie le caractère '\n' sur `stdout` ;
- lorsqu'on appelle la fonction `fclose` ;
- en appelant explicitement la fonction `fflush` prenant un flux de sortie en argument.

9.5.2 Positionnement explicite de la tête de lecture

Deux fonctions de la bibliothèque d'entrée sortie permettent de positionner explicitement la « tête de lecture » du flux :

- `fseek` permet de positionner la tête à un endroit particulier ;
- `ftell` permet de savoir où est positionner la tête ;
- `rewind` permet de « rembobiner » un fichier, c'est-à-dire le ramener la tête de lecture au début.

La position de la tête de lecture est manipulée sous la forme d'un déplacement en octets. Pour la fonction `fseek` on peut se déplacer, soit de manière absolue dans le fichier, soit relativement à la position courante ou relativement à la fin du fichier.

10 Compilation séparée

La figure 1 met en évidence les outils qui entrent en jeu dans la création d'un exécutable à partir d'un source en langage C. On distingue donc plusieurs étapes :

❶ l'appel au *préprocesseur* qui transforme le source en un autre source ; le préprocesseur permet de définir des macros, d'inclure des fichiers, d'effectuer une compilation conditionnelle (voir § 10.1) ;

❷ l'appel au *compilateur* qui crée à partir du code source, du code objet

❸ l'appel à l'*éditeur de liens* qui va résoudre les appels aux fonctions non résolus en incluant du code objet requis notamment sous la forme de bibliothèque.

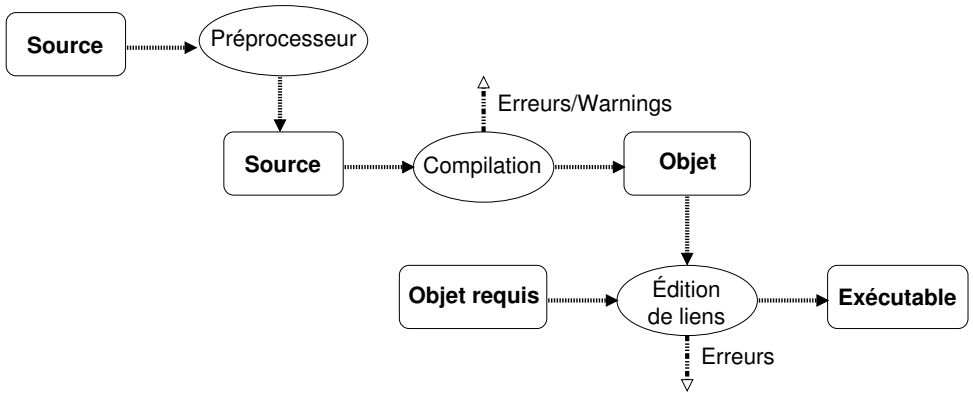


FIGURE 1 – Production d'un exécutable

10.1 Le préprocesseur

Le préprocesseur a plusieurs rôles qui consistent tous à transformer le source avant la phase de compilation. Il s'agit donc presque d'un outil de « traitement de texte ».

10.1.1 Macros

L'utilisation la plus simple du préprocesseur est d'insérer dans le source :

```
#define __MAX 100
```

cette ligne informe le préprocesseur qu'il faudra, avant de compiler le source, remplacer toutes les occurrences de `__MAX` par la chaîne 100. Il est possible de définir des macros prenant des arguments, comme par exemple :

```
#define CARRE(x) (x)*(x)
#define PLUSPETIT(a,b) (a)>(b)?b:a
```

qui transformera le source :

```
int i,j;

i=CARRE(j+1);
j=PLUSPETIT(4,2*i);
```

en :

```
int i,j;

i=(j+1)*(j+1);
j=(4)>(2*i)?2*i:4;
```

10.1.2 Compilation conditionnelle

On peut demander au préprocesseur d'inclure ou non une portion de code à l'aide de directive de type Si... Alors... Sinon... Par exemple, le code suivant :

```
#ifdef __linux__
#define __PAUSE
#else
#define __PAUSE system("pause")
#endif
```

inséré en début de source permet d'écrire :

```
__PAUSE;
```

Lorsque le préprocesseur rencontrera cette ligne il la remplacera par :

- ; si le symbole `__linux__` est défini ;
- `system("pause");` sinon.

Selon l'environnement de développement (compilateur, système d'exploitation) un grand de nombre de symbole sont définies, permettant ainsi d'adapter des portions de code aux spécificités des systèmes et compilateurs. À titre indicatif, sur un système Gnu/Linux Debian avec le compilateur `gcc`, les symboles suivants sont définis (liste non exhaustive) :

```
__ELF__ unix __i386__ linux
__ELF__ __unix__ __i386__ __linux__
__unix __linux
```

Une autre application de la compilation conditionnelle est l'utilisation des messages de debugage, messages que l'on pourra désactiver ou activer selon les circonstances :

```
#ifdef __DEBUG
#define __MESSAGE(m)  printf(m);
#else
#define __MESSAGE(m)
#endif
```

on pourra donc truffer son source d'appels à la macro `__MESSAGE` (par exemple `__MESSAGE("Début_traitement")` puis :

- **#define** __DEBUG activera l'affichage de ces messages ;
- **#undef** __DEBUG les désactivera.

10.1.3 Inclusion de fichiers

La célèbre directive **#include**, lorsqu'elle survient dans un source, est remplacée par le contenu du fichier auquel elle fait référence. Ainsi, lorsque le préprocesseur rencontre :

`#include <stdio.h>`

il remplace cette ligne par le contenu du fichier `stdio.h`. Il y a deux formes de directives :

- **#include** "..." qui inclut le fichier en le cherchant dans le répertoire courant ;
- **#include** <...> qui inclut le fichier le cherchant dans une liste de répertoires prédéfinie dépendant de l'environnement de programmation. Par exemple sur le système de votre serviteur la recherche a lieu dans :
  - `/usr/local/include`
  - `/usr/lib/gcc-lib/i386-linux/2.95.4/include`
  - `/usr/include`

  ce qui sous-entend notamment que le fichier `stdio.h` se trouve dans l'un de ces répertoires.

Enfin, il est important de noter que le mécanismes d'inclusion de fichiers constitue le mécanisme de base pour la compilation séparée (cf. plus loin).

10.2 Compilation

L'étape suivante est la création du code objet à partir du source traité par le préprocesseur. **Attention**, dans ce qui suit, le terme de *compilation* ne désigne que la phase qui crée le *code objet* à partir du source, et non l'*exécutable*. Nous allons examiner ce qui se passe lors de cette phase en s'appuyant sur la figure 2 dans laquelle on dispose d'un source tel que :

- une fonction `f` a été définie et est appelée ;
- la fonction `printf` est appelée

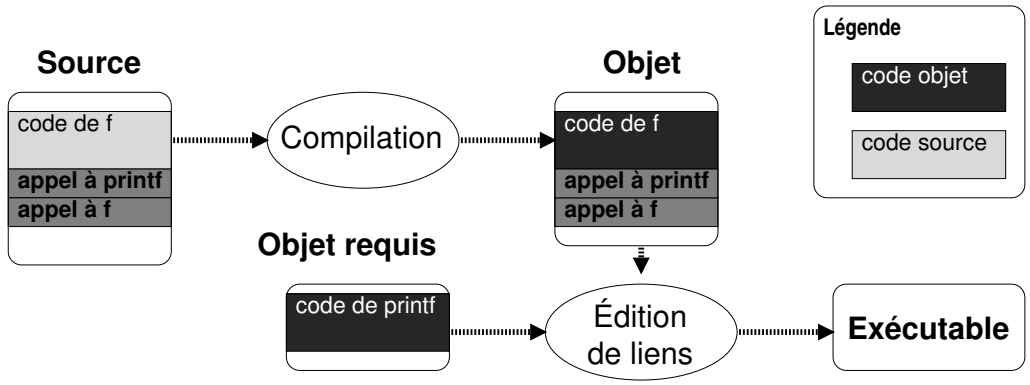


FIGURE 2 – Étude de la compilation sur un exemple.

Dans cet exemple, la phase de compilation créé un code objet contenant le code objet de la fonction `f` et les deux appels à `printf` et `f`. Pour créer l'exécutable, lors que de l'édition de liens, il faudra trouver un moyen d'inclure le code objet de `printf`.

10.2.1 Définition et déclaration de fonction

Supposons maintenant que le source contienne :

```
int main()
{
  double d=f();
  return 0;
}
double f()
{
  return 2.5;
}
```

le compilateur devrait dans ce cas avertir le programmeur avec les messages suivant :

```
comp1.c:3: warning: implicit declaration of function ‘f’
```

Ce premier message signifie que le compilateur ne connaît pas la fonction `f` et qu'il fait l'hypothèse qu'elle existe (implicit declaration) et qu'elle renvoie un entier.

```
comp1.c:7: warning: type mismatch with previous implicit declaration
comp1.c:3: warning: previous implicit declaration of ‘f’
comp1.c:7: warning: ‘f’ was previously implicitly declared to return ‘int’
```

à la ligne où apparaît la fonction `f`, le compilateur rencontre donc une nouvelle déclaration et indique par conséquent qu'elle ne correspond pas avec sa déclaration implicite (**double** au lieu de **int**).

Il est donc très important de comprendre que le comportement d'un compilateur de langage C dans ce cas précis est :

❶ de *supposer* que la fonction inconnue prend un nombre variable d'arguments et qu'elle renvoie un entier ;

❷ de *créer malgré tout* le code objet avec cette hypothèse.

Même si le code objet est créé, l'exécutable correspondant **ne fonctionnera sans doute pas comme il le devrait**. Il est donc important de toujours **déclarer** les fonctions, et ce de la manière suivante :

```
/* déclaration de la fonction f */
double f();

int main()
{
  double d=f();
  return 0;
}
/* définition de la fonction f */
double f()
{
  return 2.5;
}
```

Dans ce source, on distingue :

❶ à la ligne 2, la *déclaration* de la fonction `f` qui permet d'informer le compilateur du type renvoyé et des arguments. Noter l'utilisation ici du point virgule en fin de déclaration, qui consiste à écrire le *prototype* ou *entête* de `f` ;

❷ de la *définition* de la fonction (ligne 11–13) où on rappelle le prototype suivi du *code* de la fonction.

Ceci étant acquis, on peut maintenant comprendre quelle est la signification exacte d'une ligne comme #**include**<stdio.h>. En effet le code suivant :

```
int main()
{
  printf("bonjour\n");
  return 0;
}
```

compilera avec le message suivant :

```
comp2.c:3: warning: implicit declaration of function ‘printf’
```

puisque rien n'informe le compilateur C de l'origine de la fonction `printf`. En revanche le code :

```
#include <stdio.h>
int main()
{
  printf("bonjour\n");
  return 0;
}
```

compilera sans message, puisque le fichier `stdio.h` contient la liste des entêtes ou prototype des fonctions de la librairie d'entrée/sortie et donc celui de `printf`. Il faut donc noter ici, que les fichiers `xxxx.h` :

❶ contiennent l'entête des fonctions d'une bibliothèque, ceci permet à l'aide d'une directive #**include** de les inclure directement dans le source pour informer le compilateur des types des arguments et des valeurs renvoyées par ces fonctions ;

❷ ne contiennent pas de code objet des bibliothèques ; par exemple, sur le système de votre serviteur, le code objet de `printf` se trouve dans un fichier constituant la « librairie C » et se nommant `/lib/libc.so.6` ;

❸ se nomment `xxxx.h` pour rappeler le terme `header` (entête) en anglais.

Pour résumer, on retiendra que :

toutes les fonctions doivent être *déclarées* avant d'être utiliser.

ceci peut se faire comme on l'a vu par l'écriture des entêtes ou prototypes des fonctions utilisées, ce qui se fait généralement par l'intermédiaire de fichiers `.h`.

10.2.2 Compiler plusieurs sources

Tous les projets un tant soit peu complexes en langage C sont répartis sur plusieurs fichiers source. Il faut garder à l'esprit les règles suivantes pour pouvoir gérer un projet comportant plusieurs fichiers :

❶ chaque fichier doit pouvoir se compiler séparément et indépendamment des autres ;

❷ les fonctions et variables qui y sont définies doivent être définies une et une seule fois, et peuvent être déclarées autant de fois que c'est nécessaire.

Dans notre exemple, on pourra écrire la fonction `f` dans un fichier, et la fonction `main` dans un autre. Donc, dans un fichier nommé pour l'exemple `mod.c` :

```
double f()
{
  return 2.5;
}
```

et pour le `main` un fichier `prog.c` :

```
#include <stdio.h> /* pour printf */
double f();        /* déclaration de f */

int main()
{
  printf("f a renvoyé : %f\n", f());
  return 0;
}
```

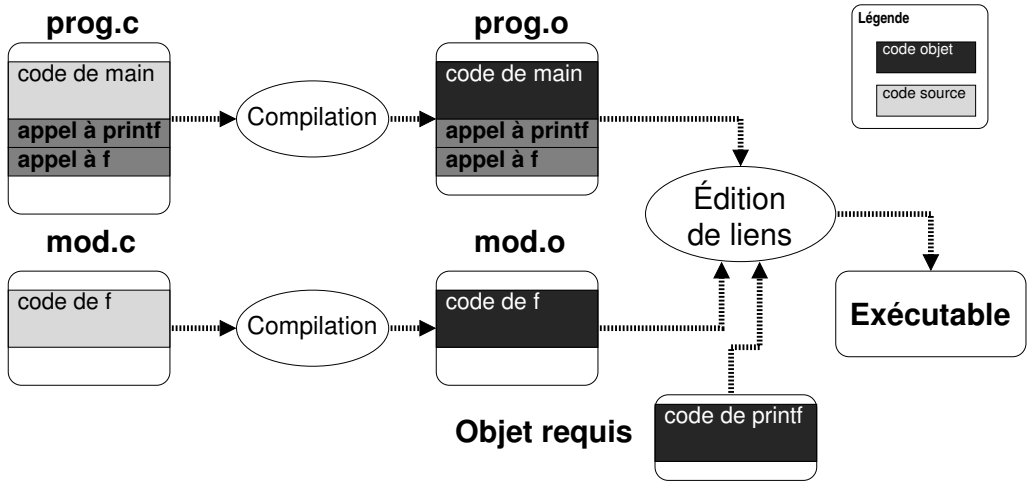


FIGURE 3 – Compilation de plusieurs sources

Notez qu'ici on déclare la fonction `f` et qu'on inclut le fichier `stdio.h` pour avoir le prototype de `printf`. Il faut alors comprendre (cf. fig 3) que l'on va compiler les deux fichiers sources `prog.c` et `mod.c` pour obtenir deux fichiers objets `prog.o` et `mod.o` ; ces deux derniers interviendront dans la phase d'édition de liens.

On comprend aisément que le fichier `mod.c` est destiné à contenir plus d'une fonction. Il n'est bien entendu pas envisageable pour le programmeur d'avoir à écrire les prototypes de toutes ces fonctions pour les utiliser dans un source composant le projet. C'est pourquoi il est habituel d'écrire les prototypes des fonctions à exportées dans un fichier portant l'extension `.h`. Pour notre exemple on créera donc un fichier `mod.h` contenant le prototype de notre fonction :

```
double f();
```

et on ajoutera au début du fichier `prog.c`, la directive :

```
#include "mod.h"
```

Le schéma de la compilation sera alors celui de la figure 4 page suivante.

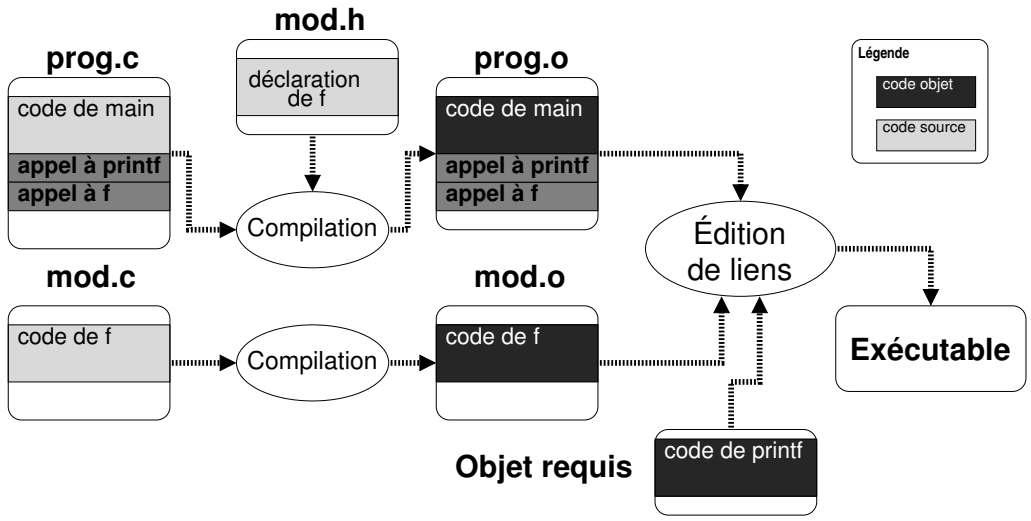


FIGURE 4 – Compilation de plusieurs sources avec fichier d'entête.

10.2.3 Le cas des variables

L'utilisation des variables dans un projet en langage C suit le même principe que les fonctions :

- une variable doit être définie à un seul endroit dans un des sources ;
- elle peut être déclarée plusieurs fois à condition d'utiliser le préfixe **extern**.

On pourra donc imaginer, pour poursuivre notre exemple, que l'on déclare une variable dans le fichier `mod.c`

```
int Globale;
double f()
{
  return 2.5;
}
```

Pour pouvoir y avoir accès dans le fichier `prog.c`, on écrira dans `mod.h` :

```
double f();
extern int Globale; /* la variable est définie ailleurs */
```

On notera d'ailleurs que lorsqu'on déclare une fonction le mot clef **extern** est explicite. Ainsi :

```
double f();
```

est équivalent à :

```
extern double f();
```

10.2.4 Le cas des types

Le meilleur endroit pour définir les types de données est dans les fichiers « include » (le `mod.h` de notre exemple). On pourra donc par exemple écrire dans `mod.h` :

```
double f();
extern int Globale;
typedef struct {
```

```
  double x,y;
} Tpoint;
```

pour que les sources qui incluent `mod.h` puissent utiliser le type `Tpoint`. Il faut cependant noter que dans les situations où le fichier `mod.h` serait inclus plusieurs fois (c'est le cas lorsque plusieurs source procède à des inclusions croisées), le compilateur enverra le message suivant :

```
mod.h: redefinition of 'Tbidule'
```

Dans ce cas on prendra soin d'écrire le fichier `mod.h` de la manière suivante :

```
#ifndef __MOD_H /* on continu si le symbole __MOD_H n'est pas défini */
#define __MOD_H /* on le définit pour d'éventuelles autres includes */
double f();
extern int Globale;
typedef struct {
  double x,y;
} Tpoint;
#endif
```

ce qui permet d'assurer que pour la compilation d'un fichier source, le fichier `mod.h` ne sera inclus qu'une fois. Ce problème est limité à la définition des types et ne se pose pas pour les variables et les fonctions, en effet, un compilateur ne se plaindra pas d'un code du type :

```
double f();
double f();
```

et considérera qu'une seule fonction `f` est déclarée.

10.3 L'édition de liens

Une fois que tous les sources du projets sont compilés, il est nécessaire d'appeler l'éditeur de liens pour assembler tous les fichiers objets et créer l'exécutable.

10.3.1 Résolution des appels

Lors de cette phase, l'éditeur de liens (*linker* en anglais) cherchera à résoudre les différents appels de fonctions qu'il a stockés dans les fichiers objets. Dans notre exemple, le linker fera correspondre :

- l'appel à `f` dans `prog.o` au code trouvé dans `mod.o` ;
- l'appel à `printf` dans `prog.o au code trouvé` dans la bibliothèque standard contenant le code objet de `printf`.

Si lors de l'éditions de liens on oublie l'un des fichiers objet, le linker émettra un message d'erreur. Par exemple, si le fichiers `mod.o` n'intervient dans l'édition de liens, le linker vous informera de la manière suivante :

```
prog.o: dans la fonction 'main':
prog.o(.text+0xa): référence indéfinie vers 'f'
```

en d'autres termes un appel à `f` a été trouvé dans `prog.o` pour lequel ne correspond pas de code objet.

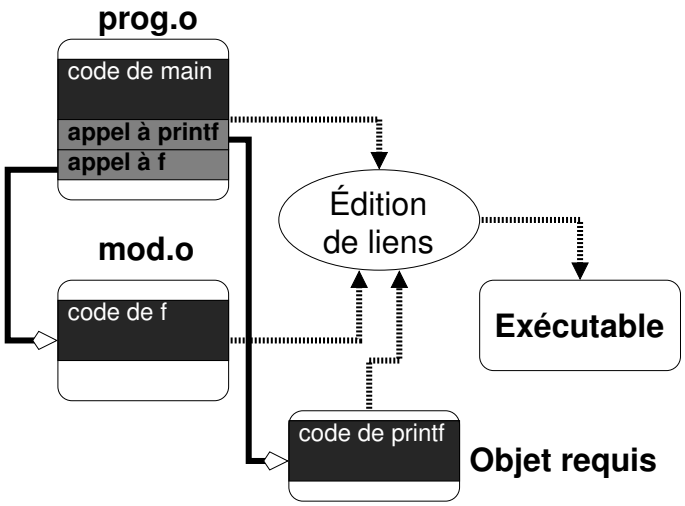


FIGURE 5 – Édition de liens : résolutions des appels

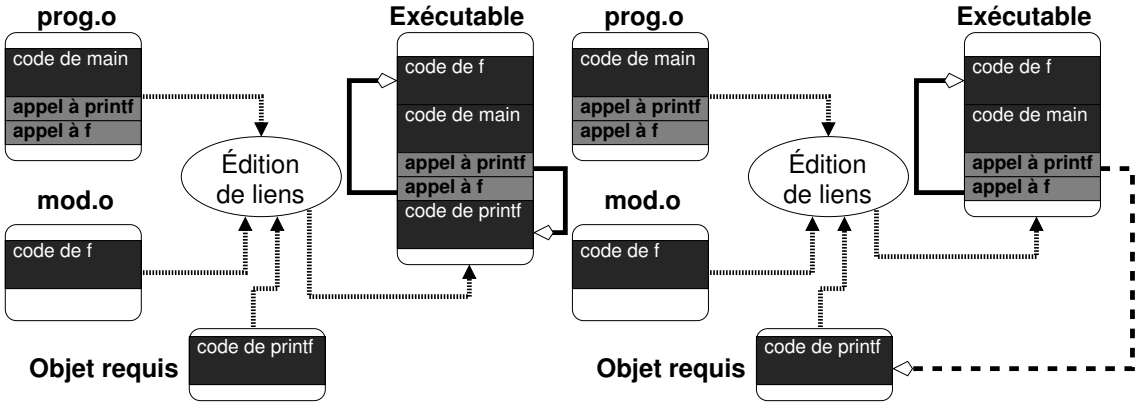
10.3.2 Création de l'exécutable

Une fois tous les appels résolus, le linker pourra créer l'exécutable à proprement parlé. Deux stratégies sont possibles :

❶ L'édition de liens statique, l'intégralité du code objet produit et requis est intégré dans l'exécutable. Ce qui produit un exécutable plus volumineux mais *autonome* ;

❷ L'édition de liens dynamique, le code objet produit est intégré dans l'exécutable, mais le code objet requis reste dans les librairies et est chargé en mémoire au moment de l'exécution. Ce qui implique que l'exécutable ne peut tourner que si le système d'exploitation dispose des bibliothèque où réside le code objet requis. L'avantage est bien sûr la relative petitesse de l'exécutable produit.

L'édition de liens dynamique offre un autre avantage : il n'est pas nécessaire de recréer l'exécutable lorsque le code de la bibliothèque change, puisque l'exécutable ne contient qu'un appel (une référence) à la fonction appelée.



(a) Statique (b) Dynamique

FIGURE 6 – Édition de liens

10.3.3 Fonctions et `static`

Dans certaines situations on peut vouloir imposer que certaines fonctions d'un source ne soient pas visibles pour d'autres sources ; ou en d'autres termes que le code d'une fonction ne soit pas accessible aux autres codes objet. Dans ce cas il suffit de définir la fonction en la préfixant par le mot clef **static** :

```
static int privee(double d);
{
  return 4;
}
```

cette fonction ne pourra être utilisée que dans le source où elle a été définie. Toute tentative d'utilisation ailleurs entraînera un message d'erreur de l'éditeur de liens.

11 Questions fréquemment posées

❒ Que veut dire le message :

```
xxxx.c:3: warning: implicit declaration of function ‘printf’
```

☛ Vous n'avez pas inclus le fichier d'entête de la bibliothèque d'entrée sortie :

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

❒ Que veut dire le message :

```
xxxx.c:4: warning: control reaches end of non-void function
```

☛ Vous avez oublié un `return` dans une fonction qui ne renvoie pas `void`.

❒ Que veut dire le message :

```
/tmp/ccsefeJR.o(.text+0x1a): In function ‘main’:
: undefined reference to ‘sqrt’
```

☛ Vous utilisez une fonction de la bibliothèque mathématique et vous oubliez de l'inclure à l'édition de liens :

`gcc -o bidule truc.c` `-lm`

❒ Pour quoi y a ça dans mon source `bidule.c` :

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@
\203^D^H4^@^@^@\220^]^@^@^@^@^@^@4^@ ^@^G^@(
^@!^@^@^@^F^@^@^@4^@^@^@4\200^D^H4\200^D^Hà^@^
```

☛ Parce que vous avez sauvegardé l'exécutable dans votre source, en tapant quelque chose du genre :

`gcc` `-o bidule.c` `bidule.c`

Ce que vous avez dans `bidule.c` est donc l'exécutable...

Index C

Table des matières