Chapitre 1

Les bases de la programmation en C

1.1 Historique

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language* [6]. Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

1.2 La compilation

Le C est un langage *compilé* (par opposition aux langages interprétés). Cela signifie qu'un programme C est décrit par un fichier texte, appelé *fichier source*. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé *compilateur*. La compilation se décompose en fait en 4 phases successives :

1. **Le traitement par le préprocesseur :** le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).

2. **La compilation :** la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.

3. **L'assemblage :** cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé *fichier objet*.

4. **L'édition de liens :** un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit *exécutable*.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par `.c`, les fichiers prétraités par le préprocesseur par `.i`, les fichiers assembleur par `.s`, et les fichiers objet par `.o`. Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe `.a`.

Le compilateur C sous UNIX s'appelle `cc`. On utilisera de préférence le compilateur `gcc` du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, `gcc` active toutes les étapes de la compilation. On le lance par la commande

```
gcc [options] fichier.c [-llibrairies]
```

Par défaut, le fichier exécutable s'appelle `a.out`. Le nom de l'exécutable peut être modifié à l'aide de l'option `-o`.

Les éventuelles librairies sont déclarées par la chaîne `-l`*`librairie`*. Dans ce cas, le système recherche le fichier `lib`*`librairie`*`.a` dans le répertoire contenant les librairies pré-compilées (généralement `/usr/lib/`). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie `-lm`. Le fichier objet correspondant est `libm.a`. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option `-L`.

Les options les plus importantes du compilateur `gcc` sont les suivantes :

`-c` : supprime l'édition de liens ; produit un fichier objet.

`-E` : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard).

`-g` : produit des informations symboliques nécessaires au débogueur.

`-I`*`nom-de-répertoire`* : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant).

`-L`*`nom-de-répertoire`* : spécifie le répertoire dans lequel doivent être recherchées les librairies précompilées (en plus du répertoire usuel).

`-o` *`nom-de-fichier`* : spécifie le nom du fichier produit. Par défaut, le exécutable fichier s'appelle `a.out`.

`-O`, `-O1`, `-O2`, `-O3` : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : `-O1` (similaire à `-O`) correspond à une faible optimisation, `-O3` à l'optimisation maximale.

`-S` : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur.

`-v` : imprime la liste des commandes exécutées par les différentes étapes de la compilation.

`-W` : imprime des messages d'avertissement (warning) supplémentaires.

`-Wall` : imprime tous les messages d'avertissement.

Pour plus de détails sur `gcc`, on peut consulter le chapitre 4 de [8].

1.3 Les composants élémentaires du C

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- les identificateurs,
- les mots-clefs,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

1.3.1 Les identificateurs

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction,
- un type défini par `typedef`, `struct`, `union` ou `enum`,
- une étiquette.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le "blanc souligné" (_).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Par exemple, `var1`, `tab_23` ou `_deb` sont des identificateurs valides ; par contre, `1i` et `i:j` ne le sont pas. Il est cependant déconseillé d'utiliser _ comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C.

Les majuscules et minuscules sont différenciées.

Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

1.3.2 Les mots-clefs

Un certain nombre de mots, appelés *mots-clefs*, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :

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

que l'on peut ranger en catégories

– les spécificateurs de stockage

  `auto   register   static   extern   typedef`

– les spécificateurs de type

  `char   double   enum   float   int   long   short   signed   struct`
  `union   unsigned   void`

– les qualificateurs de type

  `const   volatile`

– les instructions de contrôle

  `break   case   continue   default   do   else   for   goto   if`
  `switch   while`

– divers

  `return   sizeof`

1.3.3 Les commentaires

Un commentaire débute par `/*` et se termine par `*/`. Par exemple,

```
/*  Ceci est un commentaire */
```

On ne peut pas imbriquer des commentaires. Quand on met en commentaire un morceau de programme, il faut donc veiller à ce que celui-ci ne contienne pas de commentaire.

1.4 Structure d'un programme C

Une *expression* est une suite de composants élémentaires syntaxiquement correcte, par exemple
`x = 0`
ou bien

```
(i >= 0) && (i < 10) && (p[i] != 0)
```

Une *instruction* est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression". Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former une *instruction composée* ou *bloc* qui est syntaxiquement équivalent à une instruction. Par exemple,

```
if (x != 0)
{
  z = y / x;
  t = y % x;
}
```

Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule est une *déclaration*. Par exemple,

```
int a;
int b = 1, c;
double x = 2.38e4;
char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Un programme C se présente de la façon suivante :

```
[ directives au préprocesseur]
[ déclarations de variables externes]
[ fonctions secondaires]

main()
{
   déclarations de variables internes
   instructions
}
```

La fonction principale `main` peut avoir des paramètres formels. On supposera dans un premier temps que la fonction `main` n'a pas de valeur de retour. Ceci est toléré par le compilateur mais produit un message d'avertissement quand on utilise l'option `-Wall` de `gcc` (*cf.* page 67).

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale. Une fonction secondaire peut se décrire de la manière suivante :

```
 type ma_fonction (  arguments )
{
   déclarations de variables internes
   instructions
}
```

Cette fonction retournera un objet dont le type sera *`type`* (à l'aide d'une instruction comme `return` *`objet`*`;`). Les *`arguments`* de la fonction obéissent à une syntaxe voisine de celle des déclarations : on met en argument de la fonction une suite d'expressions *`type`* `objet` séparées par des virgules. Par exemple, la fonction secondaire suivante calcule le produit de deux entiers :

```
int produit(int a, int b)
{
  int resultat;

  resultat = a * b;
  return(resultat);
}
```

1.5 Les types prédéfinis

Le C est un langage *typé*. Cela signifie en particulier que toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

La mémoire de l'ordinateur se décompose en une suite continue d'octets. Chaque octet de la mémoire est caractérisé par son *adresse*, qui est un entier. Deux octets contigus en mémoire ont des adresses qui diffèrent d'une unité. Quand une variable est définie, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type.

La taille mémoire correspondant aux différents types dépend des compilateurs ; toutefois, la norme ANSI spécifie un certain nombre de contraintes.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

```
char
int
float  double
short  long    unsigned
```

1.5.1 Le type caractère

Le mot-clef `char` désigne un objet de type caractère. Un `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. La plupart du temps, un objet de type `char` est codé sur un octet ; c'est l'objet le plus élémentaire en C. Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits). La plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits), dont les 128 premiers caractères correspondent aux caractères ASCII. Les 128 derniers caractères (codés sur 8 bits) sont utilisés pour les caractères propres aux différentes langues. La version ISO-8859-1 (aussi appelée ISO-LATIN-1) est utilisée pour les langues d'Europe occidentale. Ainsi, le caractère de code 232 est le è, le caractère 233 correspond au é... Pour plus de détails sur l'historique du codage des caractères pour les différentes langues ainsi que sur la norme UNICODE (sur 16 bits, qui permet de coder les caractères pour toutes les langues) et sur la norme ISO/IEC-10646 (sur 32 bits, ce qui permet d'ajouter les caractères anciens), consulter l'article de J. André et M. Goossens [1].

	déc.	oct.	hex.		déc.	oct.	hex.		déc.	oct.	hex.
	32	40	20	@	64	100	40	‘	96	140	60
!	33	41	21	A	65	101	41	a	97	141	61
”	34	42	22	B	66	102	42	b	98	142	62
#	35	43	23	C	67	103	43	c	99	143	63
$	36	44	24	D	68	104	44	d	100	144	64
%	37	45	25	E	69	105	45	e	101	145	65
&	38	46	26	F	70	106	46	f	102	146	66
’	39	47	27	G	71	107	47	g	103	147	67
(	40	50	28	H	72	110	48	h	104	150	68
)	41	51	29	I	73	111	49	i	105	151	69
*	42	52	2a	J	74	112	4a	j	106	152	6a
+	43	53	2b	K	75	113	4b	k	107	153	6b
,	44	54	2c	L	76	114	4c	l	108	154	6c
-	45	55	2d	M	77	115	4d	m	109	155	6d
.	46	56	2e	N	78	116	4e	n	110	156	6e
/	47	57	2f	O	79	117	4f	o	111	157	6f
0	48	60	30	P	80	120	50	p	112	160	70
1	49	61	31	Q	81	121	51	q	113	161	71
2	50	62	32	R	82	122	52	r	114	162	72
3	51	63	33	S	83	123	53	s	115	163	73
4	52	64	34	T	84	124	54	t	116	164	74
5	53	65	35	U	85	125	55	u	117	165	75
6	54	66	36	V	86	126	56	v	118	166	76
7	55	67	37	W	87	127	57	w	119	167	77
8	56	70	38	X	88	130	58	x	120	170	78
9	57	71	39	Y	89	131	59	y	121	171	79
:	58	72	3a	Z	90	132	5a	z	122	172	7a
;	59	73	3b	[	91	133	5b	{	123	173	7b
<	60	74	3c	\	92	134	5c	|	124	174	7c
=	61	75	3d	]	93	135	5d	}	125	175	7d
>	62	76	3e	^	94	136	5e	~	126	176	7e
?	63	77	3f	_	95	137	5f	DEL	127	177	7f

TAB. 1.1 – *Codes ASCII des caractères imprimables*

Une des particularités du type `char` en C est qu'il peut être assimilé à un entier : tout objet de type `char` peut être utilisé dans une expression qui utilise des objets de type entier. Par exemple, si `c` est de type `char`, l'expression `c + 1` est valide. Elle désigne le caractère suivant dans le code ASCII. La table de la page 15 donne le code ASCII (en décimal, en octal et en hexadécimal) des caractères imprimables. Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
  char c = 'A';
```

```
  printf("%c", c + 1);
}
```

Suivant les implémentations, le type `char` est signé ou non. En cas de doute, il vaut mieux préciser `unsigned char` ou `signed char`. Notons que tous les caractères imprimables sont positifs.

1.5.2 Les types entiers

Le mot-clef désignant le type entier est `int`. Un objet de type `int` est représenté par un mot "naturel" de la machine utilisée, 32 bits pour un DEC alpha ou un PC Intel.

Le type `int` peut être précédé d'un attribut de précision (`short` ou `long`) et/ou d'un attribut de représentation (`unsigned`). Un objet de type `short int` a au moins la taille d'un `char` et au plus la taille d'un `int`. En général, un `short int` est codé sur 16 bits. Un objet de type `long int` a au moins la taille d'un `int` (64 bits sur un DEC alpha, 32 bits sur un PC Intel).

	DEC Alpha	PC Intel (Linux)	
`char`	8 bits	8 bits	caractère
`short`	16 bits	16 bits	entier court
`int`	32 bits	32 bits	entier
`long`	64 bits	32 bits	entier long
`long long`	n.i.	64 bits	entier long (non ANSI)

TAB. 1.2 – *Les types entiers*

Le bit de poids fort d'un entier est son signe. Un entier positif est donc représenté en mémoire par la suite de 32 bits dont le bit de poids fort vaut 0 et les 31 autres bits correspondent à la décomposition de l'entier en base 2. Par exemple, pour des objets de type `char` (8 bits), l'entier positif 12 sera représenté en mémoire par 00001100. Un entier négatif est, lui, représenté par une suite de 32 bits dont le bit de poids fort vaut 1 et les 31 autres bits correspondent à la valeur absolue de l'entier représentée suivant la technique dite du "complément à 2". Cela signifie que l'on exprime la valeur absolue de l'entier sous forme binaire, que l'on prend le complémentaire bit-à-bit de cette valeur et que l'on ajoute 1 au résultat. Ainsi, pour des objets de type `signed char` (8 bits), -1 sera représenté par 11111111, -2 par 11111110, -12 par par 11110100. Un `int` peut donc représenter un entier entre -2^{31} et $(2^{31}-1)$. L'attribut `unsigned` spécifie que l'entier n'a pas de signe. Un `unsigned int` peut donc représenter un entier entre 0 et $(2^{32}-1)$. Sur un DEC alpha, on utilisera donc un des types suivants en fonction de la taille des données à stocker :

`signed char`	$[-2^7; 2^7[$
`unsigned char`	$[0; 2^8[$
`short int`	$[-2^{15}; 2^{15}[$
`unsigned short int`	$[0; 2^{16}[$
`int`	$[-2^{31}; 2^{31}[$
`unsigned int`	$[0; 2^{32}[$
`long int` (DEC alpha)	$[-2^{63}; 2^{63}[$
`unsigned long int` (DEC alpha)	$[0; 2^{64}[$

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard `limits.h`.

Le mot-clef `sizeof` a pour syntaxe

`sizeof(`*`expression`*`)`

où *`expression`* est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet. Par exemple

```
unsigned short x;

taille = sizeof(unsigned short);
taille = sizeof(x);
```

Dans les deux cas, `taille` vaudra 4.

Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de `limits.h` ou le résultat obtenu en appliquant l'opérateur `sizeof`.

1.5.3 Les types flottants

Les types `float`, `double` et `long double` servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

	DEC Alpha	PC Intel	
`float`	32 bits	32 bits	flottant
`double`	64 bits	64 bits	flottant double précision
`long double`	64 bits	128 bits	flottant quadruple précision

Tab. 1.3 – *Les types flottants*

Les flottants sont généralement stockés en mémoire sous la représentation de la virgule flottante normalisée. On écrit le nombre sous la forme “signe $0, \text{mantisse } B^{\text{exposant}}$”. En général, $B = 2$. Le digit de poids fort de la mantisse n'est jamais nul.

Un flottant est donc représenté par une suite de bits dont le bit de poids fort correspond au signe du nombre. Le champ du milieu correspond à la représentation binaire de l'exposant alors que les bits de poids faible servent à représenter la mantisse.

1.6 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

1.6.1 Les constantes entières

Une constante entière peut être représentée de 3 manières différentes suivant la base dans laquelle elle est écrite :

- **décimale :** par exemple, `0` et `2437348` sont des constantes entières décimales.
- **octale :** la représentation octale d'un entier correspond à sa décomposition en base 8. Les constantes octales doivent commencer par un zéro. Par exemple, les représentations octales des entiers 0 et 255 sont respectivement `00` et `0377`.
- **hexadécimale :** la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de `a` à `f` sont utilisées pour représenter les nombres de 10 à 15. Les constantes hexadécimales doivent commencer par `0x` ou `0X`. Par exemple, les représentations hexadécimales de 14 et 255 sont respectivement `0xe` et `0xff`.

Par défaut, une constante décimale est représentée avec le format interne le plus court permettant de la représenter parmi les formats des types `int`, `long int` et `unsigned long int` tandis qu'une constante octale ou hexadécimale est représentée avec le format interne le plus court permettant encore de la représenter parmi les formats des types `int`, `unsigned int`, `long int` et `unsigned long int`.

On peut cependant spécifier explicitement le format d'une constante entière en la suffixant par `u` ou `U` pour indiquer qu'elle est non signée, ou en la suffixant par `l` ou `L` pour indiquer qu'elle est de type `long`. Par exemple :

constante	type
`1234`	`int`
`02322`	`int` /* octal */
`0x4D2`	`int` /* hexadécimal */
`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

1.6.2 Les constantes réelles

Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre `e` ou `E` ; il s'agit d'un nombre décimal éventuellement signé.

Par défaut, une constante réelle est représentée avec le format du type `double`. On peut cependant influer sur la représentation interne de la constante en lui ajoutant un des suffixes `f` (indifféremment `F`) ou `l` (indifféremment `L`). Les suffixes `f` et `F` forcent la représentation de la constante sous forme d'un `float`, les suffixes `l` et `L` forcent la représentation sous forme d'un `long double`. Par exemple :

constante	type
`12.34`	`double`
`12.3e-4`	`double`
`12.34F`	`float`
`12.34L`	`long double`

1.6.3 Les constantes caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. `'A'` ou `'$'`). Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par `\\` et `\'`. Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations `\?` et `\"`. Les caractères non imprimables peuvent être désignés par `'\`*`code-octal`*`'` où *`code-octal`* est le code en octal du caractère. On peut aussi écrire `'\x`*`code-hexa`*`'` où *`code-hexa`* est le code en hexadécimal du caractère (*cf.* page 15). Par exemple, `'\33'` et `'\x1b'` désignent le caractère *escape*. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

`\n`	nouvelle ligne	`\r`	retour chariot
`\t`	tabulation horizontale	`\f`	saut de page
`\v`	tabulation verticale	`\a`	signal d'alerte
`\b`	retour arrière		

1.6.4 Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple,

```
"Ceci est une chaîne de caractères"
```

Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple,

```
"ligne 1 \n ligne 2"
```

A l'intérieur d'une chaîne de caractères, le caractère `"` doit être désigné par `\"`. Enfin, le caractère `\` suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple,

```
"ceci est une longue longue  longue longue longue longue longue longue \
chaîne de caractères"
```

1.7 Les opérateurs

1.7.1 L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe `=`. Sa syntaxe est la suivante :

`variable = expression`

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *`expression`* et d'affecter la valeur obtenue à *`variable`*. De plus, cette expression possède une valeur, qui est celle *`expression`*. Ainsi, l'expression `i = 5` vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant

```
main()
{
  int i, j = 2;
  float x = 2.5;
  i = j + x;
  x = x + i;
  printf("\n %f \n",x);
}
```

imprime pour `x` la valeur 6.5 (et non 7), car dans l'instruction `i = j + x;`, l'expression `j + x` a été convertie en entier.

1.7.2 Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire `-` (changement de signe) ainsi que les opérateurs binaires

- `+` addition
- `-` soustraction
- `*` multiplication
- `/` division
- `%` reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

– Contrairement à d'autres langages, le C ne dispose que de la notation `/` pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur `/` produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple,

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

affecte à `x` la valeur 1. Par contre

```
x = 3 / 2.;
```

affecte à `x` la valeur 1.5.

– L'opérateur `%` ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction `pow(x,y)` de la librairie `math.h` pour calculer x^y.

1.7.3 Les opérateurs relationnels

`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type `int` (il n'y a pas de type booléen en C); elle vaut 1 si la condition est vraie, et 0 sinon.

Attention à ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affection `=`. Ainsi, le programme

```
main()
{
  int a = 0;
  int b = 1;
  if (a = b)
    printf("\n a et b sont egaux \n");
  else
    printf("\n a et b sont differents \n");
}
```

imprime à l'écran `a et b sont egaux`!

1.7.4 Les opérateurs logiques booléens

`&&`	et logique
`||`	ou logique
`!`	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un `int` qui vaut 1 si la condition est vraie et 0 sinon.

Dans une expression de type

expression-1 op-1 expression-2 op-2 ...expression-n

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i;
int p[10];

if ((i >= 0) && (i <= 9) && !(p[i] == 0))
...
```

la dernière clause ne sera pas évaluée si `i` n'est pas entre 0 et 9.

1.7.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (`short`, `int` ou `long`), signés ou non.

`&`	et	`|`	ou inclusif
`^`	ou exclusif	`~`	complément à 1
`<<`	décalage à gauche	`>>`	décalage à droite

En pratique, les opérateurs `&`, `|` et `~` consistent à appliquer bit à bit les opérations suivantes

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

L'opérateur unaire `~` change la valeur de chaque bit d'un entier. Le décalage à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (ce qui dépasse disparaît).

Considérons par exemple les entiers `a=77` et `b=23` de type `unsigned char` (*i.e.* 8 bits). En base 2 il s'écrivent respectivement 01001101 et 00010111.

	valeur		
expression	binaire	décimale	
`a`	`01001101`	`77`	
`b`	`00010111`	`23`	
`a & b`	`00000101`	`5`	
`a | b`	`01011111`	`95`	
`a ^ b`	`01011010`	`90`	
`~a`	`10110010`	`178`	
`b << 2`	`01011100`	`92`	multiplication par 4
`b << 5`	`11100000`	`112`	ce qui dépasse disparaît
`b >> 1`	`00001011`	`11`	division entière par 2

1.7.6 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

```
+=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
```

Pour tout opérateur *`op`*, l'expression

`expression-1 op= expression-2`

est équivalente à

`expression-1 = expression-1 op expression-2`

Toutefois, avec l'affection composée, *`expression-1`* n'est évaluée qu'une seule fois.

1.7.7 Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation `++` et de décrémentation `--` s'utilisent aussi bien en suffixe (`i++`) qu'en préfixe (`++i`). Dans les deux cas la variable `i` sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de `i` alors que dans la notation préfixe se sera la nouvelle. Par exemple,

```
int a = 3, b, c;
b = ++a;      /* a et b valent 4 */
c = b++;      /* c vaut 4 et b vaut 5 */
```

1.7.8 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

expression-1, expression-2, ... , expression-n

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple, le programme

```
main()
{
  int a, b;
  b = ((a = 3), (a + 2));
  printf("\n b = %d \n",b);
}
```

imprime `b = 5`.

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie. Par exemple l'instruction composée

```
{
int a=1;
printf("\%d \%d",++a,a);
}
```

(compilée avec gcc) produira la sortie `2 1` sur un PC Intel/Linux et la sortie `2 2` sur un DEC Alpha/OSF1.

1.7.9 L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

condition ? *expression-1* : *expression-2*

Cette expression est égale à *expression-1* si *condition* est satisfaite, et à *expression-2* sinon. Par exemple, l'expression

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre. De même l'instruction

```
m = ((a > b) ? a : b);
```

affecte à `m` le maximum de `a` et de `b`.

1.7.10 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet. On écrit

`(`*type*`)` *objet*

Par exemple,

```
main()
{
  int i = 3, j = 2;
  printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

1.7.11 L'opérateur adresse

L'opérateur d'adresse `&` appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est

`&`*objet*

1.7.12 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est définie par la flèche de la seconde colonne du tableau. On préferera toutefois mettre des parenthèses en cas de doute...

<table>
<tr><th>opérateurs</th><th></th></tr>
<tr><td>() [] -> .</td><td>→</td></tr>
<tr><td>! ~ ++ -- -(unaire) (type) *(indirection) &(adresse) sizeof</td><td>←</td></tr>
<tr><td>* / %</td><td>→</td></tr>
<tr><td>+ -(binaire)</td><td>→</td></tr>
<tr><td><< >></td><td>→</td></tr>
<tr><td>< <= > >=</td><td>→</td></tr>
<tr><td>== !=</td><td>→</td></tr>
<tr><td>&(et bit-à-bit)</td><td>→</td></tr>
<tr><td>^</td><td>→</td></tr>
<tr><td>|</td><td>→</td></tr>
<tr><td>&&</td><td>→</td></tr>
<tr><td>||</td><td>→</td></tr>
<tr><td>? :</td><td>←</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>←</td></tr>
<tr><td>,</td><td>→</td></tr>
</table>

TAB. 1.4 – *Règles de priorité des opérateurs*

Par exemple, les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions. Par exemple, il faut écrire `if ((x ^ y) != 0)`

1.8 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les *instructions de branchement* et les *boucles*. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

1.8.1 Branchement conditionnel `if---else`

La forme la plus générale est celle-ci :

```
if ( expression-1 )
   instruction-1
else if ( expression-2 )
   instruction-2
    ...
else if ( expression-n )
   instruction-n
else
   instruction-∞
```

avec un nombre quelconque de `else if ( ... )`. Le dernier `else` est toujours facultatif. La forme la plus simple est

```
if ( expression )
   instruction
```

Chaque *instruction* peut être un bloc d'instructions.

1.8.2 Branchement multiple `switch`

Sa forme la plus générale est celle-ci :

```
switch ( expression )
  {
  case  constante-1:
     liste d'instructions 1
    break;
  case  constante-2:
     liste d'instructions 2
    break;
    ...
  case  constante-n:
     liste d'instructions n
```

```
    break;
  default:
     liste d'instructions ∞
    break;
  }
```

Si la valeur de *expression* est égale à l'une des *constantes*, la *liste d'instructions* correspondant est exécutée. Sinon la *liste d'instructions* ∞ correspondant à `default` est exécutée. L'instruction `default` est facultative.

1.9 Les boucles

Les *boucles* permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

1.9.1 Boucle `while`

La syntaxe de `while` est la suivante :

```
while ( expression )
  instruction
```

Tant que *expression* est vérifiée (*i.e.*, non nulle), *instruction* est exécutée. Si *expression* est nulle au départ, *instruction* ne sera jamais exécutée. *instruction* peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
i = 1;
while (i < 10)
  {
    printf("\n i = %d",i);
    i++;
  }
```

1.9.2 Boucle `do---while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do---while`. Sa syntaxe est

```
do
    instruction
while ( expression );
```

Ici, *instruction* sera exécutée tant que *expression* est non nulle. Cela signifie donc que *instruction* est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;

do
```

```
  {
    printf("\n Entrez un entier entre 1 et 10 : ");
    scanf("%d",&a);
  }
while ((a <= 0) || (a > 10));
```

1.9.3 Boucle `for`

La syntaxe de `for` est :

```
for ( expr 1 ; expr 2 ; expr 3)
   instruction
```

Une version équivalente plus intuitive est :

```
 expr 1;
 while ( expr 2 )
   {
     instruction
     expr 3;
   }
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)
  printf("\n i = %d",i);
```

A la fin de cette boucle, `i` vaudra 10. Les trois expressions utilisées dans une boucle `for` peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
  fact *= i;
printf("%d ! = %d \n",n,fact);
```

On peut également insérer l'instruction `fact *= i;` dans la boucle `for` ce qui donne :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; fact *= i, i++);
printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

1.10 Les instructions de branchement non conditionnel

1.10.1 Branchement non conditionnel `break`

On a vu le rôle de l'instruction `break;` au sein d'une instruction de branchement multiple `switch`. L'instruction `break` peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      printf("i = %d\n",i);
      if (i == 3)
        break;
    }
  printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

1.10.2 Branchement non conditionnel `continue`

L'instruction `continue` permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
    {
      if (i == 3)
        continue;
      printf("i = %d\n",i);
    }
  printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime

```
i = 0
```

```
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle = 5
```

1.10.3 Branchement non conditionnel `goto`

L'instruction `goto` permet d'effectuer un saut jusqu'à l'instruction ***etiquette*** correspondant. Elle est à proscrire de tout programme C digne de ce nom.

1.11 Les fonctions d'entrées-sorties classiques

Il s'agit des fonctions de la librairie standard `stdio.h` utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie `stdio.h` par la directive au préprocesseur

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

n'est pas nécessaire pour utiliser `printf` et `scanf`.

1.11.1 La fonction d'écriture `printf`

La fonction `printf` est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est

`printf("`*`chaîne de contrôle `*`",`*`expression-1`*`, ..., `*`expression-n`*`);`

La ***chaîne de contrôle*** contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère `%`, suivi d'un caractère désignant le format d'impression. Les formats d'impression en C sont donnés à la table 1.5.

En plus du caractère donnant le type des données, on peut éventuellemnt préciser certains paramètres du format d'impression, qui sont spécifiés entre le `%` et le caractère de conversion dans l'ordre suivant :

- largeur minimale du champ d'impression : `%10d` spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier. Par défaut, la donnée sera cadrée à droite du champ. Le signe `-` avant le format signifie que la donnée sera cadrée à gauche du champ (`%-10d`).
- précision : `%.12f` signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule. De même `%10.2f` signifie que l'on réserve 12 caractères (incluant le caractère `.`) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule. Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule. Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés : `%30.4s` signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

format	conversion en	écriture
`%d`	`int`	décimale signée
`%ld`	`long int`	décimale signée
`%u`	`unsigned int`	décimale non signée
`%lu`	`unsigned long int`	décimale non signée
`%o`	`unsigned int`	octale non signée
`%lo`	`unsigned long int`	octale non signée
`%x`	`unsigned int`	hexadécimale non signée
`%lx`	`unsigned long int`	hexadécimale non signée
`%f`	`double`	décimale virgule fixe
`%lf`	`long double`	décimale virgule fixe
`%e`	`double`	décimale notation exponentielle
`%le`	`long double`	décimale notation exponentielle
`%g`	`double`	décimale, représentation la plus courte parmi `%f` et `%e`
`%lg`	`long double`	décimale, représentation la plus courte parmi `%lf` et `%le`
`%c`	`unsigned char`	caractère
`%s`	`char*`	chaîne de caractères

TAB. 1.5 – *Formats d'impression pour la fonction* `printf`

Exemple :

```
#include <stdio.h>
main()
{
  int i = 23674;
  int j = -23674;
  long int k = (1l << 32);
  double x = 1e-8 + 1000;
  char c = 'A';
  char *chaine = "chaine de caracteres";

  printf("impression de i: \n");
  printf("%d \t %u \t %o \t %x",i,i,i,i);
  printf("\nimpression de j: \n");
  printf("%d \t %u \t %o \t %x",j,j,j,j);
  printf("\nimpression de k: \n");
  printf("%d \t  %o \t %x",k,k,k);
  printf("\n%ld \t %lu \t %lo \t %lx",k,k,k,k);
  printf("\nimpression de x: \n");
  printf("%f \t %e \t %g",x,x,x);
  printf("\n%.2f \t %.2e",x,x);
  printf("\n%.20f \t %.20e",x,x);
  printf("\nimpression de c: \n");
  printf("%c \t %d",c,c);
  printf("\nimpression de chaine: \n");
  printf("%s \t %.10s",chaine,chaine);
```

```
  printf("\n");
}
```

Ce programme imprime à l'écran :

```
impression de i:
23674    23674          56172   5c7a
impression de j:
-23674   4294943622     37777721606     ffffa386
impression de k:
0        0      0
4294967296      4294967296      40000000000     100000000
impression de x:
1000.000000     1.000000e+03    1000
1000.00         1.00e+03
1000.00000001000000000000       1.00000000001000000000e+03
impression de c:
A       65
impression de chaine:
chaine de caracteres     chaine de
```

1.11.2 La fonction de saisie `scanf`

La fonction `scanf` permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonctions.

`scanf("`*`chaîne de contrôle`*`",`*`argument-1`*`,...,`*`argument-n`*`)`

La *`chaîne de contrôle`* indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de `\n`). Comme pour `printf`, les conversions de format sont spécifiées par un caractère précédé du signe `%`. Les formats valides pour la fonction `scanf` diffèrent légèrement de ceux de la fonction `printf`.

Les données à entrer au clavier doivent être séparées par des blancs ou des `<RETURN>` sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple `%3s` pour une chaîne de 3 caractères, `%10d` pour un entier qui s'étend sur 10 chiffres, signe inclus.

Exemple :

```
#include <stdio.h>
main()
{
  int i;
  printf("entrez un entier sous forme hexadecimale i = ");
  scanf("%x",&i);
  printf("i = %d\n",i);
}
```

Si on entre au clavier la valeur `1a`, le programme affiche `i = 26`.

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

TAB. 1.6 – *Formats de saisie pour la fonction* `scanf`

1.11.3 Impression et lecture de caractères

Les fonctions `getchar` et `putchar` permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

La fonction `getchar` retourne un `int` correspondant au caractère lu. Pour mettre le caractère lu dans une variable `caractere`, on écrit

```
caractere = getchar();
```

Lorsqu'elle détecte la fin de fichier, elle retourne l'entier `EOF` (End Of File), valeur définie dans la librairie `stdio.h`. En général, la constante `EOF` vaut -1.

La fonction `putchar` écrit *`caractere`* sur la sortie standard :

```
putchar(caractere);
```

Elle retourne un `int` correspondant à l'entier lu ou à la constante `EOF` en cas d'erreur.

Par exemple, le programme suivant lit un fichier et le recopie caractère par caractère à l'écran.

```
#include <stdio.h>
main()
{
```

```
  char c;

  while ((c = getchar()) != EOF)
    putchar(c);
}
```

Pour l'exécuter, il suffit d'utiliser l'opérateur de redirection d'Unix :
programme-executable < nom-fichier

Notons que l'expression `(c = getchar())` dans le programme précédent a pour valeur la valeur de l'expression `getchar()` qui est de type `int`. Le test `(c = getchar()) != EOF` compare donc bien deux objets de type `int` (signés).

Ce n'est par contre pas le cas dans le programme suivant :

```
#include <stdio.h>
main()
{
  char c;
  do
    {
      c = getchar();
      if (c != EOF)
        putchar(c);
    }
  while (c != EOF);
}
```

Ici, le test `c != EOF` compare un objet de type `char` et la constante `EOF` qui vaut -1. Si le type `char` est non signé par défaut, cette condition est donc toujours vérifiée. Si le type `char` est signé, alors le caractère de code 255, ÿ, sera converti en l'entier -1. La rencontre du caractère ÿ sera donc interprétée comme une fin de fichier. Il est donc recommandé de déclarer de type `int` (et non `char`) une variable destinée à recevoir un caractère lu par `getchar` afin de permettre la détection de fin de fichier.

1.12 Les conventions d'écriture d'un programme C

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions.

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.

- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.

- On laisse un blanc

  - entre les mots-clefs `if`, `while`, `do`, `switch` et la parenthèse ouvrante qui suit,

  - après une virgule,
  - de part et d'autre d'un opérateur binaire.

- On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.

- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées. Le mieux est d'utiliser le mode C d'`Emacs`.