Programmation en langage C ANSI

Alain Crouzil

Table des matières

Chapitre 1

INTRODUCTION

1.1 Historique

Le langage C a été créé par Denis RITCHIE au début des années 1970 dans les laboratoires de recherche de "Bell Telephone" pour écrire le système d'exploitation Unix.

- Première définition du langage : KERNIGHAN et RITCHIE, 1978.
- Normalisations :
  - ANSI (American National Standard Institute), 1989 ;
  - ISO (International Standardization Organization), 1990.

  Ces normes sont identiques et on parle de "C ANSI" ou "C norme ANSI". Une seconde norme ANSI, la norme C99, a été établie en 1999.

1.2 Caractéristiques du langage

Le langage C est suffisamment « près de la machine » (bas niveau) pour permettre d'écrire un système d'exploitation et suffisamment « au dessus » (haut niveau) pour permettre d'écrire des logiciels indépendants de chaque machine particulière.

Le langage C est d'une grande souplesse. Sa syntaxe offre souvent plusieurs possibilités pour effectuer une même action. Un compilateur C ne fera pas autant de vérifications que d'autres compilateurs (par exemple Pascal ou Ada).

⇒ Problèmes :
- de lisibilité
- de mise au point

⇒ Nécessité d'une programmation rigoureuse.

1.3 Exemple de programme

Le programme suivant :

temperature.c

```
1  /* Affichage d'une table de correspondance entre des températures en °F et en °C */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7    int debut,fin,pas;
8    float fahr,celsius;
9
10   debut=0; fin=300; pas=20;
11   fahr=debut;
12   while (fahr<=fin)
13   {
14     celsius=(5.0/9.0)*(fahr-32.0);
15     printf("%4.0f %6.1f\n",fahr,celsius);
16     fahr=fahr+pas;
17   }
18   return(0);
19 }
```

Exemple de programme en langage C ANSI.

produit sur l'écran l'affichage suivant :

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

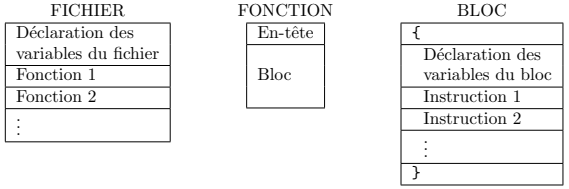
1.4 Quelques remarques

1. Les mots-clés du langage sont en minuscules.
2. Les structures de blocs apparaissent nettement.
3. Le programme principal est une fonction de nom `main`.
4. Pour pouvoir effectuer des entrées-sorties, on doit faire appel à des fonctions de la bibliothèque standard des entrées-sorties (par exemple `printf`). Pour cela, on doit donner au préprocesseur (voir chapitre 9) la directive suivante :
   `#include <stdio.h>`
5. La fonction `main` retourne un entier. Grâce à l'opérateur `return` et en retournant 0, on « précise » au système d'exploitation que tout s'est bien passé.
6. Dans une fonction, on peut faire appel :
   - à des fonctions des bibliothèques standard du langage,
   - à des fonctions de bibliothèques écrites par un programmeur,
   - à des fonctions définies dans le même fichier ou dans d'autres fichiers.

1.5 Structures du langage

La structure la plus importante est la *fonction*. Un programme C est un ensemble de fonctions disjointes dont une, et une seule, porte le nom `main`. Cette fonction est la *fonction principale*, correspondant au *programme principal* d'autres langages, c'est-à-dire qu'elle constitue le point d'entrée du programme, en d'autres termes l'endroit où se trouve la première instruction à exécuter.

Une fonction est structurée en blocs et les fonctions sont regroupées en fichiers. Un programme peut occuper un ou plusieurs fichiers. On a donc la structure à trois niveaux suivante :



Propriétés de ces structures :

- Un bloc peut contenir des blocs
  Exemple :

```
{ /* Début du bloc de niveau 1 */
    int c;
    c=getchar();
    while (c!=EOF)
    { /* Début du bloc de niveau 2 */
        putchar(c);
        c=getchar();
    } /* Fin du bloc de niveau 2 */
{ /* Fin du bloc de niveau 1 */
```

  On préfèrera déclarer toutes les variables d'une fonction au début de son bloc de niveau 1.
- Une définition de fonction ne peut pas contenir d'autres définitions de fonctions. En revanche, on peut faire *appel* à d'autres fonctions déclarées préalablement.
- Un fichier contenant un programme source est en général composé :
  - éventuellement d'une partie de déclaration de variables globales au fichier ;
  - d'une séquence de définitions de fonctions.

Exemple :

puissances.c

```
1  #include <stdio.h>
2
3  /* Fonction de calcul de x à la puissance n */
4  int puissance(int x, int n)
5  {
6    int i,p; /* variables connues dans la fonction puissance */
7
8    p=1; i=1;
9    while (i<=n)
10   {
11     i=i+1;
12     p=p*x;
13   }
14   return p;
15 }
16
```



```
17 /* Fonction principale */
18 int main(void)
19 {
20   int i; /* i est connue dans la fonction main et n'est pas la même variable
21             que celle de la fonction puissance */
22   i=0;
23   while (i<=10)
24   {
25     printf("%d %d\n",i,puissance(2,i));
26     i++; /* équivalent à i=i+1; */
27   }
28   return 0;
29 }
```

Affichage des puissances successives de 2.

Chapitre 2

ÉLÉMENTS DE BASE DU LANGAGE

2.1 Format d'un programme

Un programme se présente comme du texte structuré en lignes (une ligne se termine par le caractère "*new line*"). La « mise en page » des divers composants d'un programme est libre. Ceci doit être utilisé pour écrire des programmes lisibles : indentation des blocs, ...
Les commentaires sont de la forme suivante :
`/* commentaire */`
Ils peuvent apparaître partout où on peut mettre un espace, donc pas à l'intérieur des unités syntaxiques. Par exemple, `ma/*prog...*/in(void)` n'est pas permis.

```
{
    ..........
    ..........
    ..........
    {
        ..........
        ..........
        {
            ..........
            ..........
        }
        ..........
    }
}
```

2.2 Identificateurs

- Un identificateur est une suite d'éléments de l'ensemble suivant :
  a,b,...,z,A,B,...,Z,_,0,1,...9
- Majuscules et minuscules sont différenciées.
- Le premier caractère d'un identificateur ne doit pas être un chiffre.
- Sauf dans de très rares cas, les 31 premiers caractères d'un identificateur sont significatifs.
- Les mots-clés du langage ne peuvent pas être utilisés en tant qu'identificateurs. Le tableau 2.1 présente les 32 mots-clés du langage C.

Exemples : `3AZ` n'est pas un identificateur valide.
`alpha` et `Alpha` sont deux identificateurs différents.
`while` n'est pas un identificateur valide.

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

TAB. 2.1 – *Mots-clés du langage C.*

2.3 Constantes

La représentation interne des constantes (et des variables) dépend de la machine et du compilateur utilisé. Elle est paramétrée par :
- la taille de la représentation (1, 2, 4, ... octets),
- la technique de codage des nombres (complément à 2, virgule flottante, ...),
- le codage des caractères (ASCII).

On distingue quatre types de constantes : nombre entier, nombre réel, caractère et chaîne de caractères.

2.3.1 Constantes entières

On peut utiliser la représentation en base 10, 8 ou 16 :
- en décimal (base 10) : `14`, `-5`, `65535`
- en octal (base 8), on fait précéder le nombre du chiffre `0` : `016`, `0177777` (attention : `016` $\neq$ `16`)
- en hexadécimal (base 16), on fait précéder le nombre de `0x` (ou `0X`) : `0xe`, `0XFFFF`

2.3.2 Constantes réelles

On peut utiliser :
- la notation décimale : `3.141`
- la notation « scientifique » : `0.3141e+1` ou `0.3141E+1`
  `0.03141e+2` ou `0.03141E+2`
  `31.41e-1` ou `31.41E-1`

2.3.3 Constantes caractères

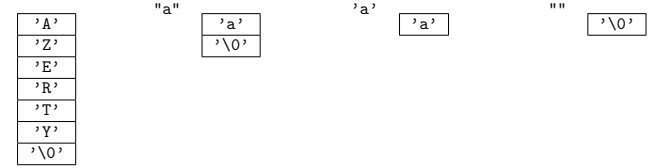
- Caractères « imprimables » (ou « éditables ») : `'A'`, `'4'`, `'O'`
- Caractères disposant d'une « séquence d'échappement » (notation symbolique) :

Notation	Dénomination	Notation	Dénomination	Notation	Dénomination
`'\n'`	saut de ligne (*new line*)	`'\f'`	saut de page	`'\?'`	point d'interrogation
`'\t'`	tabulation	`'\\'`	anti-slash	`'\a'`	bip
`'\b'`	retour arrière (*backspace*)	`'\''`	apostrophe	`'\v'`	tabulation verticale
`'\r'`	retour chariot	`'\"'`	guillemets		

- Désignation d'un caractère par son code (anti-slash suivi du code ASCII du caractère en octal ou en hexadécimal) : `'\x41'` (`'A'`), `'\101'` (`'A'`), `'\0'` (caractère « nul »).

2.3.4 Constantes chaînes de caractères

Elles sont placées entre guillemets (`"`). Elles peuvent contenir des caractères en notation éditable, octale, hexadécimale ou symbolique :
`"AZERTY"` est une chaîne de 6 caractères,
`"a3(\n"` est une chaîne de 4 caractères,
`"a3(\\n"` est une chaîne de 5 caractères,
`"bonjour\nmonsieur"` est une chaîne de 16 caractères,
`"ab\"cd\""` est une chaîne de 6 caractères.
Important : en mémoire, une constante chaîne de caractères est complétée par le caractère `'\0'` :



2.4 Variables et types

2.4.1 Déclaration

Toutes les variables utilisées dans un programme C doivent avoir été déclarées avant leur utilisation. Une déclaration associe une liste d'identificateurs de variables à un type :

type `ident`$_1$`,`*ident*$_2$`,...,`*ident*$_n$`;`

2.4.2 Types de base

Types caractères

Pour représenter un caractère, on utilise le type `char`. Pour représenter des petits entiers, on utilise, selon le cas, `unsigned char` ou `signed char`.

Exemple :

```
char alpha,beta;
unsigned char pixel;
char c;
c=':';
```

La représentation mémoire de la variable `c` est :

0	0	1	1	1	1	0	0

c'est-à-dire : $(+60)_{10}$.

Remarque : une variable de type caractère peut être utilisée dans une expression arithmétique.

Types entiers

Il existe six types d'entiers qui se différencient par :

- un paramètre de signe facultatif : `signed` ou `unsigned` ;
- un paramètre de taille facultatif : `short` ou `long`.

Pour choisir entre ces types, on se demandera si l'on veut manipuler des entiers naturels ou relatifs et de quel domaine les entiers en question ont besoin.

- Entier court signé : `signed short int` ou `short`.
- Entier court non signé : `unsigned short int` ou `unsigned short`.
- Entier signé : `signed int` ou `int`.
- Entier non signé : `unsigned int` ou `unsigned`.
- Entier long signé : `signed long int` ou `long`.
- Entier long non signé : `unsigned long int` ou `unsigned long`.

Les tailles minimales des types entiers imposées par la norme sont données dans la tableau 2.2. La norme impose en outre la règle suivante :

$$taille(\text{entiers courts}) \leq taille(\text{entiers}) \leq taille(\text{entiers longs}).$$

Les entiers non signés sont représentés en binaire pur et, généralement, les entiers signés sont représentés en complément à 2.

Types flottants

Il s'agit des types qui permettent de représenter les réels. Les trois types de flottants se distinguent par le domaine et la précision. Une précision égale à n signifie que tout entier d'au plus n chiffres s'exprime sans erreur en flottant.

- Flottant : `float`.
- Flottant double précision : `double`.
- Flottant long double précision : `long double`.

Le domaine et les précisions des types flottants sont donnés dans la tableau 2.2.

2.4.3 Initialisation

Lors de leur déclaration, on peut initialiser les variables à une certaine valeur spécifiée par une expression (qui doit être compatible avec le type de la variable). Une variable qui n'a pas été initialisée a une valeur indéterminée jusqu'à ce qu'on lui affecte une valeur.

Dénomination	Types	Taille minimale (en octets)	Domaine minimal
Caractère non signé	`unsigned char` `char` (*)	1 exactement	[0..255] exactement
Caractère signé	`signed char` `char` (*)	1 exactement	$[-127..+127]$

Entier court signé	`short`
	`short int`
	`signed short`
	`signed short int`
		2	$[-32767..+32767]$

Entier court non signé	`unsigned short` `unsigned short int`	2	[0..65535]

Entier signé	`int`
	`signed int`
	`signed`
		2	$[-32767..+32767]$

Entier non signé	`unsigned int` `unsigned`	2	[0..65535]

Entier long signé	`long`
	`long int`
	`signed long`
	`signed long int`
		4	$[-2147483647..+2147483647]$

Entier long non signé	`unsigned long` `unsigned long int`	4	[0..4294967295]
Flottant	`float`	–	$[-10^{+37}..-10^{-37}]+[10^{-37}..10^{+37}]$ Précision minimale : 6
Flottant double précision	`double`	–	$[-10^{+37}..-10^{-37}]+[10^{-37}..10^{+37}]$ Précision minimale : 10
Flottant long double précision	`long double`	–	$[-10^{+37}..-10^{-37}]+[10^{-37}..10^{+37}]$ Précision minimale : 10

(*) Ça dépend de l'implémentation.

Tab. 2.2 – *Taille des types de base.*

Exemples :

```
int Somme=0;
int delai=12*60;
int a,b=0; /* attention : a n'est pas initialisée */
char c='X';
```

2.4.4 Tableaux

Un tableau est un ensemble d'éléments de même type désignés par un identificateur unique. L'accès à chaque élément se fait par l'intermédiaire d'un indice qui désigne sa position au sein de l'ensemble.

Tableaux à une dimension

- **Déclaration**

type_element nom_tableau `[`*nombre_d_elements*`];`

Exemple : `int tab[10];`
déclare un tableau `tab` comportant 10 éléments de type `int`.
Remarque : les éléments d'un tableau sont toujours rangés à la suite les uns des autres en mémoire.

- **Accès à un élément**
  nom_tableau `[`*indice*`]`
  Attention : en C, le premier élément est toujours celui d'indice 0.
  Exemples : `tab[0]` désigne le premier élément du tableau `tab` et `tab[9]` désigne, dans notre exemple, le dernier élément du tableau `tab`.
- **Initialisation**
  Il est possible d'initialiser un tableau au moment de sa déclaration.
  Exemple : `int t[3]={1,2,3};` déclare un tableau de trois `int` et place les valeurs 1, 2 et 3 dans les trois cases de `t`.
  Attention : ce genre d'affectation « globale » du contenu d'un tableau n'est possible qu'au moment de sa déclaration.

Tableaux à deux dimensions

- **Déclaration**
  type_element nom_tableau `[`*nombre_de_lignes*`][`*nombre_de_colonnes*`];`
  Exemple : `int t[2][3];`
  déclare un tableau `t` comportant six éléments de type `int`.
- **Accès à un élément**
  nom_tableau `[`$indice_1$`][`$indice_2$`]`
  Exemple : `t[1][2]` désigne l'élément du tableau `t` qui se trouve sur la ligne 1 et la colonne 2.
- **Arrangement en mémoire**
  Les éléments d'un tableau sont rangés en mémoire « ligne par ligne ».
  Exemple :
  ```
  t[0][0]
  t[0][1]
  t[0][2]
  t[1][0]
  t[1][1]
  t[1][2]
  ```
- **Initialisation**
  Exemple : `int t[2][3]={{1,2,3},{4,5,6}};`

2.4.5 Structures

Une structure permet de déclarer des variables composées d'un ensemble de données (champs) pouvant être de types différents.

- **Définition et déclaration**
  Définition d'une structure :
  ```
  struct identificateur
  {
    déclaration_des_champs
  };
  ```
  Exemple de définition d'une structure :
  ```
  struct point
  {
    int numero;
    float x;
    float y;
  };
  ```
  Exemple de déclaration de variables structurées : `struct point pt1,pt2;`

- **Accès aux champs d'une variable structurée**
  identificateur_de_variable.identificateur_de_champ
  Exemples :
  ```
  pt1.numero=1;
  pt1.x=0.5;
  pt1.y=12.3;
  printf("%f\n",pt1.x);
  ```
- **Initialisation**
  On peut initialiser une variable structurée au moment de sa déclaration.
  Exemple : `struct point pt={12, 5.35, 10.4};`
- **Affectation « globale »**
  Contrairement aux tableaux, on peut affecter « d'un seul coup » le contenu d'une variable structurée à une autre variable structurée du même type.
  Exemple : `pt2=pt1;`

2.4.6 Type énuméré

Le type énuméré est un cas particulier de type entier. Il permet de remplacer les constantes symboliques.
Exemples :

- `enum couleur {rouge, vert, bleu};`
  définit un type `enum couleur` comportant trois valeurs possibles désignées par les constantes `rouge`, `vert` et `bleu`, qui sont des entiers ordinaires (ce qui implique une très grande souplesse d'utilisation...) valant respectivement 0, 1 et 2.
- avec `enum couleur {rouge=2, vert=4, bleu};`
  les trois constantes valent respectivement 2, 4 et 5.
- `enum couleur c1,c2;`
  déclare deux variables `c1` et `c2` de type `enum couleur`. On peut alors écrire :
  ```
  c1=rouge;
  printf("%d\n",c1);
  c2=c1;
  ```
  En revanche, il n'est pas possible d'écrire : `rouge=3;`

2.4.7 Définition de synonymes de types

L'instruction `typedef` permet de définir des synonymes de types, c'est-à-dire de donner un nom à un type quelconque (sans créer de nouveau type), aussi complexe soit-il, puis d'utiliser ce nom comme spécificateur de type pour simplifier les déclarations.
Exemples :

- ```
  typedef int Entier;
  int a,b;  ⇔  Entier a,b;
  ```
- ```
  typedef int *Ptr;
  int *p1,*p2;  ⇔  Ptr p1,p2;
  ```
- ```
  typedef double Point[3];
  double p[3],q[3];  ⇔  Point p,q;
  ```
- ```
  typedef struct
          {
            char Nom[30];
            char Appelation[20];
            int Millesime;
            int Quantite;
          } Vin;
  Vin v1,v2;
  ```

- ```
  typedef enum {faux=0, vrai=1} Booleen;
  Booleen Indic;
  Indic=vrai;
  ...
  if (Indic)
  ...
  ```

2.5 Opérateurs et expressions

Une *expression* est constituée d'*opérandes* et d'*opérateurs*. En C, toute donnée placée en mémoire centrale est considérée comme une valeur numérique.

⇒ Un caractère peut être utilisé dans une expression arithmétique. Le code du caractère est alors considéré comme un `int`.

⇒ Une expression logique (« booléenne ») est considérée comme un entier valant 0 (faux) ou 1 (vrai). Par exemple, `(7>3)` vaut 1.

⇒ Des conversions implicites de types peuvent avoir lieu. Une « hiérarchie » entre les types est définie :

`char` → `int` → `long` → `float` → `double` → `long double`

Exemple : si l'on déclare :
`int n; double x;`
dans l'expression `n+x`, `n` est convertie en `double` et le résultat est de type `double`.

2.5.1 Opérateurs d'affectation

Affectation simple

L'opérateur d'affectation est le caractère `=`.

Exemple :
```
int i;
i=12; /* la variable i reçoit la valeur 12 */
```

Opérateurs d'incrémentation et de décrémentation

Désignation	Expression	Effet	Exemple
Post-incrémentation	*variable*`++`	La valeur de *variable* est augmentée de 1. L'expression vaut l'ancienne valeur de *variable*.	`i=1; a=i++;` `a` vaut 1 et `i` vaut 2
Pré-incrémentation	`++`*variable*	La valeur de *variable* est augmentée de 1. L'expression vaut la nouvelle valeur de *variable*.	`i=1; a=++i;` `a` vaut 2 et `i` vaut 2
Post-décrémentation	*variable*`--`	La valeur de *variable* est diminuée de 1. L'expression vaut l'ancienne valeur de *variable*.	`i=1; a=i--;` `a` vaut 1 et `i` vaut 0
Pré-décrémentation	`--`*variable*	La valeur de *variable* est diminuée de 1. L'expression vaut la nouvelle valeur de *variable*.	`i=1; a=--i;` `a` vaut 0 et `i` vaut 0

Affectations multiples

L'évaluation est effectuée de la droite vers la gauche.

```
int i,j,k;
i=j=k=0;  ⇔  k=0; j=0; i=0;
```

Affectations élargies

On peut condenser

variable = *variable opérateur expression*

en

variable opérateur= *expression*

Les opérateurs d'affectation élargie sont : `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `~=`, `<<=` et `>>=`.

Exemples : `x+=5;` ⇔ `x=x+5;`
`x*=x;` ⇔ `x=x*x;`

2.5.2 Opérateurs arithmétiques

Les opérateurs arithmétiques sont : `+`, `-`, `*`, `/` et `%`. Ils sont tous applicables sur des entiers ou des flottants sauf `%` (reste de la division entière) qui n'est applicable que sur des entiers. Le groupe (`%`,`*`,`/`) est prioritaire sur le groupe (`+`,`-`) et, au sein de chaque groupe, l'évaluation se fait de la gauche vers la droite. Le tableau 2.3 résume les priorités de tous les opérateurs.

2.5.3 Opérateurs relationnels

Les opérateurs relationnels sont : `==`, `!=`, `>`, `<`, `>=`, `<=`.

2.5.4 Opérateurs logiques

Les opérateurs logiques sont : `!` (NON logique), `&&` (ET logique), `||` (OU logique).

2.5.5 Expressions logiques

Une expression logique vaut 0 si elle est fausse ou 1 si elle est vraie. Une expression est considérée comme fausse si elle vaut 0, vraie sinon ($\neq 0$).

Exemples : `x=7;`
`!((x!=2) && (5>=0))` vaut 0 (faux)
`(-5) && (x!=2)` vaut 1 (vrai)
`(24!=1) || (5<0)` vaut 1 (vrai)
`(7>2) && 5` vaut 1 (vrai)
`!3` vaut 0 (faux)

Remarque : les opérateurs `&&` et `||` impliquent une évaluation de la gauche vers la droite optimisée (à l'économie), c'est-à-dire que l'évaluation s'arrête dès que la décision peut être prise.

Exemple : dans les expressions suivantes, le second opérande n'est pas évalué :
`(12>3) || ((3+4)<10)` `(3>4) && (4>2)`

⇒ Attention lorsque ces expressions contiennent des instructions (par exemple des affections ou des appels à des fonctions).

2.5.6 Opérateur conditionnel

expression$_1$ `?` *expression*$_2$ `:` *expression*$_3$

vaut $\begin{cases} \textit{expression}_2 \text{ si } \textit{expression}_1 \neq 0 \text{ (vraie)} \\ \textit{expression}_3 \text{ si } \textit{expression}_1 \text{ vaut 0 (fausse)} \end{cases}$

Exemples :
```
x=n>0?n:-n;  /* valeur absolue */
x=a>b?a:b;  /* max */
```

2.5.7 Opérateurs de manipulation de bits

Ils sont applicables sur les types entiers et de préférence avec des entiers non signés (`unsigned char`, `unsigned int`, ...).

`&`	ET logique bit à bit	`^`	OU EXCLUSIF bit à bit	`<<`	décalage vers la gauche
`|`	OU logique bit à bit	`~`	complément à 1	`>>`	décalage vers la droite
```

Exemples : `unsigned int i,n;`

- Forcer la valeur d'un bit de position variable :
  - Forcer le bit de rang[1] i à 1 : `n=n|(1u<<i);`
  - Forcer le bit de rang i à 0 : `n=n&~(1u<<i);`
- Connaître la valeur d'un bit de position variable : `(n>>i)&1u` ou `(n&(1u<<i))>>i`

2.5.8 Opérateur séquentiel

expression$_1$`,`*expression*$_2$

L'évaluation est effectuée de la gauche vers la droite (*expression*$_1$ puis *expression*$_2$) et l'expression complète vaut alors la valeur de la dernière expression évaluée (*expression*$_2$).
Exemple : après la séquence suivante : `int a,b=0,c=2; a=(b++,c++);`
`b` vaut 1, `c` vaut 3 et `a` vaut 2. Cette écriture est déconseillée...

2.5.9 Conversions forcées de type

Il est possible de forcer la conversion d'une expression dans un type donné grâce à l'opérateur de *cast* :

`(`*type*`)`

Exemple : `int a=7,b=2;`
L'expression `(double)(a/b)` est de type `double` et vaut 3.0. L'expression `(double)a/b` est aussi de type `double` et vaut 3.5 (`a` est d'abord convertie en `double`, puis la division est effectuée en `double`).

L'affectation permet aussi de forcer la conversion d'une expression.

Exemple :
```
int a=7,b=2; double x;
x=a/b; /* x vaut 3.0 */
a=x/b; /* a vaut 1 */
```

Attention : la dernière affectation donne un résultat satisfaisant car la partie entière de `x/b` est représentable dans le type `int`. De manière générale, il faut être prudent quand on effectue des conversions dans « le mauvais sens » de la hiérarchie des types.

2.5.10 Priorités des opérateurs

Le tableau 2.3 montre les priorités par ordre décroissant des différents opérateurs (certains seront présentés plus loin) ainsi que les sens d'évaluation (gauche ⟶ droite ou gauche ⟵ droite).

Catégorie	Opérateurs	Sens d'évaluation (associativité)
Référence	`()  []  ->  .`	⟶
Unaires	`+  -  ++  --  !  ~  *  &  (`*cast*`)  sizeof`	⟵
Arithmétiques	`*  /  %`	⟶
Arithmétiques	`+  -`	⟶
Décalage	`<<  >>`	⟶
Relationnels	`<  <=  >  >=`	⟶
Manipulation de bits	`&`	⟶
Manipulation de bits	`^`	⟶
Manipulation de bits	`|`	⟶
Logique	`&&`	⟶
Logique	`||`	⟶
Conditionnel	`? :`	⟵
Affectation	`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	⟵
Séquentiel	`,`	⟶

TAB. 2.3 – *Priorités décroissantes des opérateurs.*

1. On considère que le rang du bit de plus faible poids est 0.

2.6 Instructions

On ne présente ici que les instructions les plus élémentaires.
Le point-virgule (`;`) est le *terminateur d'instruction*.
Dans la suite, on adopte les conventions suivantes :

- *inst* désigne une instruction simple ou un bloc d'instructions ;
- *expr* désigne une expression ;
- les crochets ([ ]) entourent une partie facultative.

2.6.1 Affectation

variable `=` *expression*`;`

Exemple : `y=3*x-1;`

2.6.2 Choix

`if (`*expression*`)` *inst*$_1$ [`else` *inst*$_2$]

Exemples :
```
if (x==3) printf("OK\n");
if (a>b) m=a; else m=b;
```

Choix en cascade :

Exemple :
```
if (prix<100) tranche=1;
else if (prix<200) tranche=2;
     else if (prix<300) tranche=3;
          else tranche=4;
```

Attention aux imbrications : un `else` se rapporte au dernier `if` rencontré dans le même bloc auquel un `else` n'a pas encore été attribué.

Exemple : la séquence suivante :
```
int a=1,b=3,c=2;
if (a<b)
  if (b<c) printf("a<b<c\n");
else printf("a>=b\n");
```
va provoquer l'affichage de : `a>=b` !

Mais la séquence suivante :
```
int a=1,b=3,c=2;
if (a<b)
{
  if (b<c) printf("a<b<c\n");
}
else printf("a>=b\n");
```
ne va provoquer aucun affichage.

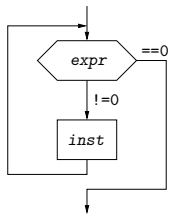
Attention aux opérateurs
`if (r=n%d) ...` peut se « traduire » par : si `r` (qui reçoit `n` *modulo* `d`) est différent de `0` ...
`if (r==n%d) ...` peut se « traduire » par : si `r` est égal à `n` *modulo* `d` ...

2.6.3 Répétitions

Boucle `while`

`while (`*expr*`)` *inst*

Tant que *expr* est différente de 0, *inst* est exécutée (si *expr* vaut 0 au départ, *inst* n'est pas exécutée).



Exemple : la séquence suivante :
```
a=0;
while (a<6)
{
  printf("%d ",a);
  a+=2;
}
```
provoque l'affichage de `0 2 4` . Mais si `a` est initialisée à `6`, on n'obtient aucun affichage.

Boucle `do while`

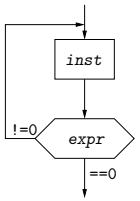
```
do inst while (expr);
```

Ici le test est effectué à la fin de chaque itération. Cela entraîne que *inst* est exécutée au moins une fois.

Exemple : la séquence suivante :

```
a=0;
do
{
  printf("%d ",a);
  a+=2;
} while (a<6);
```

provoque l'affichage de `0 2 4` . Mais si `a` est initialisée à `6`, on obtient l'affichage de `6` .



Remarques :

```
do inst while (expr);          while (expr) inst
        ⇕                              ⇕
inst                           if (expr)
while (expr) inst                do inst while (expr);
```

Boucle `for`

```
for ([expr1];[expr2]; [expr3]) inst
```

La boucle `for` peut se réécrire à l'aide de la boucle `while` :

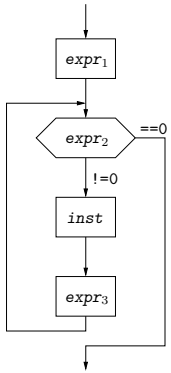
```
expr1;
while (expr2)
{
  inst
  expr3;
}
```

Exemple :

```
for (i=0;i<10;i++)
{
  s+=tab[i];
  p*=tab[i];
}
```

Exemple de boucle utilisant l'opérateur séquentiel :

```
for (i=0,j=9;i<10;i++,j--)          i=0;j=9;
{                                   while (i<10)
  ...                          ⟺   {
}                                     ...
                                      i++;
                                      j--;
                                    }
```



Remarques :

- Si *expr*$_2$ est absente, elle est « remplacée » par `1` ⇒ condition vraie.
- boucle infinie : `for(;;)` ou bien `while(1)`

2.6.4 Choix multiple

```
switch (expr)
{
  case expr_cte1 : [séq_inst1]
  case expr_cte2 : [séq_inst2]
  ⋮
  case expr_cten : [séq_instn]
  [default : séq_inst]
}
```

expr : expression de type entier (`char`, `short`, `int` ou `long`) signé ou pas.

expr_cte$_i$: expression constante entière.

séq_inst$_i$: séquence d'instructions quelconques.

Si *expr* vaut *expr_cte*$_i$ alors on exécute *séq_inst*$_i$, puis *séq_inst*$_{i+1}$, ..., puis *séq_inst*. Cette exécution peut être interrompue par l'instruction `break;`. Si aucune des n *expr_cte*$_i$ ne correspond à la valeur de *expr*, seule *séq_inst* est exécutée (si elle est présente).

Exemple :

```
switch (n)
{
  case 0 : printf("Nul\n");
  case 1 :
  case 2 : printf("Petit\n");
           break;
  case 3 : printf("Moyen\n");
           break;
  case 4 :
  case 5 : printf("Grand\n");
           break;
  default : printf("Énorme\n");
            break; /* pas utile */
}
```

À l'exécution :

Valeur de `n`	Affichage
0	`Nul` `Petit`
1	`Petit`
2	`Petit`
3	`Moyen`
4	`Grand`
5	`Grand`
6	`Énorme`

2.6.5 Compléments sur les boucles

Instruction `break`

Elle permet de sortir de l'instruction `switch` ou de la boucle (`for`, `while` ou `do while`) la plus interne.

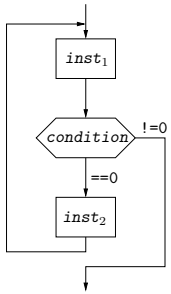
Instruction `continue`

Elle permet de forcer le passage au tour suivant de la boucle la plus interne en se branchant à la fin du corps de cette boucle (dans une boucle `for`, *expr*$_3$ est exécutée).

Exemples d'utilisation

• **Boucle à sortie intermédiaire :**

```
while (1)                    fini=0;
{                     ⟺     do
  inst1                      {
  if (condition) break;        inst1
  inst2                        if (condition) fini=1;
}                              else { inst2 }
                             } while (!fini);
```

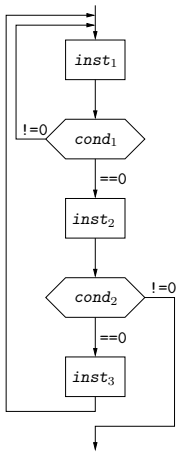


Exemple :

```
while (1)
{
  printf("Donner un entier positif ou -1 pour terminer :\n");
  scanf("%d",&n);
  if (n==-1) break;
  /* Traitement normal de n */
  ...
}
/* Fin */
...
```

• **Boucle à sortie intermédiaire et « rebouclage » :**

```
while (1)                    fini=0;
{                     ⟺     do
  inst1                      {
  if (cond1) continue;         inst1
  inst2                        if (!cond1)
  if (cond2) break;            {
  inst3                          inst2
}                                if (cond2) fini=1;
                                 else { inst3 }
                               }
                             } while (!fini);
```



Exemple :

```
while (1)
{
  printf("Donner un entier positif ou -1 pour terminer :\n");
  scanf("%d",&n);
  if (n<-1) continue;
  if (n==-1) break;
  /* Traitement normal de n */
  ...
}
/* Fin */
...
```

Chapitre 3

FONCTIONS

3.1 Introduction

La fonction est la seule sorte de sous-programme existant en C. Mais elle joue un rôle très général regroupant celui des fonctions et des procédures d'autres langages.

3.2 Définition d'une fonction

```
type_retourné nom_fonction(déclaration_paramètres)
{
  [déclaration_variables_locales]
  instructions
}
```

3.3 Valeur de retour

- Une fonction peut retourner une valeur de n'importe quel type sauf de type tableau.
- Si une fonction ne retourne aucune valeur, son *type_retourné* est `void`.
- Si l'on omet le *type_retourné*, le compilateur considère qu'il s'agit du type `int`.
- Dans une fonction, l'instruction :

  `return` *expression*`;`

  permet :
  - de préciser la valeur de retour de la fonction (la valeur de *expression*),
  - d'interrompre l'exécution de la fonction, c'est-à-dire de « redonner la main » à l'appelant.
- La valeur de retour d'une fonction peut être ignorée par l'appelant. Pour éviter des messages d'avertissement produits par des utilitaires de mise au point comme `lint`, on peut préciser que l'on est conscient de cette ignorance en faisant précéder l'appel de la fonction de `(void)`. *Exemple :* `(void)printf("OK\n");`

3.4 Paramètres

Une déclaration de paramètres est de la forme:

$type_1\ ident_1, type_2\ ident_2, \ldots, type_n\ ident_n$

Si une fonction n'a aucun paramètre, cette déclaration se réduit à `void`.
Attention : le passage des paramètres ne se fait que par valeur. Une fonction a trois possibilités directes pour transmettre des informations à l'appelant :

- par la valeur de retour,
- par l'intermédiaire de variables globales,
- par ses paramètres, en utilisant les pointeurs (voir le chapitre 5).

3.5 Déclaration d'une fonction

Une *déclaration* de fonction peut prendre les formes suivantes :

$type\_retourné\ nom\_fonction(type_1\ ident_1, type_2\ ident_2, \ldots, type_n\ ident_n)$

ou bien

$type\_retourné\ nom\_fonction(type_1, type_2, \ldots, type_n)$

Remarques :

- Si une fonction est utilisée sans avoir été ni déclarée, ni définie au préalable, le compilateur considère qu'elle retourne une valeur de type `int`.
- Dans tous les cas, pour que l'éditeur de liens puisse produire le fichier exécutable, la définition d'une fonction doit être effectuée quelque part, soit dans le même fichier, soit dans un autre fichier (il faut alors le préciser lors de l'édition de liens).

3.6 Utilisation des fonctions

Une fonction peut être appelée si elle a été préalablement *définie* ou bien *déclarée*.

- **Après avoir été définie :**

```
/* Définition de f1 */
... f1(...)
{
  ...
  ...
}

/* Définition de f2 */
... f2(...)
{
  ...
  ...f1(...); /* Appel de f1 */
  ...
}
```

- **Après avoir été déclarée :**

```
/* Déclaration globale de f1 */
... f1(...); /* Elle peut être
utilisée partout dans la suite */

/* Définition de f2 */
... f2(...)
{
  ...
  ...f1(...); /* Appel de f1 */
  ...
}

/* Définition de f1 */
... f1(...)
{
  ...
  ...
}
```

ou bien

```
/* Définition de f2 */
... f2(...)
{
  /* Partie déclarative de f2 */
  ...
  /* Déclaration locale de f1 */
  ... f1(...); /* Elle ne peut
  être utilisée que dans f2 */
  ...

  ...
  ...f1(...); /* Appel de f1 */
  ...
}

/* Définition de f1 */
... f1(...)
{
  ...
  ...
}
```

Chapitre 4

ENTRÉES-SORTIES

4.1 Introduction

Les fonctions d'entrées-sorties font partie de la bibliothèque standard des entrées-sorties. Pour pouvoir les utiliser, il faut insérer la directive `#include <stdio.h>`.

Trois unités standard sont définies :

- l'entrée standard (`stdin`) : par défaut, le clavier ;
- la sortie standard (`stdout`) : par défaut, l'écran ;
- la sortie standard des erreurs (`stderr`) : par défaut, l'écran.

L'entrée est ouverte en lecture et les sorties sont ouvertes en écriture.

En C ANSI, les entrées-sorties sont « bufferisées » (voir paragraphe 4.7).

4.2 Écritures sur `stdout`

4.2.1 Écriture d'un caractère sur `stdout`

`int putchar(int c)`

affiche sur `stdout` le caractère passé en paramètre et retourne ce caractère ou la constante symbolique `EOF` en cas de problème.

Exemple : `char c;  c='a'; putchar(c);`

4.2.2 Écriture formatée sur `stdout`

`int printf(const char *format[,liste_d_expressions])`

convertit, met en forme et affiche ses paramètres sur `stdout` sous le contrôle de la chaîne de caractères *`format`*. Cette fonction retourne le nombre de caractères écrits ou un nombre négatif en cas de problème. La chaîne *`format`* peut contenir :

- des caractères ordinaires qui sont simplement recopiés sur `stdout` ;
- des spécifications de conversions.

Principaux spécificateurs de format

`%c` : `char`
`%d` : `int` ou `char` en décimal (`%ld` pour `long int`)
`%o` : `int` ou `char` en octal (`%lo` pour `long int`)
`%x` : `int` ou `char` en hexadécimal (`%lx` pour `long int`)
`%f` : `float` ou `double` en virgule flottante (*ex :* 35.43)
`%e` : `float` ou `double` en virgule fixe (*ex :* 3.54300e+01)
`%p` : adresse mémoire (en hexadécimal)

Exemples :

1. `int␣a=75;`
   `printf("adec=%d,␣aoct=%o,␣ahex=%x,␣acar=%c\n",a,a,a,a);`
   affiche sur `stdout` :
   `adec=75,␣aoct=113,␣ahex=4b,␣acar=K`
2. `long int i=2;`
   `float x=1.23;`

Instruction	Sortie (`stdout`)

`printf("%ld\nREEL=␣%f\n",i,x);`	`2↩`
	`REEL=␣1.230000↩`
	(par défaut, 6 chiffres après le .)

`printf("%4ld\nREEL=␣%5.1f\n",i,x);` (`4`, `5` : nombre minimum de caractères) (`1` : nombre de chiffres après le .)	`␣␣␣2↩` `REEL=␣␣␣1.2↩`
`printf("%-4ld\nREEL=␣%-5.1f\n",i,x);` (`-` : cadrage à gauche)	`2␣␣␣↩` `REEL=␣1.2␣␣↩`

`printf("%+ld\nREEL=␣%+-10.1e\n",i,x);`
(`+` : affichage du signe même pour les positifs)
	`+2↩`
	`REEL=␣+1.2e+00␣␣↩`
	(le signe compte dans le nombre de caractères)

4.3 Lectures sur `stdin`

4.3.1 Lecture d'un caractère sur `stdin`

`int getchar(void)`

retourne le premier caractère disponible sur `stdin` ou la constante symbolique `EOF` (définie à -1 dans `stdio.h`) si la fin de fichier est atteinte (au clavier : Ctrl-D sur Unix ou Ctrl-Z sur DOS).

Exemple : `char c;  c=getchar();`

4.3.2 Lecture formatée sur `stdin`

`int scanf(const char *format[,liste_d_expressions])`

lit une suite de caractères sur `stdin`, l'interprète suivant la chaîne de caractères *`format`* et stocke les résultats aux adresses définies par *`liste_d_expressions`* (adresses de variables en général). La chaîne *`format`* contient des spécifications de conversions permettant d'interpréter les entrées.

Principaux spécificateurs de format

`%d` : la donnée attendue en entrée est un entier en base 10
`%u` : la donnée attendue en entrée est un entier non signé (naturel) en base 10
`%o` : la donnée attendue en entrée est un entier non signé (naturel) en base 8
`%x` : la donnée attendue en entrée est un entier non signé (naturel) en base 16
`%c` : la donnée attendue en entrée est un caractère
`%f` : la donnée attendue en entrée est un flottant

Variantes :

`%ld`, `%lu`, `%lo`, `%lx` : entiers longs
`%hd`, `%hu`, `%ho`, `%hx` : entiers courts
`%lf` : flottant double précision
`%Lf` : flottant long double précision

Exemple :

```
long int i;
float x;
scanf("%ld%f",&i,&x);
```

Sur `stdin`: `56789␣789.7` `i` ⟵ `56789`, `x` ⟵ `789.7`

Remarques :

- Pour que `scanf` modifie la valeur d'une variable, il faut lui passer son adresse (par exemple `&i`).
- La valeur de retour de `scanf` est le nombre d'éléments lus et mémorisés (peut être utilisé pour « fiabiliser » la lecture). Dans l'exemple précédent, `scanf` retourne 2.

Espace entre les spécificateurs de format

Il permet de sauter les caractères séparateurs (espaces, tabulations, saut de lignes).
Exemple : `int i; char c:`

Cas 1	*Cas 2*
`scanf("%d␣%c",&i,&c);`	`scanf("%d%c",&i,&c);` ⇕ `scanf("%d",&i); scanf("%c",&c);`

`printf("'%d','%c'\n",i,c);`

Entrée (`sdtin`)	Sortie (`stdout`)	
	Cas 1	*Cas 2*
`12␣d`↩	`'12','d'`↩	`'12','␣'`↩
`12d`↩	`'12','d'`↩	`'12','d'`↩
`12`↩ `d`↩	`'12','d'`↩	`'12','`↩ `'`↩

Remarque : dans le tableau précedent, le dernier affichage dans le cas 2 est effectué dès que l'on a tapé `12`↩.

Ignorer une valeur : `*`

Exemple :

```
int␣nblus,h,m,s;
nblus=scanf("%d␣%*c␣%d␣%*c␣%d␣%*c",&h,&m,&s);
```

Sur `stdin`: `17␣H␣35␣M␣30␣S` `h` ⟵ `17`, `m` ⟵ `35`, `s` ⟵ `30`, `nblus` ⟵ `3`

Remarque : les espaces placés entre les spécificateurs de format servent à sauter les séparateurs.

4.4 Entrées-sorties de chaînes de caractères

4.4.1 Représentation des chaînes de caractères

Une chaîne de caractères est stockée dans un tableau de caractères dans lequel on met le caractère `'\0'` pour préciser la fin de la chaîne.

- *Initialisation à la déclaration :* `char ch[4]="abc";`
  Ici `"abc"` n'est pas une constante chaîne de caractères, mais une facilité d'écriture. En effet :
  `char ch[4]="abc";` ⟺ `char ch[4]={'a','b','c','\0'};`
  On peut omettre la taille :
  `char ch[]="abc";` réserve 4 caractères.
- *Affectation ou initialisation après la déclaration :*
  `char ch[4];`
  `ch="abc";` n'est pas valide ; en revanche, on peut écrire :
  `ch[0]='a'; ch[1]='b'; ch[2]='c'; ch[3]='\0';`
  On peut aussi utiliser les outils fournis par la bibliothèque standard de traitement des chaînes de caractères (`string.h`).

4.4.2 Écritures sur `stdout`

- `printf` *avec le spécificateur de format* `%s` :
  Exemple :
  `char␣ch[]="bonjour";`
  `printf("Voici␣ma␣chaîne␣:␣%s!\n",ch);`
  produit :
  `Voici␣ma␣chaîne␣:␣bonjour!`↩
  `printf("'%5.3s'\n",ch);`
  où 3 est le nombre maximal de caractères à prélever dans la chaîne et 5 est le nombre minimal de caractères à afficher, produit :
  `'␣␣bon'`↩
- *la fonction* `puts` :
  `puts(ch);`
  produit :
  `bonjour`↩

4.4.3 Lectures sur `stdin`

Dans tous les cas, il faut avoir prévu la place suffisante pour pouvoir stocker la chaîne.

- `scanf` *avec le spécificateur* `%s` :
  Exemple :

```
char ch[10];
scanf("%s",ch);
```

Sur `stdin`: `abc␣def`↩

`ch[0]`	`'a'`
	`'b'`
	`'c'`
	`'\0'`
`ch[9]`	

  - `'\0'` est rajouté automatiquement ;
  - la lecture s'arrête au premier séparateur rencontré ;
  - **problème** si la chaîne tapée sur `stdin` est trop longue (plus de 9 caractères dans l'exemple précédent).

  Une solution au dernier problème consiste à préciser le nombre maximum de caractères lus (on ne compte pas le `'\0'`).
  Exemple :

```
char ch[10];
scanf("%9s",ch);
```

Sur `stdin`: `abcdefghijklmn`↩

`ch[0]`	`'a'`
	`'b'`
	`'c'`
	`'d'`
	`'e'`
	`'f'`
	`'g'`
	`'h'`
	`'i'`
`ch[9]`	`'\0'`

  Remarque : les caractères `jklmn`↩ restent dans le buffer d'entrée.
- *La fonction* `gets` :
  Exemple :
  `char ch[10]; gets(ch); printf("'%s'\n",ch);`
  Sur `stdin`: `abc␣def`↩ ⟹ sur `stdout`: `'abc␣def'`↩
  - `'\0'` est rajouté automatiquement ;

- la lecture s'arrête à la fin de la ligne (`'\n'`) ;
- le caractère `'\n'` est consommé mais pas stocké dans la chaîne ;
- **problème** si la chaîne tapée sur `stdin` est trop longue (plus de 9 caractères dans l'exemple précédent).

- *La fonction* `fgets` :
  Exemple :

```
char ch[6];
fgets(ch,6,stdin);
printf("'%s'\n",ch);
```

stdin	stdout	ch
ab␣c↩	'ab␣c↩ '↩	'a' 'b' '␣' 'c' '\n' '\0'
ab␣cde↩	'ab␣cd'↩ Les caractères 'e' et '\n' restent dans le buffer d'entrée.	'a' 'b' '␣' 'c' 'd' '\0'

Cette fonction ressemble à `gets`, mais :

- on précise le nombre maximum de caractères lus + 1 (6 dans l'exemple précédent), ce qui permet d'éviter les débordements ;
- le caractère `'\n'` est stocké dans la chaîne si elle n'est pas trop longue ;
- la fonction `fgets` retourne l'adresse de la chaîne ou `NULL` en cas de problème.

4.5 Entrées-sorties sur les chaînes de caractères

4.5.1 Écriture dans une chaîne : `sprintf`

La fonction `sprintf` fonctionne comme `printf` mais, au lieu d'écrire sur `stdout`, elle écrit dans une chaîne passée en premier paramètre.

Exemple :

```
char ch[100];
float pi=3.14;
int i=12;
sprintf(ch,"pi=%f, i=%d",pi,i);
printf("'%s'\n",ch);
```

produit sur `stdout` :
`'pi=3.140000,␣i=12'↩`

La valeur de retour de `sprintf` est le nombre de caractères écrits dans la chaîne sans compter le `'\0'` ou un entier négatif en cas de problème (mais pas en cas de débordement...). Dans l'exemple précédent, la valeur retournée est `17`.

Exemple de lecture contrôlée de chaîne de caractères :

On souhaite lire une chaîne de caractères avec `scanf` en évitant les débordements et en utilisant une

constante symbolique pour définir le nombre maximal de caractères autorisés pour la chaîne.

Solution incorrecte	Solution correcte

`#define MAX 10`
⋮
`char ch[MAX+1];`
`scanf("%MAXs",ch);`
	`#define MAX 10`
	⋮
	`char ch[MAX+1];`
	`char format[13];`
	`sprintf(format,"%%%ds",MAX);`
	`scanf(format,ch);`

Prendre `13` caractères pour la chaîne `format` permet d'avoir une constante symbolique `MAX` dont la valeur est un entier pouvant comporter jusqu'à 10 chiffres.

4.5.2 Lecture dans une chaîne : `sscanf`

La fonction `sscanf` fonctionne comme `scanf` mais, au lieu de lire sur `stdin`, elle lit dans une chaîne passée en premier paramètre.

Exemple :

```
char ch[]="12 3.14";
int i; float x;
sscanf(ch,"%d%f",&i,&x);
```

produit les affectations : `i` ⟵ `12` et `x` ⟵ `3.14`.

La valeur de retour de `sscanf` est le nombre de valeurs mémorisées ou `EOF` en cas de problème (par exemple si aucune valeur n'a pu être lue en accord avec le format avant la fin de la chaîne). Dans l'exemple précédent, la valeur retournée est `2`.

4.6 Exemple d'utilisation des chaînes de caractères pour réaliser des entrées

On souhaite effectuer une lecture sur `stdin` avec le comportement suivant :

- si l'utilisateur appuie seulement sur la touche « Entrée » (`'\n'`), c'est une valeur par défaut qui est prise en compte ;
- si l'utilisateur saisit autre chose qu'un entier, le programme redemande la saisie ;
- si l'utilisateur tape un entier, il est pris en compte.

```
#include <stdio.h>
#define DEFAUT 1000

int main(void)
{
  char ligne[BUFSIZ];
  int valeur=DEFAUT;

  while (1)
  {
    printf("Donner un entier (%d par défaut) : ",valeur); fflush(stdout);
    fgets(ligne,BUFSIZ,stdin);
    if (ligne[0]=='\n') break;
    if (sscanf(ligne,"%d",&valeur)==1) break;
  }
  /* Traitement de la valeur */
  ...
```

4.7 Gestion des tampons d'entrée et de sortie

À chaque flux d'entrée et de sortie est associée une zone mémoire appelée tampon ou *buffer* qui sert d'intermédiaire entre le programme et le périphérique concerné.

4.7.1 Vidage du tampon d'entrée

Le tampon d'entrée contient les caractères tapés par l'utilisateur. Il fonctionne selon le principe de la file (premier arrivé = premier sorti). Lors de l'exécution de l'instruction `car=getchar();` deux situations peuvent se produire. Si le tampon d'entrée n'est pas vide, la variable `car` reçoit le caractère le plus ancien se trouvant dans le tampon d'entrée et ce caractère est enlevé du tampon d'entrée. Si le tampon d'entrée est vide, le déroulement du programme est stoppé jusqu'à ce que le tampon d'entrée reçoive un ou plusieurs caractères. L'ajout de caractères dans le tampon d'entrée ne se fait qu'au moment ou l'utilisateur valide sa frappe avec le caractère saut de ligne (représenté ici par le symbole ↩ et correspondant à la constante caractère `'\n'`). Par exemple, si l'utilisateur tape les caractères `texTe TApé`↩, alors le contenu du tampon d'entrée ne sera modifié qu'au moment de la frappe de la touche ↩ et, à ce moment-là, le tampon d'entrée recevra onze caractères (neuf caractères alphabétiques, un espace et le `'\n'`).

Parmi les conséquences d'un tel fonctionnement, il y a la nécessité de vider le tampon d'entrée. Ce vidage n'est utile que lors de la lecture d'un caractère, à un moment où le tampon d'entrée risque de ne pas être vide.

Exemple : la séquence suivante n'est pas correctement écrite :

```
printf("Êtes-vous d'accord (o/n) ?\n"); /* M1 */
rep=getchar();
while ((rep!='o') && (rep!='n'))
{
  printf("Tapez o ou n :\n"); /* M2 */
  rep=getchar();
}
```

En effet, si l'utilisateur tape, par mégarde, la séquence `a`↩, en réponse au message `M1`, alors le message `M2` s'affichera deux fois au lieu d'une, puisque le tampon d'entrée contient deux caractères (`a` et ↩) différents des deux caractères autorisés (`o` et `n`). Pour remédier à ce problème, il faut procéder à un vidage conditionnel du tampon d'entrée avant l'instruction de lecture de la boucle :

```
printf("Êtes-vous d'accord (o/n) ?\n"); /* M1 */
rep=getchar();
while ((rep!='o') && (rep!='n'))
{
  printf("Tapez o ou n :\n"); /* M2 */
  if (rep!='\n') while (getchar()!='\n');
  rep=getchar();
}
```

La condition `(rep!='\n')` permet de prendre en compte la possibilité que l'utilisateur ait pu taper ↩ seulement. En effet, dans ce cas, si l'on omet la condition, la boucle `while (getchar()!='\n');` serait bloquante.

4.7.2 Vidage du tampon de sortie

Les fonctions d'affichage envoient les caractères non pas directement à l'écran, mais dans le tampon de sortie. L'affichage sur l'écran accompagné du vidage du tampon de sortie se fait dans deux cas :

- soit quand le caractère `'\n'` doit être affiché ;
- soit quand on utilise explicitement l'instruction `fflush(stdout);`

Pour éviter des confusions dues à une mauvaise synchronisation entre les entrées clavier et les affichages à l'écran, il est conseillé de faire suivre les instructions d'affichage de l'instruction `fflush(stdout);` si la chaîne à afficher ne se termine pas par la caractère `'\n'`.

Chapitre 5

POINTEURS

5.1 Introduction

Un pointeur est une variable susceptible de contenir une adresse mémoire.

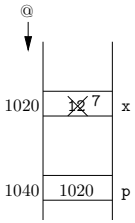
Déclaration : *type_pointé* * *identificateur*;

5.2 Opérateurs

Opérateur d'adresse : `&`*expression*
Expression doit désigner un objet de type quelconque qui possède une adresse mémoire. L'expression &*Expression* vaut l'adresse de cet objet.

Opérateur d'indirection : `*`*adresse*
L'opérateur d'indirection, ou de « déréférenciation », permet d'accéder à l'objet pointé : l'expression `*`*adresse* désigne l'objet qui se trouve à l'adresse *adresse*.
Exemple :

```
1  int x=12;
2  int *p;
3  p=&x;
4  *p=7;
```



À partir de la troisième instruction, on dit que :
« p pointe sur x » ⟺ « p contient l'adresse de x » ⟺ « la valeur de la variable p est l'adresse de x ».

5.3 Arithmétique des pointeurs

pointeur+*entier* = valeur du *pointeur* + (*entier* × $\overbrace{\text{taille}}^{\text{octets}}$ de l'objet pointé).
pointeur-*entier* = valeur du *pointeur* − (*entier* × taille de l'objet pointé).
$pointeur_1$-$pointeur_2$ = (valeur du $pointeur_1$ − valeur du $pointeur_2$) / taille de l'objet pointé.

5.4 Comparaisons de pointeurs

Pour comparer deux pointeurs, on peut utiliser les opérateurs : `==` `!=` `<` `<=` `>` `>=`

5.5 Pointeur « nul »

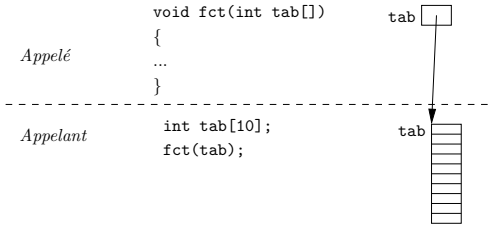
Constante symbolique `NULL` valant 0 qui désigne une adresse mémoire qui n'existe pas. La constante `NULL` est définie dans `stdio.h`, `stdlib.h` et `stddef.h`. Après l'instruction `int *p=NULL;` on dit que p pointe sur « rien ».

5.6 Pointeurs et tableaux

Le nom d'un tableau est considéré comme une constante adresse qui vaut l'adresse du premier élément du tableau. Après la déclaration `int tab[10];` on a `tab == &tab[0]`

Conséquences

- **Accès aux éléments d'un tableau :** `tab[i] == *(tab+i)`
- **Passage d'un tableau en paramètre :**

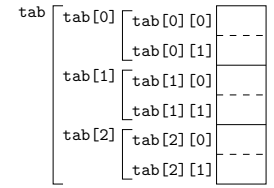


Un paramètre formel de type tableau est en fait un pointeur (vers le premier élément du tableau effectivement passé à la fonction). Dans l'exemple précédent, on aurait pu aussi écrire : `void fct(int *tab)`

Exemple de parcours d'un tableau :

```
int tab[10],*p,*fin;
fin=tab+10;
for (p=tab; p<fin; p++) *p=0;
```

Pointeurs et tableaux à deux indices
La déclaration `int tab[3][2];` est équivalente à `int (tab[3])[2];`. Donc `tab[3]` est un tableau de 2 `int` et `tab` est un tableau de 3 tableaux de 2 `int`.



On peut écrire :

```
tab[i][j] == (tab[i])[j]
          == *((tab[i])+j)
          == *(*(tab+i)+j)
```

Dans la dernière expression, pour faire l'addition avec `j`, on doit seulement connaître la taille des `tab[i][j]` (ici la taille d'un `int`). Pour faire l'addition avec `i`, on doit connaître la taille des éléments de `tab`, c'est-à-dire multiplier le nombre d'éléments des `tab[i]` (ici 2) par la taille des `tab[i][j]`. Ceci explique que, quand on passe un tableau à 2 indices en paramètre, il faut préciser la seconde dimension (ici 2).

Remarques : si `nbc` contient le nombre de colonnes du tableau `tab` (2 dans l'exemple précédent), on peut écrire :

```
tab[i][j] == *(*(tab+i)+j)
          == *(&tab[0][0]+i*nbc+j)
          == *(tab[0]+i*nbc+j)
```

```
tab[i] == *(tab+i)
       == &tab[i][0]
```

`tab[i]` est l'adresse du premier élément de la $i^{ème}$ ligne.

5.7 Pointeurs et structures

Priorité des opérateurs

L'opérateur `.` est prioritaire sur l'opérateur `*`
Exemple : `struct point {float x,y;} pt,*pp; pp=&pt;`
On a alors : `(*pp).x==pt.x` et `(*pp).y==pt.y`, **mais** `*pp.x` n'a pas de sens car `pp.x` n'a pas de sens.

Opérateur ->

L'expression `p->a` désigne « le champ `a` de la variable structurée pointée par `p` ».
Dans l'exemple précédent, on a : `(*pp).x==pp->x`

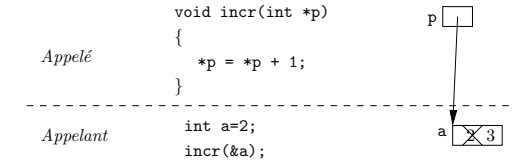
Exemple

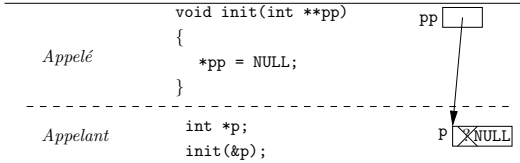
Une liste est une suite finie, éventuellement vide, d'éléments de même type. Une manière de représenter en mémoire une liste consiste à utiliser des variables structurées (les cellules de la liste) dont le dernier champ est un pointeur vers l'élément suivant (la cellule suivante). On parle alors de liste chaînée.
Exemple : pour représenter un polygone, on peut définir la structure suivante :

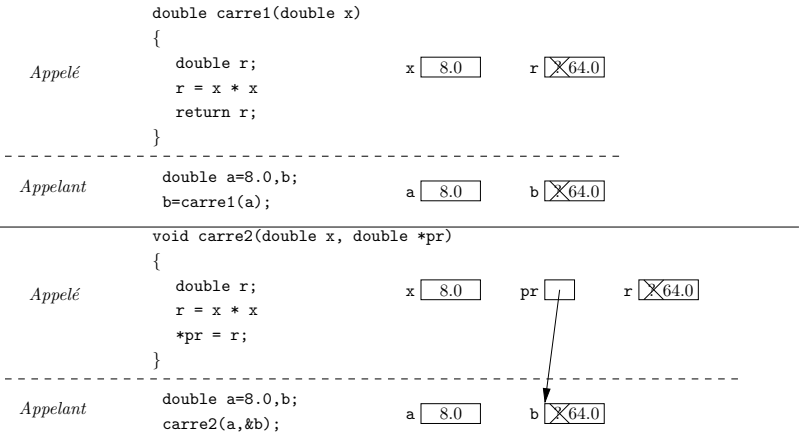
```
struct sommet
{
  float x;
  float y;
  struct sommet *pSuivant;
};
```

5.8 Utilisation des pointeurs pour le passage de paramètres

Quand un appelant veut qu'une fonction modifie un objet, l'appelant doit passer l'adresse de l'objet, le paramètre formel de la fonction doit être un pointeur vers l'objet et la fonction doit manipuler l'objet pointé.
Exemples :







5.9 Pointeurs génériques

Un *pointeur générique* ou *pointeur universel* correspond au type `void *`. Il est utilisé pour manipuler des adresses d'objets de type quelconque. Une variable de type `void *`

- ne peut pas intervenir dans des opérations arithmétiques,
- ne peut pas être « déréférencée » (opérateur `*`),
- peut recevoir n'importe quelle adresse (la valeur de n'importe quel pointeur),
- peut être affectée à n'importe quel pointeur si les contraintes d'alignement en mémoire sont respectées (selon leur type, certaines valeurs ne peuvent pas être stockées à n'importe quelle adresse en mémoire).

Remarque : quand on veut accéder aux objets pointés par un pointeur générique, on est amené à effectuer des conversions de type (vers le type `char *`, par exemple).
Des exemples d'utilisation des pointeurs génériques sont donnés au chapitre 12.

5.10 Pointeurs de fonctions

Une variable de type pointeur de fonctions est susceptible de contenir l'adresse d'une fonction (l'adresse de début du code exécutable d'une fonction). Une déclaration de pointeur de fonctions peut prendre les formes suivantes :

$$type\_retourné\ (*nom\_variable)(type_1\ ident_1, type_2\ ident_2, \ldots, type_n\ ident_n)$$

ou bien

$$type\_retourné\ (*nom\_variable)(type_1, type_2, \ldots, type_n)$$

Exemple : `int (*pf)(float r); /* ou int (*pf)(float); */`

- `pf` : le nom de la variable déclarée est `pf`.
- `*pf` : la variable `pf` est un pointeur.
- `(*pf)()` : l'objet pointé est une fonction.
- `int (*pf)(float r)` : l'objet pointé est une fonction retournant un entier et ayant un seul paramètre de type flottant.

Attention : il ne faut pas oublier les parenthèses autour de `*pf`, sinon le compilateur considèrerait que `pf` est une fonction retournant un pointeur sur un entier et ayant un seul paramètre de type flottant.
Si les deux fonctions suivantes sont définies :

```
int f1(float);
```

```
int f2(float);
```
alors on peut écrire :
```
pf=&f1; /* ou pf=f1; */
pf=&f2; /* ou pf=f2; */
```
et faire l'appel :
```
(*pf)(3.14);
```
qui sera équivalent, selon le cas, à `f1(3.14);` ou à `f2(3.14);`.

L'intérêt essentiel des pointeurs de fonctions est de pouvoir passer des fonctions en paramètres à une fonction. Des exemples sont donnés au chapitre 12.

Chapitre 6

GESTION DYNAMIQUE DE LA MÉMOIRE

6.1 Motivations

La gestion dynamique de la mémoire permet d'utiliser des objets dont la taille n'est connue qu'au moment de l'exécution du programme. On peut manipuler des emplacements en mémoire alloués et libérés selon les besoins. L'accès à ces emplacements se fait grâce aux pointeurs.

6.2 Fonctions de gestion de la mémoire

Ces fonctions nécessitent de faire appel à la directive :

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

- **malloc**

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

  La fonction `malloc` alloue *`taille`* octets consécutifs et retourne l'adresse [1] de l'emplacement, c'est-à-dire l'adresse du premier octet, ou `NULL` s'il y a un problème.

- **calloc**

```
void *calloc(size_t nb_blocs, size_t taille);
```

  La fonction `calloc` alloue *`nb_blocs`* blocs consécutifs de *`taille`* octets, les initialise à zéro binaire (tous les bits à 0) et retourne l'adresse de l'emplacement ou `NULL` s'il y a un problème.
  Remarque : `malloc(m*n)` et `calloc(m,n)` allouent le même nombre d'octets.

- **realloc**

```
void *realloc(void *p, size_t taille);
```

  La fonction `realloc` modifie la taille d'un emplacement qui a déjà été alloué dynamiquement.
  - *`p`* : adresse de l'emplacement dont on veut modifier la taille. Cette adresse doit avoir été obtenue préalablement par un appel à `malloc`, `calloc` ou `realloc`.
  - *`taille`* : nouvelle taille.

  La fonction `realloc` retourne l'adresse du nouvel emplacement qui peut être différente ou pas de l'ancienne.
  - Si la nouvelle taille est inférieure à l'ancienne, le contenu des *`taille`* premiers octets est inchangé.
  - Si la nouvelle taille est supérieure à l'ancienne, le contenu de l'ancien emplacement est conservé.

  S'il y a un problème, `NULL` est retourné et l'ancien emplacement n'est alors pas libéré.
  Remarques : `realloc(NULL,n)` ⇔ `malloc(n)`
  `realloc(p,0)` ⇔ `free(p)`
  `realloc(p,0)` retourne `NULL`

- **free**

```
void free(void *p);
```

1. Ces fonctions retournent des pointeurs universels qui peuvent, ici, être affectés à n'importe quel pointeur.

La fonction `free` libère l'emplacement d'adresse *`p`*. Cette adresse doit avoir été obtenue préalablement par un appel à `malloc`, `calloc` ou `realloc`.

6.3 Accès à la taille d'un emplacement en mémoire

L'opérateur `sizeof` permet de connaître la taille en octets d'un objet. La valeur obtenue est de type `size_t`. Son opérande peut être :

- un type : `sizeof` *`type`*
  Exemple : `sizeof(int *)`
- une expression : `sizeof` *`expression`*
  Attention : cette possibilité est à utiliser avec précaution. En effet, si l'on déclare :
  `int n; int t[10]; short s;`
  - `sizeof(n)` est la taille de `n` ;
  - `sizeof(t)` vaut `10*sizeof(int)` *sauf* si `t` est le paramètre formel d'une fonction, dans ce cas `sizeof(t)==sizeof(int *)` ;
  - `sizeof(n++)` est la taille de `n` *mais* la variable `n` n'est pas incrémentée ;
  - `sizeof(s+1)!=sizeof(s)` et `sizeof(s+1)==sizeof(int)` car l'expression `s+1` est de type `int` puisqu'une conversion implicite est effectué (`short` → `int`).

6.4 Exemple

vartab.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h> /* pour isspace */
4
5  /* Nombre minimum d'éléments dans les tableaux */
6  #define NBMIN 5
7
8  /* Lit sur stdin une série de int et les place dans un tableau alloué
9   * dynamiquement.
10  * La lecture s'arrête quand 0 est rencontré. Il est consommé mais pas stocké.
11  * L'adresse du tableau doit être fournie en paramètre.
12  * Retourne le nombre d'entiers lus (sans le 0) ou 0 si problème.
13  * La libération de la zone mémoire est à la charge de l'appelant.
14  */
15 int LireTab(int **ptab)
16 {
17   int *t,*aux,n;
18   size_t nb=NBMIN;
19   int i=0;
20
21   t=malloc(nb*sizeof(int));
22   if (t==NULL) return 0;
23   printf("Tapez des entiers positifs et 0 pour finir :\n");
24   scanf("%d",&n);
25   while (n!=0)
26   {
27     if (i==nb) /* on a remplit le tableau */
28     {
29       nb+=NBMIN;
30       aux=realloc(t,nb*sizeof(int)); /* on agrandit le tableau */
```

```
31        if (aux==NULL) { free(t); return 0; }
32        t=aux;
33        printf("réallocation de %d éléments\n",nb); /* pour vérifer */
34      }
35      t[i++]=n;
36      scanf("%d",&n);
37    }
38    *ptab=t;
39    return i;
40  }
41
42  /* Lit sur stdin une chaîne de caractères et la place dans une zone mémoire
43   * allouée dynamiquement. Les séparateurs précédant la chaîne sont ignorés.
44   * La lecture s'arrête au premier saut de ligne rencontré. Ce caractère est
45   * consommé mais pas stocké dans la chaîne.
46   * Le caractère '\0' est automatiquement rajouté en fin de chaîne.
47   * Retourne l'adresse de la chaine ou NULL si problème.
48   * La libération de la chaîne est à la charge de l'appelant.
49   */
50  char *LireChaine(void)
51  {
52    char *ch,*aux;
53    int car;
54    size_t nb=NBMIN;
55    int i=0;
56
57    ch=malloc(nb);
58    if (ch==NULL) return NULL;
59    do car=getchar(); while (isspace(car)); /* pour sauter les séparateurs */
60    while ((car!='\n')&&(car!=EOF))
61    {
62      if (i==nb) /* on a remplit le tableau */
63      {
64        nb+=NBMIN;
65        aux=realloc(ch,nb); /* on agrandit le tableau */
66        if (aux==NULL) { free(ch); return NULL; }
67        ch=aux;
68        printf("réallocation de %d éléments\n",nb); /* pour vérifer */
69      }
70      ch[i++]=car;
71      car=getchar();
72    }
73    ch[i]='\0';
74    return ch;
75  }
76
77  int main(void)
78  {
79    int *tab;
80    int i,n;
81    char *chaine;
82
83    n=LireTab(&tab);
84    for (i=0;i<n;i++) printf("%d ",tab[i]);
85    printf("\n%d éléments\n",n);
```



```
86    if (n!=0) free(tab);
87    printf("Tapez une chaîne :\n");
88    chaine=LireChaine();
89    if (chaine!=NULL)
90    {
91      printf("%s\n",chaine);
92      free(chaine);
93    }
94    return 0;
95  }
```

Exemple de tableaux dont la taille varie pendant l'exécution du programme.

L'exécution de ce programme produit le résultat suivant (les caractères tapés par l'utilisateur sont les caractères gras):

```
Tapez des entiers positifs et 0 pour finir :
12↩
54↩
85↩
63↩
47↩
58↩
réallocation de 10 éléments
52↩
15↩
63↩
58↩
0↩
12 54 85 63 47 58 52 15 63 58
10 éléments
Tapez une chaîne :
␣␣␣␣Ceci est une chaîne de caractères↩
réallocation de 10 éléments
réallocation de 15 éléments
réallocation de 20 éléments
réallocation de 25 éléments
réallocation de 30 éléments
réallocation de 35 éléments
Ceci est une chaîne de caractères
```

Chapitre 7

GESTION DES FICHIERS

7.1 Introduction

Ce chapitre a pour objectif de présenter les outils de gestion des fichiers fournis par la bibliothèque standard du langage C.

Même si, au bout du compte, tout est binaire, on distingue généralement deux types de fichiers :

- les fichiers de texte : chaque octet qui les compose représente un caractère et ces caractères sont organisés en lignes (le contenu du fichier apparaît de manière lisible dans la fenêtre d'un éditeur de texte) ;
- les fichiers binaires : ce sont tous les autres fichiers (leur contenu apparaît sous la forme de caractères illisibles dans la fenêtre d'un éditeur).

La différence entre les deux types de fichiers réside dans la manière dont les informations y sont codées.
Exemple : l'entier 24 est codé par :

- `00110010 00110100` dans un fichier de texte : un octet pour le code ASCII du caractère 2, qui vaut 50, et un octet pour le code ASCII du caractère 4, qui vaut 52 ;
- `00000000 00000000 00000000 00011000` dans un fichier binaire si les entiers sont codés sur quatre octets.

Même si ce n'est pas une obligation, on n'utilise généralement pas les mêmes fonctions pour réaliser des entrées-sorties dans les fichiers de texte et les fichiers binaires.

Les fonctions de gestion des fichiers nécessitent la directive `#include <stdio.h>`.

7.2 Ouverture

La manipulation d'un fichier va se faire à travers un identificateur de fichier (aussi appelé *flux* ou *stream*) de type `FILE *`.

La fonction

`FILE *fopen(const char *nom, const char *mode)`

ouvre le fichier de désignation *nom* dans le mode *mode* et retourne son identificateur ou `NULL` s'il y a un problème.

Le *mode* est une chaîne de un, deux ou trois caractères :

- le premier caractère indique si le fichier doit être ouvert en lecture (`r` pour *read*), en écriture (`w` pour *write*) ou en écriture en fin de fichier (`a` pour *append*) ;
- le deuxième caractère, qui peut être ajouté au premier, est le signe `+` qui précise que l'on souhaite effectuer une mise à jour (lecture et écriture) ;
- le troisième caractère indique si le fichier doit être considéré comme un fichier de texte (`t` pour *text*), qui est le type par défaut, ou comme un fichier binaire (`b` pour *binary*).

Exemple :
```
FILE *id_fich;
id_fich=fopen("data.txt","rt");
```
Remarque : trois identificateurs de fichiers particuliers sont disponibles et n'ont besoin ni d'être ouverts, ni d'être fermés : `stdin` (entrée standard), `stdout` (sortie standard) et `stderr` (sortie standard des erreurs).

7.3 Lecture

7.3.1 Lecture dans un fichier de texte

Lecture d'un caractère

La fonction

`int fgetc(FILE *id_fich)`

lit le caractère courant dans le fichier d'identificateur *id_fich* et retourne ce caractère ou `EOF` si la fin du fichier est atteinte.
Remarques : la macro `getc` effectue la même chose que `fgetc` et la macro `getchar` effectue la même chose que `fgetc(stdin)`.

Lecture d'une chaîne de caractères

La fonction

`char *fgets(char *ch, int n, FILE *id_fich)`

lit au maximum $n-1$ caractères dans le fichier d'identificateur *id_fich*, s'arrête si elle rencontre le caractère `'\n'`, les range (y compris l'éventuel `'\n'`) dans la chaîne d'adresse *ch* en complétant par le caractère `'\0'`. Elle retourne l'adresse de la chaîne ou `NULL` s'il y a un problème ou si la fin de fichier a été atteinte.

Lecture formatée

La fonction

`int fscanf(FILE *id_fich, const char *format[,liste_d_expressions])`

se comporte de la même manière que `scanf` (voir page 22) mais effectue la lecture dans le fichier d'identificateur *id_fich* au lieu du clavier.

7.3.2 Lecture dans un fichier binaire

La fonction

`size_t fread(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)`

lit dans le fichier d'identificateur *id_fich* *nb_infos* informations (blocs d'octets) de *taille* octets chacunes et les range en mémoire à l'adresse *pt* (la zone mémoire doit avoir été préalablement allouée). Elle retourne le nombre d'informations effectivement lues. Ce nombre peut être inférieur à *nb_infos* si la fin du fichier est atteinte.

7.4 Écriture

7.4.1 Écriture dans un fichier de texte

Écriture d'un caractère

La fonction

`int fputc(int c, FILE *id_fich)`

écrit dans le fichier d'identificateur *id_fich* le caractère *c* (convertit en `unsigned char`) et retourne le caractère écrit ou `EOF` s'il y a un problème.

Écriture d'une chaîne de caractères

La fonction

`int fputs(const char *ch, FILE *id_fich)`

écrit dans le fichier d'identificateur *id_fich* la chaîne d'adresse *ch* et retourne `EOF` s'il y a un problème ou une valeur négative dans le cas contraire.

Écriture formatée

La fonction

```
int fprintf(FILE *id_fich, const char *format[,liste_d_expressions])
```

se comporte de la même manière que `printf` (voir page 21) mais effectue l'écriture dans le fichier d'identificateur `id_fich` au lieu de l'écran.
Remarque : on utilise très souvent `fprintf` pour afficher les messages d'erreur sur `stderr`.
Exemple :

```
FILE *f;
f=fopen("toto.txt","rt");
if (f==NULL) fprintf(stderr,"Fichier toto.txt inexistant.\n");
```

7.4.2 Écriture dans un fichier binaire

La fonction

```
size_t fwrite(void *pt, size_t taille, size_t nb_infos, FILE *id_fich)
```

écrit dans le fichier d'identificateur `id_fich` `nb_infos` informations (blocs d'octets) de `taille` octets chacunes situées en mémoire à l'adresse `pt`. Elle retourne le nombre d'informations effectivement écrites. Ce nombre peut être inférieur à `nb_infos` dans le cas d'un disque saturé.

7.5 Fermeture

La fonction

```
int fclose(FILE *id_fich)
```

ferme le fichier d'identificateur `id_fich` et retourne 0 ou `EOF` s'il y a un problème.
Remarques :

- Il ne faut fermer un fichier que s'il est ouvert.
- Il est conseillé de fermer un fichier dès que son traitement est terminé.

7.6 Gestion de la position courante

À chaque fichier ouvert est associé un « pointeur de fichier » qui donne la position courante dans le fichier, c'est-à-dire le rang du prochain octet à lire ou à écrire. Après chaque opération de lecture ou d'écriture, ce « pointeur » est automatiquement incrémenté du nombre d'octets transférés. On parle d'*accès séquentiel* au fichier.

7.6.1 Détection de fin de fichier

La fonction

```
int feof(FILE *id_fich)
```

retourne une valeur non nulle si la fin du fichier d'identificateur `id_fich` est atteinte, ou 0 sinon.
Attention : la valeur de retour de `feof` n'est valide qu'après avoir effectué au moins une lecture.
Remarque : dans le cas d'une lecture dans un fichier binaire, la valeur de retour de la fonction `fread` peut remplacer avantageusement l'appel à la fonction `feof`.

7.6.2 Positionnement dans un fichier

Il est possible d'accéder à un fichier de manière directe (accès direct) en modifiant sa position courante grâce à la fonction

```
int fseek(FILE *id_fich, long decalage, int depart)
```


où
- `id_fich` est l'identificateur du fichier ;
- `decalage` est le nombre d'octets dont on veut se décaler par rapport à la position `depart` ;
- `depart` peut être égal à :
  - `SEEK_SET` ou `0` : début de fichier,
  - `SEEK_CUR` ou `1` : position courante dans le fichier,
  - `SEEK_END` ou `2` : fin de fichier.

La fonction `fseek` retourne 0 si tout s'est bien passé ou -1 sinon, en particuler dans le cas où on essaie de revenir en arrière avant le début du fichier. Néanmoins, quand on essaie d'avancer au-delà de la fin du fichier, `fseek` retourne 0 (tout se passe comme si on « piétinait » à la fin du fichier).

La fonction

```
void rewind(FILE *id_fich)
```

place le pointeur du fichier d'identificateur `id_fich` à son début.
Remarque : `rewind(f);` ⇔ `fseek(f,0,SEEK_SET);`

La fonction

```
long ftell(FILE *id_fich)
```

retourne la position courante du fichier d'identificateur `id_fich` ou -1 s'il y a un problème.

7.7 Suppression d'un fichier

La fonction

```
int remove(const char *nom_fichier)
```

supprime le fichier de nom `nom_fichier` et retourne 0 ou une valeur non nulle s'il y a un problème.

7.8 Conseils

Le tableau 7.1 fait une synthèse des principales fonctions généralement utilisées pour effectuer des lectures, des écritures et pour détecter la fin de fichier en fonction du type de fichier.

	Lecture			**Écriture**	
Type fichier	*Mode ouverture*	*Fonction*	*Détection fin de fichier*	*Mode ouverture*	*Fonction*
Texte	`"rt"`	`fscanf`	`feof`	`"wt"` ou `"at"`	`fprintf`
Binaire	`"rb"`	`fread`	valeur de retour de `fread`	`"wb"` ou `"ab"`	`fwrite`

TAB. 7.1 – *Conseils pour la gestion des fichiers.*

Chapitre 8

COMMUNICATION AVEC LE SYSTÈME D'EXPLOITATION

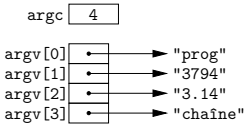
8.1 Paramètres d'un programme

Il est possible de récupérer les paramètres (arguments) qui sont passés au programme au moment du lancement de son exécution. Pour cela, il faut modifier l'en-tête de la fonction principale et utiliser :

```
int main(int argc, char *argv[])
```

où
- *argc* contient le nombre de paramètres effectivement passés au programme augmenté de 1 pour le nom du programme exécutable ;
- *argv* est un tableau de *argc* pointeurs vers les chaînes de caractères qui constituent les paramètres effectifs du programme.

Exemple : si `prog` est le nom du programme exécutable correspondant au fichier source `prog.c` et si on lance son exécution en tapant la ligne de commande : `prog 3794 3.14 chaîne`
alors on a :



Ces paramètres peuvent être utilisés dans le programme, sachant qu'il peut être nécessaire d'effectuer des conversions de chaînes de caractères en nombres grâce par exemple à `atoi` ou `atof` (déclarées dans `stdlib.h`) ou bien à `sscanf` (voir page 26).

8.2 Lancement de commandes du système

La fonction (déclarée dans `stdlib.h`)

```
int system (const char *commande)
```

permet de lancer l'exécution de la commande *commande*. Si le paramètre *commande* est égal à `NULL`, la fonction `system` retourne une valeur non nulle s'il existe au moins une commande du système qui lui est accessible. Si le paramètre *commande* est différent de `NULL`, la valeur de retour dépend de l'implémentation.

8.3 Sortie d'un programme

Il y a deux manières de terminer l'exécution d'un programme et de retourner au système un entier appelé *code de retour* :

- en utilisant dans la fonction `main` l'instruction

```
return code_retour;
```


- en faisant appel depuis n'importe où dans le programme à la fonction (déclarée dans `stdlib.h`)

```
void exit(int code_retour)
```

Remarques :
- L'exécution de l'opérateur `return` dans une fonction autre que `main` permet d'interrompre l'exécution de la fonction en question et de rendre la main à la fonction appelante.
- Les conventions des commandes Unix préconisent qu'un programme retourne 0 quand tout s'est bien passé ou un entier positif permettant de décrire la situation anormale.

Chapitre 9

UTILISATION DU PRÉPROCESSEUR

9.1 Introduction

Le préprocesseur est un programme qui effectue un pré-traitement, lors de la compilation d'un programme C, en supprimant dans un premier temps les commentaires, puis en traitant les directives (lignes commençant par le caractères `#`). Ensuite, le préprocesseur envoie le programme modifié au compilateur.
Remarque : sous Unix, la commande `gcc -E -P prog.c` affiche sur `stdout` le programme `prog.c` après le passage du préprocesseur.

Les trois principaux types de directives sont :

- les directives d'inclusion de fichiers,
- les directives de compilation conditionnelle,
- les directives de substitution symbolique.

9.2 Directives d'inclusion de fichiers

La directive

`#include <`*nom_fichier*`>`

insère le contenu du fichier *nom_fichier*. Le préprocesseur va chercher ce fichier dans un ou plusieurs répertoires particuliers (souvent `/usr/include`).

La variante

`#include "`*nom_fichier*`"`

permet d'aller chercher le fichier d'abord dans le répertoire courant. La chaîne de caractères *nom_fichier* peut être une désignation complète (relative ou absolue) d'un fichier.

9.3 Directives de compilation conditionnelle

Ces directives permettent de considérer ou d'ignorer certaines lignes d'un programme.

```
#if comparaison1
texte1
#elif comparaison2
texte2
#else
texte3
#endif
```

Les comparaisons portent sur des constantes entières (pas sur des variables du programme dont l'évaluation n'est possible qu'au moment de l'exécution).

Exemple :

```
#define TEST 20
...
#if TEST > 5
printf("OK\n");
#elif TEST <= 0
scanf("%d",&a);
#else
scanf("%d",&b);
#endif
...
```

Ici, le compilateur recevra le fragment de programme suivant :

```
...
printf("OK\n");
...
```

```
#ifdef symbole
texte
#endif
```

```
#ifndef symbole
texte
#endif
```

Avec cette forme, *texte* est envoyé au compilateur si la constante symbolique *symbole* est définie (`#ifdef`) ou n'a pas été définie (`#ifndef`).
Exemple :

```
...
#define DEBOGAGE
...
#ifdef DEBOGAGE
fprintf(stderr,"Fichier %s, ligne %d : a==%d, b==%d\n",__FILE__,__LINE__,a,b);
#endif
```

Remarques :

- Dans l'exemple précédent, on utilise les constantes symboliques `__FILE__` qui est égale au nom du fichier source et `__LINE__` qui est égale au numéro de la ligne à l'intérieur du fichier source.
- Dans le même esprit, la bibliothèque standard fournit un outil d'aide à la mise au point qui fait largement appel aux directives du préprocesseur. La directive `#include <assert.h>` permet de faire appel à une macro dont le pseudo-prototype est le suivant :

  `void assert(int` *expression*`)`

  Si *expression* vaut 0, `assert` provoque l'interruption de l'exécution du programme et l'affichage d'un message d'erreur sur `stderr` contenant :
  - le nom du fichier source
  - le numéro de la ligne
  - l'expression

  Exemple :

```
#include <assert.h>
...
  int tab[10];
  ...
  while (condition)
  {
    assert((i>=0) && (i<10));
    tab[i]=...
    ...
  }
...
```

  Pour supprimer l'effet de la macro `assert`, il suffit de placer la ligne :
  `#define NDEBUG`
  avant la ligne :
  `#include <assert.h>`

9.4 Directive de substitution symbolique

9.4.1 Forme simple

`#define` *`symbole équivalent`*

Le préprocesseur remplace dans la suite toutes les occurrences de *`symbole`* par son *`équivalent`* (éventuellement vide), excepté :

- dans les lignes commençant par le caractère `#`,
- dans les constantes chaînes de caractères (situées entre guillemets),
- au milieu d'un identificateur (si *`symbole`* est précédé ou suivi d'un des caractères autorisés pour les identificateurs de variables).

Ce remplacement s'arrête si le préprocesseur rencontre la directive

`#undef` *`symbole`*

qui met fin à la définiton du symbole.

Le préprocesseur tient à jour une table des symboles qu'il gère et utilise de la manière suivante :

- Dès qu'il rencontre une directive `#define` :
  - si le symbole est déjà dans la table, il modifie son équivalent ;
  - si le symbole n'est pas dans la table, il ajoute une nouvelle ligne.
- Dès qu'il rencontre une directive `#undef`, il supprime la ligne correspondante dans la table.
- Dès qu'il rencontre un symbole présent dans la table et « remplaçable », il effectue des remplacements successifs jusqu'à ce que :
  - il n'ait plus rien à remplacer ou
  - il rencontre un symbole qu'il est en train de remplacer (on « reboucle »).

9.4.2 Macro-instructions

Les macro-instructions sont une forme paramétrée de la substitution symbolique. Leur syntaxe est la suivante :

`#define` *`symbole(paramètre`*$_1$*`,paramètre`*$_2$*`,...,paramètre`*$_n$*`) équivalent`*

Les « paramètres effectifs » qui suivent une occurrence de *`symbole`* dans le programme sont identifiés aux « paramètres formels ». L'*`équivalent`* est envoyé au compilateur après avoir remplacé les paramètres formels par les paramètres effectifs.

Attention : il ne doit pas y avoir d'espace entre le *`symbole`* et la parenthèse ouvrante.

Exemple : `#define ABS(x) ((x)>0?(x):-(x))`

Remarque : si l'*`équivalent`* est écrit sur plusieurs lignes, il faut terminer chaque ligne, sauf la dernière, par le caractère `\`.

Exemple :

```
#define AFF_TAB_INT(tab,n)\
{\
  int i;\
  for (i=0;i<n;i++) printf("%d ",tab[i]);\
  printf("\n");\
}
```

Chapitre 10

COMPLÉMENTS SUR LES DÉCLARATIONS

10.1 Qualifieurs

10.1.1 Qualifieur `const`

Le qualifieur `const` permet de déclarer que la valeur d'une variable ne doit pas changer lors de l'exécution du programme. Les éventuelles instructions permettant la modification de la valeur de cette variable sont signalées par le compilateur sous la forme d'avertissements (*warnings*).
Exemples :

- `const int a=12;` déclare un entier constant de valeur 12. La valeur de `a` ne peut pas être changée par la suite :

```
a=20; /* produit un avertissement */
a++;  /* produit un avertissement */

int *p;
p=&a; /* produit un avertissement */
*p=20;
```

- avec les pointeurs :

```
const char * pc1; /* la zone pointée est constante, le pointeur est modifiable */
char c;
char * const pc2=&c; /* la zone pointée est modifiable, le pointeur est constant */
const char * const pc3=&c; /* la zone pointée et le pointeur sont constants */
```

- avec les variables structurées :

```
struct date
{
  unsigned int jour;
  unsigned int mois;
  unsigned int annee;
};
const struct date revolution={14,7,1789}; /* tous les champs sont constants */

struct date
{
  unsigned int jour;
  const unsigned int mois;
  const unsigned int annee;
};
struct date revolution={14,7,1789}; /* le champ jour est modifiable */
```

Attention : les entiers déclarés avec le qualifieur `const` ne peuvent pas être utilisés pour définir la taille d'un tableau.

10.1.2 Qualifieur `volatile`

Le qualifieur `volatile` indique au compilateur que la variable peut être modifiée indépendamment des instructions du programme (interruption, périphérique). Cela permet d'interdire au compilateur de faire certaines optimisations.
Exemple : `volatile int n;`

10.2 Classes de mémorisation

10.2.1 Classe `static`

- Pour une variable globale : `static` signifie qu'elle n'est pas visible depuis d'autres fichiers.
- Pour une fonction : `static` signifie qu'elle n'est pas visible depuis d'autres fichiers.
- Pour une variable locale : la variable conserve sa valeur d'un appel à l'autre (variable rémanente).
  Exemple :
  Si l'on définit la fonction suivante :

```
void f(void)
{
  static int i=0;
  i++;
  printf("appel numéro : %d\n",i);
}
```

les deux appels : `f(); f();`
produisent l'affichage de :

```
appel numéro : 1
appel numéro : 2
```

10.2.2 Classe `extern`

Pour une fonction ou une variable, `extern` permet de préciser que la fonction ou la variable est définie ailleurs.
Exemple :

```
extern int a;
extern int f(int n);
```

La variable `a` et la fonction `f` sont alors utilisables mais il faut qu'elles soient définies [1] ailleurs et visibles.

10.2.3 Classe `register`

Les variables locales et les paramètres formels simples (scalaires) peuvent être de classe `register`. Dans ce cas, on demande au compilateur d'utiliser, dans la mesure du possible, un registre du processeur plutôt qu'un emplacement mémoire.

⇒ rapidité (discutable),

⇒ absence d'adresse mémoire (on ne peut pas utiliser l'opérateur `&`).

Exemple :

```
int f(double x, register int a)
{
  register int i;
  register double y;
  ...
}
```

10.2.4 Classe `auto`

La classe `auto` est la classe par défaut.

1. Nous considérons qu'une variable est définie lorsqu'elle est déclarée et que cette déclaration produit la réservation d'un emplacement en mémoire. La déclaration `extern int a;` déclare la variable `a` mais ne réserve aucun emplacement en mémoire.

Chapitre 11

PROGRAMMATION MODULAIRE

11.1 Introduction

Le mécanisme de compilation séparée permet d'écrire des programmes modulaires, c'est-à-dire des programmes dans lesquels les fonctions associées à un même traitement sont regroupées en modules.

11.1.1 Objectifs

- **Lisibilité :** facilité à comprendre le comportement et la mise en œuvre d'un programme en lisant le code source.
- **Maintenabilité :** facilité à détecter des erreurs dans une partie de programme et à les corriger sans en provoquer de nouvelles dans le reste du programme.
- **Portabilité :** facilité à adapter les fonctionnalités d'un programme à un nouvel environnement.
- **Extensibilité :** facilité à modifier l'implémentation d'une partie de programme ou à rajouter des fonctionnalités sans modifier le comportement de l'ensemble.
- **Réutilisabilité :** facilité à utiliser des parties de programmes différents pour en écrire de nouveaux.

11.1.2 Principes

- **Abstraction des constantes :** utiliser des constantes symboliques.

Exemple :

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

- **Factorisation de code :** éviter la duplication de code en définissant des fonctions.
- **Masquage de l'implémentation :** séparer, pour chaque module, son *interface* (ce qui doit être connu par un utilisateur du module) de son *implémentation* (la mise en œuvre). On doit pouvoir utiliser le module sans en connaître l'implémentation. On doit pouvoir modifier l'implémentation sans que l'utilisateur en soit affecté.

11.2 Modules

11.2.1 Constitution d'un module

Un module est constitué :

- d'un fichier d'en-tête (d'extension `.h`) constituant son interface et contenant principalement :
  - des définitions de structures (`struct...`),
  - des définitions de synonymes de types (`typedef...`),

  - des définitions de constantes symboliques (`#define...`),
  - des déclarations de fonctions externes (`extern...`),
  - des inclusions de fichiers d'en-tête (`#include...`),
  - des commentaires,
- d'un fichier d'implémentation (d'extension `.c`) contenant :
  - l'inclusion du fichier d'en-tête correspondant,
  - des inclusions de fichiers d'en-tête,
  - des défintions de structures et de synonymes de types privés,
  - des définitions de constantes symboliques privées,
  - des définitions de fonctions privées, c'est-à-dire locales au fichier (`static...`),
  - les définitions des fonctions déclarées dans le fichier d'en-tête correspondant.

Un utilisateur de ce module doit simplement inclure le fichier d'en-tête.

11.2.2 Encapsulation des traitements

La partie visible du module, c'est-à-dire le fichier d'en-tête, ne contient que les déclarations des fonctions (les références aux fonctions) et pas leur définitions. Leurs corps sont « cachés » dans le fichier d'implémentation.

module.h

```
/* PARTIE PUBLIQUE */
...
/* Commentaire */
extern ...f(...);
...
```

module.c

```
/* PARTIE PRIVÉE */
...
... f(...)
{
  ...
  ...
  ...
}
...
```

11.2.3 Encapsulation des données

On réalise une encapsulation des données quand il est n'est pas possible d'agir directement sur les données d'un module et qu'il est nécessaire de passer par l'appel des fonctions publiques du module (souvent appelées *méthodes* en programmation orientée objet) qui constituent une interface obligatoire. On réalise une *abstraction des données*, c'est-à-dire que, vu de l'extérieur, un module se caractérise uniquement par les spécifications de ses fonctions publiques (contenues dans le fichier d'en-tête), les détails d'implémentation étant sans importance.

L'encapsulation des données facilite la maintenance. En effet, une modification éventuelle de la structure des données d'un module n'a d'incidence que sur le module lui-même ; les utilisateurs du module ne seront pas concernés par la teneur de cette modification.

Pour réaliser cela, on peut utiliser la définition retardée des structures. Il est possible de déclarer un synonyme de type « pointeur vers une structure » dans le fichier d'en-tête et de définir le contenu (les champs) de la structure uniquement dans le fichier d'implémentation.

L'utilisateur peut alors déclarer des variables de ce type et les manipuler (les faire passer en paramètre de fonctions par exemple). La seule chose qu'il ne peut pas faire est d'accéder à l'objet pointé, c'est-à-dire au contenu de la structure pointée. En revanche, le concepteur du module peut accéder à ce contenu à l'intérieur du fichier d'implémentation. Il peut aussi changer d'implémentation sans changer le contenu du fichier d'en-tête, c'est-à-dire de manière transparente pour l'utilisateur du module. La seule contrainte est que l'implémentation doit passer par une structure.

Exemple :

module.h
```
...
typedef struct Liste *Liste;
...
extern Liste CreerListe(void);
...
```

module.c
```
...
struct Liste
{
  int x;
  int y;
  struct Liste *Suivant;
  /* ou Liste Suivant; */
};
...
Liste CreerListe(void)
{
  return NULL;
}
...
```

Le concepteur du module peut alors changer l'implémentation de la liste de points en modifiant uniquement le fichier `module.c` :

module.c
```
...
#define MAX 1000
struct Point { int x,y; };
struct Liste
{
  int NbPoints;
  struct Point TabPoints[MAX];
};
...
Liste CreerListe(void)
{
  Liste l;
  l=malloc(sizeof(struct Liste));
  l->NbPoints=0;
  return l;
}
...
```

11.2.4 Inclusions multiples

Pour pouvoir utiliser un module, il faut inclure son fichier d'en-tête. Or les modules peuvent faire appel à d'autres modules. Le phénomène des inclusions multiples survient lorsqu'un même fichier d'en-tête a été inclus plusieurs fois. Le compilateur peut alors détecter des erreurs comme la redéfinition de structure par exemple. Pour éviter cela, on peut utiliser des directives du préprocesseur selon le schéma suivant :

module.h
```
#ifndef MODULE_H
#define MODULE_H
/* Contenu du fichier d'en-tête */
...
...
#endif /*MODULE_H*/
```

Lors de la première inclusion, la constante symbolique `MODULE_H` n'est pas définie, elle l'est donc et le contenu du fichier d'en-tête est inclus. À partir de la deuxième inclusion, la constante symbolique `MODULE_H` étant définie, le contenu du fichier d'en-tête n'est plus inclus.

11.3 Compilation séparée

11.3.1 Compilation d'un module

Un module ne contenant pas de fonction `main`, on ne peut pas produire un fichier exécutable. Mais on peut produire un fichier objet en effectuant une compilation sans édition de lien. L'option `-c` de `gcc` permet d'éviter l'édition de liens.
Exemple : la commande `gcc -c module.c` produit le fichier objet `module.o`.

11.3.2 Compilation d'un programme

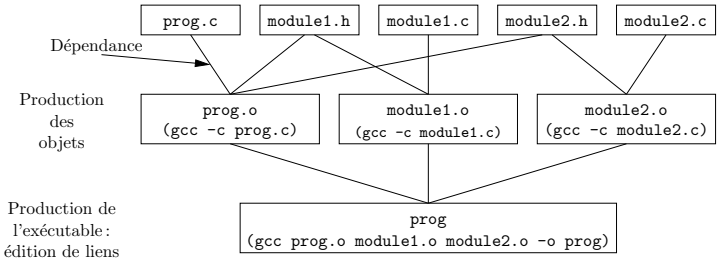
Soit le fichier `prog.c` qui contient la fonction `main` et qui utilise les fonctionnalités de deux modules (`module1.h`, `module1.c`) et (`module2.h`, `module2.c`).

prog.c
```
...
#include "module1.h"
#include "module2.h"
...
int main(void)
...
```

module1.c
```
...
#include "module1.h"
...
```

module2.c
```
...
#include "module2.h"
...
```

On peut alors établir le graphe de dépendances suivant :



Si le fichier `module1.c` est modifié, il suffit de relancer les deux commandes suivantes :

```
gcc -c module1.c
gcc prog.o module1.o module2.o -o prog
```

Pour gérer ces dépendances et ne recompiler que ce qui doit l'être, on peut faire appel à l'utilitaire `make`.

11.3.3 Utilitaire `make`

Sous Unix, l'utilitaire `make` permet de gérer la compilation séparée de manière optimale.

- **Invocation de `make` :** on peut lancer la commande
  `make -f fichier_des_dépendances`
  ou bien plus simplement
  `make`
  si le fichier de description des dépendances se trouve dans le répertoire courant et s'appelle `makefile` ou `Makefile`.

- **Syntaxe d'une description de dépendances :**

  cible:*liste_de_fichiers*
  ⟼*commande*

  le symbole ⟼ représente le caractère tabulation.
  Cette description signifie : « *cible* dépend des fichiers de la liste ». Si au moins un des fichiers de la liste a été modifié plus récemment que la *cible*, alors la *commande* est exécutée.
- **Ordre des descriptions de dépendances :** la première cible doit être l'objectif final.
- **Commentaires :** `# ceci est un commentaire`
- **Variables :**
  - Affectation : *identificateur*`=`*valeur* (par exemple : `CC=gcc`)
  - Utilisation : `$(`*identificateur*`)` (par exemple : `$(CC)`)

Makefile

```
1  # Fichier de description des dépendances du programme testlist
2
3  CC=gcc
4  CFLAGS=-ansi -pedantic -Wall -Werror
5  OBJ=testlist.o list.o
6
7  testlist:$(OBJ)
8          $(CC) $(CFLAGS) $(OBJ) -o testlist
9
10 testlist.o:testlist.c list.h
11         $(CC) $(CFLAGS) -c testlist.c
12
13 list.o:list.c list.h
14         $(CC) $(CFLAGS) -c list.c
```

Exemple de fichier de description de dépendances.

11.4 Bibliothèques sous Unix avec gcc

Une bibliothèque est une archive qui réunit un ensemble de fichiers objets (d'extension `.o`) en un seul fichier. Du point de vue de la programmation modulaire, une bibliothèque correspond à un module qui a été découpé en « sous-modules ». Chaque sous-module est un fichier source (d'extension `.c`) avec, éventuellement, un fichier d'en-tête associé (d'extension `.h`). Il est aussi possible d'avoir un fichier d'en-tête global associé à la bibliothèque.

- **Bibliothèques statiques :** il s'agit de fichiers d'extension `.a` qui sont inclus dans le fichier exécutable au moment de l'édition de liens.
  Exemple :
  - *Création :* `ar -ru libbiblio.a fich1.o fich2.o`
    L'option `-r` précise que les objets existant dans l'archive doivent être remplacés et l'option `-u` permet de n'insérer dans l'archive que les objets plus récents que les objets existants de même nom.
  - *Utilisation :* `gcc exe.o -lbiblio -o exe`
    L'option `-l` permet de réaliser l'édition de liens avec la bibliothèque `libbiblio.a`, sachant que le préfixe `lib` et l'extension sont rajoutés automatiquement. Si la bibliothèque n'est pas dans le répertoire courant, il est possible d'en préciser le chemin d'accès en utilisant l'option `-L` suivie de la désignation du répertoire en question.

- **Bibliothèques dynamiques partagées :** il s'agit de fichiers d'extension `.so` qui ne sont pas inclus dans le fichier exécutable au moment de l'édition de liens mais qui sont liés seulement au moment de l'exécution.
  - *Avantages :*
    - les exécutables sont moins volumineux ;
    - on peut modifier la bibliothèque sans devoir refaire une édition de liens ;
    - un seul exemplaire du code de la bibliothèque est chargé en mémoire même si plusieurs processus l'utilisent.
  - *Inconvénients :*
    - les exécutables ne sont pas « auto-suffisants » ;
    - un exécutable peut changer de comportement si la bibliothèque est modifiée ;
    - le chemin d'accès au répertoire contenant la bibliothèque doit se trouver dans la variable d'environnement `LD_LIBRARY_PATH`.

  Exemple :
  - *Création :* `gcc -shared -o libbiblio.so fich1.o fich2.o`
    La commande `ld` peut être utilisée à la place de `gcc`.
  - *Utilisation :* `gcc exe.o -lbiblio -o exe`
    L'option `-l` permettra, au moment de l'exécution, de réaliser l'édition de liens avec la bibliothèque `libbiblio.so`, sachant que le préfixe `lib` et l'extension sont rajoutés automatiquement. Si la bibliothèque n'est pas dans le répertoire courant, il est possible d'en préciser le chemin d'accès en utilisant l'option `-L` suivie de la désignation du répertoire en question.
  - *Exécution :* il est nécessaire de rajouter à la liste des répertoires contenus dans la variable d'environnement `LD_LIBRARY_PATH` le répertoire contenant la bibliothèque. La technique pour effectuer cette modification dépend de l'interpréteur de commandes utilisé. Par exemple, pour `bash` on pourra rajouter dans le fichier `.bash_profile` les lignes suivantes :

    ```
    LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home/moi/chemin_de_la_bibliotheque
    export LD_LIBRARY_PATH
    ```

    Pour `csh` on pourra rajouter dans le fichier `.login` la ligne suivante :

    ```
    setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/home/moi/chemin_de_la_bibliotheque
    ```
  - *Vérification des dépendances dynamiques :* la commande `ldd` permet de vérifier les dépendances dynamiques d'un fichier exécutable (pour l'exemple précédent : `ldd exe`).

Chapitre 12

GÉNÉRICITÉ

12.1 Introduction

L'idée derrière le concept de généricité, tel que l'on va l'illustrer en C, est de construire des modèles de traitements indépendants des types des objets qu'ils manipulent. Une fonction générique est une fonction qui agit sur des objets dont le type n'est *a priori* pas connu. Ce n'est qu'au moment de son utilisation qu'on lui fait passer les informations concernant les types effectifs des objets à manipuler.

Avec le langage C, on peut atteindre un certain degré de généricité en utilisant les pointeurs génériques et en les associant aux pointeurs de fonctions.

Mais on peut aussi faire appel au préprocesseur en écrivant des macro-instructions qui peuvent être utilisées avec des type différents.

12.2 Utilisation des macro-instructions

Les macro-instructions permettent d'effectuer des traitements avec un certain degré de généricité dans le sens où leurs paramètres ne sont pas typés (tout se fait par remplacement de texte).
Exemples :

- La macro-instruction : `#define ABS(x) ((x)>0?(x):-(x))`
  peut être utilisée avec des valeurs numériques de type quelconque (`char`, `short`, `int`, `long`, `float`, `double`, `long double`).
  `int a=-4; float x=12.5; printf("%d %f\n",ABS(a),ABS(x));`
- La macro-instruction :

```
#define AFF_TAB(tab,n,format)\
{\
  int i;\
  for (i=0;i<n;i++) printf(format,tab[i]);\
  printf("\n");\
}
```

  peut être utilisée pour afficher des éléments d'un tableau contenant des valeurs de n'importe quel type scalaire de base.

```
int ti[3]={1,2,3}; float tf[4]={2.72,3.14,1.41,1.62};
AFF_TAB(ti,3,"%d "); AFF_TAB(tf,4,"%f ");
```

Toutefois, les macro-instructions présentent certaines limites :

- le traitement qu'elles effectuent ne peut raisonnablement pas être trop complexe ;
- l'interprétation des messages d'erreur du compilateur peut être rendue difficile ;
- de mauvaises utilisations des macro-instructions (instructions dans les paramètres effectifs par exemple) peuvent provoquer des erreurs difficiles à localiser.

12.3 Utilisation des pointeurs génériques

Les pointeurs génériques permettent d'écrire des fonctions qui peuvent être appelées avec des paramètres effectifs de type inconnu *a priori*, car un paramètre formel de type `void *`, comme une variable de type `void *`, peut recevoir comme valeur n'importe quelle adresse.

12.3.1 Fonctions retournant une valeur de type `void *`

Ces fonctions effectuent un traitement sur une zone mémoire et retourne l'adresse de début de cette zone qui peut être récupérée dans un pointeur de type quelconque. Ces fonctions doivent tenir compte des éventuelles contrainte d'alignement en mémoire.
Exemple :

```
char *ch;
struct point { double x,y; } *p;
ch=malloc(10);
p=malloc(20*sizeof(struct point));
```

12.3.2 Fonctions à paramètres de type `void *`

Il s'agit de fonctions dont un paramètre formel au moins est de type `void *`.
Exemple : dans le programme suivant, la fonction `echange` permet d'échanger les valeurs pointées par deux pointeurs de même type, quel que soit ce type :

echange.c

```
1  #include <stdio.h>
2
3  void echange(void *p1, void *p2, size_t taille)
4  {
5    char aux;
6    size_t i;
7    for (i=0;i<taille;i++)
8    {
9      aux=((char *)p2)[i];
10     ((char *)p2)[i]=((char *)p1)[i];
11     ((char *)p1)[i]=aux;
12   }
13 }
14
15 #define T 8
16
17 int main(void)
18 {
19   int i1=7,i2=5;
20   char ch1[T]="OK",ch2[T]="Va bene";
21   printf("Avant échange : i1==%d i2==%d. ",i1,i2);
22   echange(&i1,&i2,sizeof(int));
23   printf("Après échange : i1==%d i2==%d.\n",i1,i2);
24   printf("Avant échange : ch1==\"%s\" ch2==\"%s\". ",ch1,ch2);
25   echange(ch1,ch2,T);
26   printf("Après échange : ch1==\"%s\" ch2==\"%s\".\n",ch1,ch2);
27   return 0;
28 }
```

Exemple d'utilisation des pointeurs génériques.

produit sur l'écran l'affichage suivant :

```
Avant échange : i1==7 i2==5. Après échange : i1==5 i2==7.
Avant échange : ch1=="OK" ch2=="Va bene". Après échange : ch1=="Va bene" ch2=="OK".
```

12.4 Utilisation des pointeurs de fonctions

Les pointeurs de fonction permettent d'écrire des fonctions qui appliquent une fonction inconnue *a priori* à des objets dont le type est lui aussi inconnu *a priori*.
Exemple : dans le programme suivant, la fonction `applique` applique la fonction, qui lui est passée en paramètre, aux éléments d'un tableau dont elle ignore le type mais dont la taille et le nombre lui sont passés en paramètres.

affiche.c

```
1  #include <stdio.h>
2
3  struct point {double x,y; };
4
5  void applique(void *tab, int n, size_t taille, void (*pf)(void *))
6  {
7    char *p,*fin;
8    p=tab;
9    fin=(char *)tab+n*taille;
10   while (p<fin)
11   {
12     (*pf)(p);
13     p+=taille;
14   }
15 }
16
17 void affiche_point(void *p)
18 {
19   printf("(%f,%f) ",((struct point *)p)->x,((struct point *)p)->y);
20 }
21
22 void affiche_int(void *p)
23 {
24   printf("%d ",*((int *)p));
25 }
26
27 int main(void)
28 {
29   int ti[3]={1,2,3};
30   struct point tp[2]={{1.23,4.56},{7.89,1.01}};
31   applique(ti,3,sizeof(int),&affiche_int); printf("\n");
32   applique(tp,2,sizeof(struct point),&affiche_point); printf("\n");
33   return 0;
34 }
```

Exemple d'utilisation des pointeurs de fonctions.

L'exécution du programme précédent produit sur l'écran l'affichage suivant :

```
1 2 3
(1.230000,4.560000) (7.890000,1.010000)
```

12.5 Utilisation des fonctions à nombre variable de paramètres

La directive `#include <stdarg.h>` permet d'écrire des fonctions à nombre variable de paramètres.
L'en-tête des fonctions à nombre variable de paramètres est de la forme :

type_retourné identificateur(*déclaration_paramètres_fixes*, ...)

Trois macros permettent de gérer l'accès aux paramètres variables :

- `void va_start(va_list` *liste*, *dernier*`)`
  permet de récupérer la liste des paramètres variables dans une variable de type `va_list`. L'argument *dernier* doit être l'identificateur du dernier (le plus à droite) paramètre fixe de la fonction.
- *type* `va_arg(va_list` *liste*, *type*`)`
  retourne la valeur du paramètre courant de la liste de paramètres variables *liste* et avance d'une position dans la liste (le suivant devient le courant). L'argument *type* est le type du paramètre courant. L'argument *liste* doit avoir été préalablement initialisé par `va_start`. S'il n'y a plus de paramètre disponible dans *liste*, ou si *type* est incorrect, le comportement du programme est indéterminé.
- `void va_end(va_list` *liste*`)`
  permet de préciser la fin de l'utilisation de la liste de paramètres variables *liste*.

Exemple :

```
#include <stdarg.h>
...
double moy(int n, double x, ...)
{
  double m=x;
  int i;
  va_list params;

  va_start(params,x);
  for (i=1;i<n;i++)
    m+=va_arg(params,double);
  m/=n;
  va_end(params);
  return(m);
}
...
printf("%f %f\n",moy(2,3.0,4.0),moy(3,1.5,2.0,2.5));
...
```

Exemple de transmission d'une liste de paramètres variables :

```
#include <stdarg.h>
...
double somme(int n, double x, va_list params)
{
  double s=x;
  int i;

  for (i=1;i<n;i++)
    s+=va_arg(params,double);
```

```
  return(s);
}

double moy(int n, double x, ...)
{
  double m;
  va_list params;

  va_start(params,x);
  m=somme(n,x,params)/n;
  va_end(params);
  return(m);
}
```

Il est possible d'utiliser les fonctions à nombre variable de paramètres pour avoir un modèle plus général des fonctions que l'on passe en paramètres.
Exemple :
Dans le programme suivant, la fonction `applique` du programme `affiche.c` a été modifiée de telle sorte que l'on puisse y passer en paramètre des fonctions à nombre variable de paramètres. Elles peuvent avoir d'autres paramètres formels en plus du paramètre de type `void *` qui est obligatoire. Dans le corps de ces fonctions, ces paramètres formels sont accessibles grâce à la macro `va_arg`. Les paramètres effectifs correspondants sont passés comme derniers paramètres effectifs de la fonction `applique`.

Le programme :

augmente.c

```
1   #include <stdio.h>
2   #include <stdarg.h>
3
4   struct point {double x,y; };
5
6   void applique(void *tab, int n, size_t taille, void (*pf)(void *, va_list),...)
7   {
8     char *p,*fin;
9     va_list params;
10    p=tab;
11    fin=(char *)tab+n*taille;
12    while (p<fin)
13    {
14      va_start(params,pf);
15      (*pf)(p,params);
16      va_end(params);
17      p+=taille;
18    }
19  }
20
21  void augmente_point(void *p, va_list params)
22  {
23    double increment=va_arg(params,double);
24   ((struct point *)p)->x+=increment;
25   ((struct point *)p)->y+=increment;
26  }
27
28  void affiche_point(void *p, va_list params)
29  {
30    printf("(%f,%f) ",((struct point *)p)->x,((struct point *)p)->y);
31  }
```



```
32
33  int main(void)
34  {
35    struct point tp[2]={{1.23,4.56},{7.89,1.01}};
36    applique(tp,2,sizeof(struct point),&affiche_point); printf("\n");
37    applique(tp,2,sizeof(struct point),&augmente_point,0.1);
38    applique(tp,2,sizeof(struct point),&affiche_point); printf("\n");
39    return 0;
40  }
```

Exemple d'utilisation des fonctions à nombre variable de paramètres.

produit sur l'écran l'affichage suivant :

```
(1.230000,4.560000) (7.890000,1.010000)
(1.330000,4.660000) (7.990000,1.110000)
```