Université Toulouse III – Paul Sabatier
Département Informatique

Licence Informatique
Licence MIASHS
Semestre 2

Programmation en C

Alain CROUZIL

Département
d'Informatique

Faculté
des Sciences
et d'Ingénierie

Table des matières

Avant-propos

Objectif

Ce support est un complément aux séances de cours et à leurs diaporamas. Il ne les remplace pas.

Il n'est pas complet car, lors des séances de cours, vous seront présentées des notions et vous seront données des explications, des exemples et des illustrations qui ne figurent pas dans ce support.

Par ailleurs, il ne couvre pas tous les aspects de la programmation en langage C. Par exemple, la gestion dynamique de la mémoire et la généricité ne vous seront présentées qu'en deuxième année de licence.

Norme

Les caractéristiques du langage qui sont décrites dans ce support correspondent à la norme C90. Parmi les évolutions plus récentes du langage (normes C99 et C11), on peut noter que sont très souvent utilisés :

- les commentaires de fin de ligne qui commencent par un double slash et se terminent à la fin de la ligne :
```
// Une ligne de commentaire */
int n; // Nombre de valeurs
```
- la possibilité de ne pas regrouper les instructions de déclaration en début de bloc :
```
int a;
a=12;
int b;
b=2*a;
```
- la possibilité de déclarer une variable dans la partie initialisation d'une boucle `for` :
```
for(int i = 0; i < 10; i++)
  Tab[i]=0;
```

Chapitre 1

ÉLÉMENTS DE BASE DU LANGAGE

1.1 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 par l'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 :

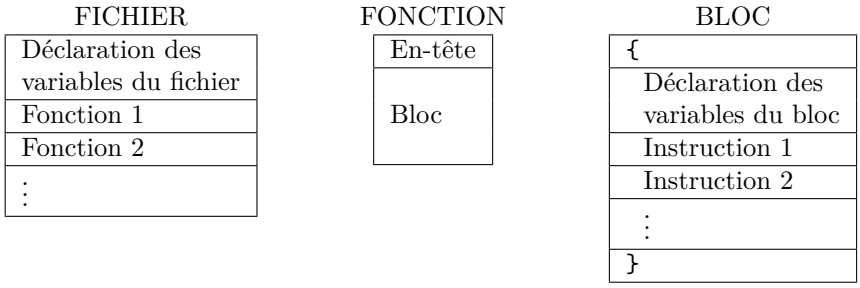
```
ma/*prog...*/in(void)
```

n'est pas permis.

1.2 Structure d'un programme

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 */
```

Il est fréquent de 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=i+1;
27   }
28   return 0;
29 }
```

Affichage des puissances successives de 2.

1.3 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 1.1 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

TABLE 1.1 – Mots-clés du langage C.

1.4 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.

1.4.1 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`

1.4.2 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`

1.4.3 Constantes caractères

- Caractères « imprimables » (ou « éditables ») : `'A'`, `'4'`, `'0'`
- Caractères disposant d'une « séquence d'échappement » (notation symbolique) :

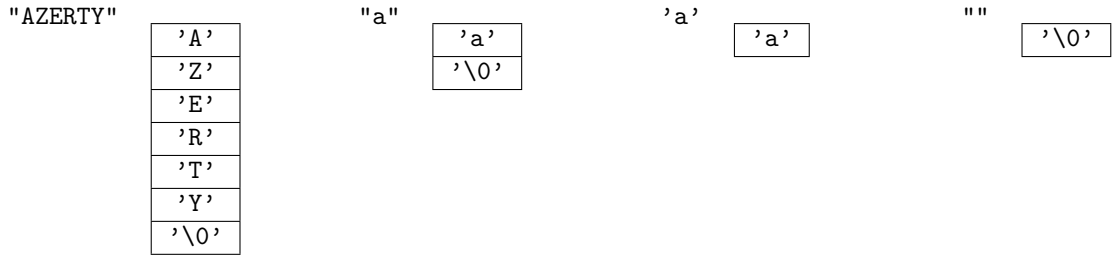
Notation	Dénomination	Notation	Dénomination	Notation	Dénomination
`'\n'`	saut de ligne (*new line*)	`'\f'`	saut de page	`'\?'`	point d'interrogation
`'\t'`	tabulation	`'\\'`	anti-slash	`'\a'`	bip
`'\b'`	retour arrière (*backspace*)	`'\''`	apostrophe	`'\v'`	tabulation verticale
`'\r'`	retour chariot	`'\"'`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

1.4.4 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'` :



1.5 Variables et types

1.5.1 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 :

$$\texttt{type ident}_1\texttt{,ident}_2\texttt{,...,ident}_n\texttt{;}$$

1.5.2 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 1.2. La norme impose en outre la règle suivante :

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

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 1.2.

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

∗. Cela dépend de l'implémentation.

TABLE 1.2 – Taille des types de base.

1.5.3 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.

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

1.5.4 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}};`

1.5.5 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;`

1.5.6 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;`

1.5.7 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)
  ...
  ```

1.6 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`, la valeur de `n` est convertie en `double` et le résultat est de type `double`.

1.6.1 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;`

1.6.2 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 1.3 résume les priorités de tous les opérateurs.

1.6.3 Opérateurs relationnels

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

1.6.4 Opérateurs logiques

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

1.6.5 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).

1.6.6 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 */
```

1.6.7 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`

1.6.8 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...

1.6.9 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 (la valeur de `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.

1.6.10 Priorités des opérateurs

Le tableau 1.3 montre les priorités par ordre décroissant des différents opérateurs (certains seront présentés plus loin) ainsi que les sens d'évaluation (gauche $\longrightarrow$ droite ou gauche $\longleftarrow$ droite).

1.7 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.

1.7.1 Affectation

variable `=` *expression*`;`

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

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

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

TABLE 1.3 – Priorités décroissantes des opérateurs.

1.7.2 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.

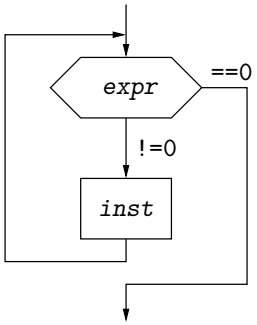
Attention aux opérateurs
`if (r=n%d) ...` peut se « traduire » par : si `r` (qui reçoit `n` *modulo* `d`) est différent de `0` ...
`if (r==n%d) ...` peut se « traduire » par : si `r` est égal à `n` *modulo* `d` ...

1.7.3 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 :



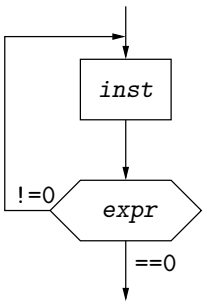

```
a=0;
while (a<6)
{
  printf("%d ",a);
  a+=2;
}
```

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 ` .

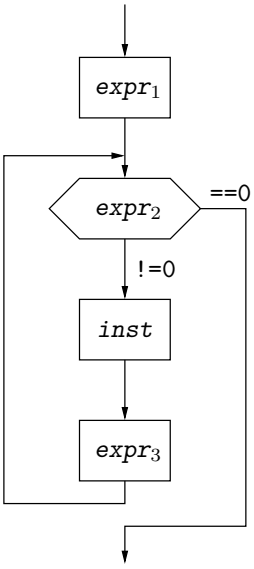
Remarques :

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

Boucle `for`

`for (`[*expr*$_1$]`;`[*expr*$_2$]`;` [*expr*$_3$]`)` *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)`

1.7.4 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`

Chapitre 2

FONCTIONS

2.1 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.

2.2 Définition d'une fonction

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

2.3 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");`

2.4 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 4).

2.5 Déclaration d'une fonction

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

type_retourné nom_fonction(*type*$_1$ *ident*$_1$,*type*$_2$ *ident*$_2$,...,*type*$_n$ *ident*$_n$)

ou bien

type_retourné nom_fonction(*type*$_1$,*type*$_2$,...,*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).

2.6 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 3

ENTRÉES-SORTIES

3.1 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 3.7).

3.2 Écritures sur `stdout`

3.2.1 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);`

3.2.2 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)

3.3 Lectures sur `stdin`

3.3.1 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();`

3.3.2 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 réel

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);`

`print("'%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.

3.4 Entrées-sorties de chaînes de caractères

3.4.1 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`).

3.4.2 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`↩

3.4.3 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.

3.5 Entrées-sorties sur les chaînes de caractères

3.5.1 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.

3.5.2 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`.

3.6 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 */
  ...
```

3.7 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.

3.7.1 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.

3.7.2 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 4

POINTEURS

4.1 Introduction

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

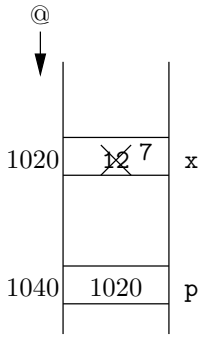
Déclaration : *type_pointé* * *identificateur*;

4.2 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 » $\Longleftrightarrow$ « p contient l'adresse de x » $\Longleftrightarrow$ « la valeur de la variable p est l'adresse de x ».

4.3 Arithmétique des pointeurs

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

4.4 Comparaisons de pointeurs

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

4.5 Pointeur « nul »

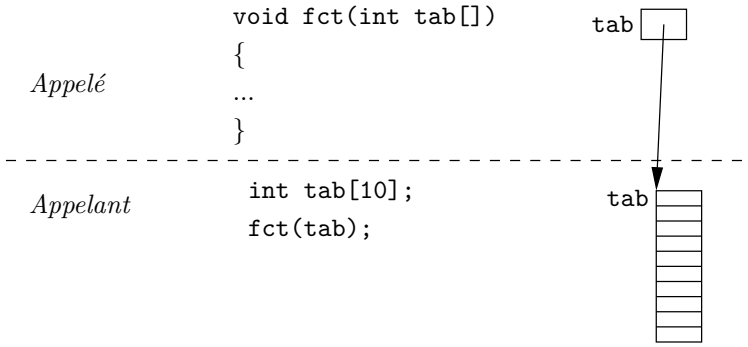
Constante symbolique `NULL` valant 0 qui désigne une adresse mémoire qui n'existe pas. La constante `NULL` est définie dans `stdio.h`, `stdlib.h` et `stddef.h`. Après l'instruction `int *p=NULL;` on dit que `p` pointe sur « rien ».

4.6 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 :**



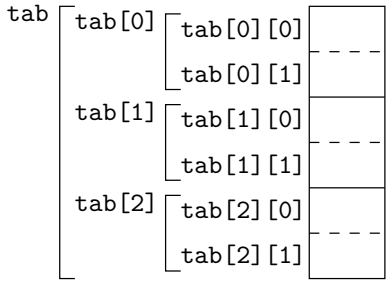
Un paramètre formel de type tableau est en fait un pointeur (vers le premier élément du tableau effectivement passé à la fonction). Dans l'exemple précédent, on aurait pu aussi écrire : `void fct(int *tab)`

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 $\texttt{i}^{\text{ème}}$ ligne.

4.7 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 :

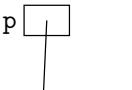
```
struct sommet
{
  float x;
  float y;
  struct sommet *pSuivant;
};
```

4.8 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 :

Appelé

```
void incr(int *p)
{
  *p = *p + 1;
}
```

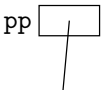


Appelant

```
int a=2;
incr(&a);
```

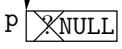
Appelé

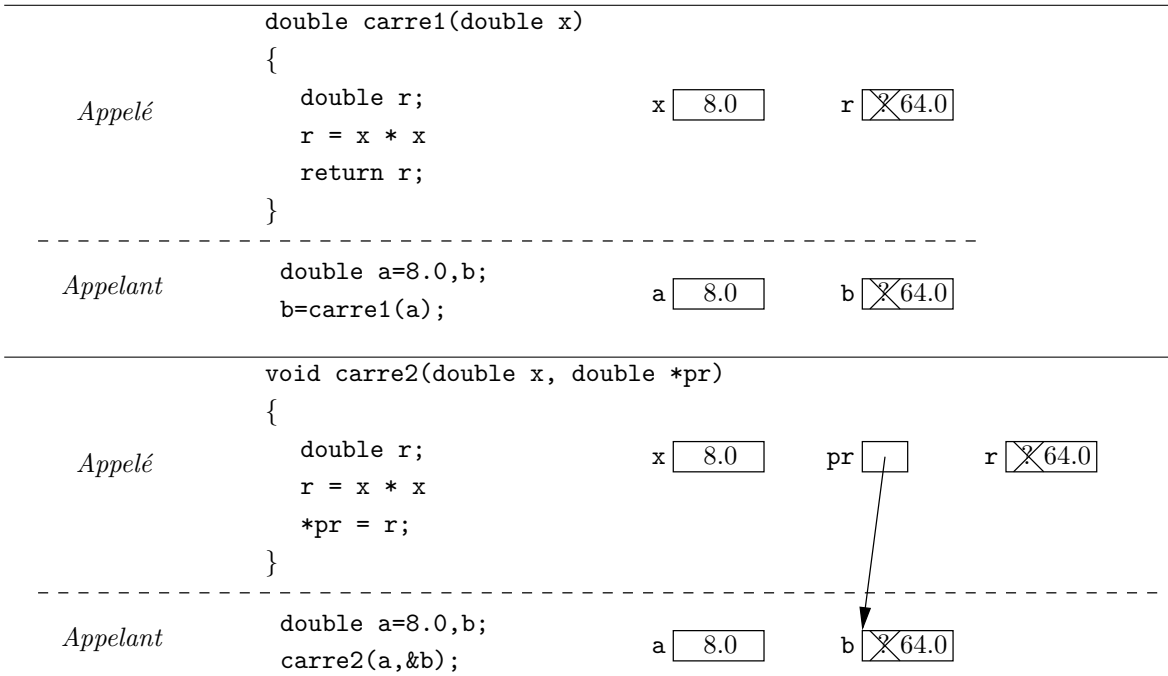
```
void init(int **pp)
{
  *pp = NULL;
}
```



Appelant

```
int *p;
init(&p);
```




double carre1(double x)
{
double r;
r = x * x
return r;
}
Appelé
x 8.0
r ?64.0
double a=8.0,b;
b=carre1(a);
Appelant
a 8.0
b ?64.0
void carre2(double x, double *pr)
{
double r;
r = x * x
*pr = r;
}
Appelé
x 8.0
pr
r ?64.0
double a=8.0,b;
carre2(a,&b);
Appelant
a 8.0
b ?64.0

Chapitre 5

UTILISATION DU PRÉPROCESSEUR

5.1 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.

5.2 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.

5.3 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*	`#ifndef` *symbole*
texte	*texte*
`#endif`	`#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>`

5.4 Directive de substitution symbolique

5.4.1 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 »).

5.4.2 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");\
}
```

Attention : les macro-instructions recelant de nombreux pièges, elles doivent être utilisées avec beaucoup de précautions...