Les bases de la programmation en C

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

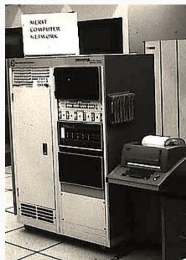
Historique

La création du langage C

Création du C

Le langage C a été conçu en 1972 par **Dennis Ritchie** et **Ken Thompson**, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11.

Kenneth Thompson (à gauche)
et Dennis Ritchie (à droite)



Un PDP-11

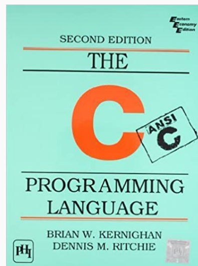
La création du langage C

Création du C

En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language*.

Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières.

Brian Kernighan



Dennis Ritchie

La création du langage C

Standardisation du C

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.

La compilation

Le C, un langage compilé

Langage compilé

Il y a deux grandes familles de langages : les langages **compilés**, et les langages **interprétés**.

Le langage C est un langage compilé.

Cela signifie qu'un programme C est décrit par un fichier texte, appelé **fichier source**.

Ce fichier n'est pas directement exécutable par un microprocesseur : il faut d'abord le traduire en **langage machine**.

Le C, un langage compilé

La compilation

Cette opération de traduction s'appelle la **compilation**, et est effectuée par un programme appelé le **compilateur**.

La compilation se décompose 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. . .).

Le C, un langage compilé

La compilation

Cette opération de traduction s'appelle la **compilation**, et est effectuée par un programme appelé le **compilateur**.

La compilation se décompose en 4 phases successives :

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.

La compilation

Cette opération de traduction s'appelle la **compilation**, et est effectuée par un programme appelé le **compilateur**.

La compilation se décompose en 4 phases successives :

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.

La compilation

Cette opération de traduction s'appelle la **compilation**, et est effectuée par un programme appelé le **compilateur**.

La compilation se décompose en 4 phases successives :

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.

Le C, un langage compilé

Les différents fichiers

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 C, un langage compilé

Quel compilateur utiliser ?

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.

L'émulateur de terminal

Terminal / Console

Un terminal (aussi appelé une console) est une fenêtre dans laquelle on peut taper des commandes que votre ordinateur va exécuter.

- Sous Linux, il existe pleins d'émulateurs de terminal différents ; en TP, nous utiliserons **Konsole** du projet **KDE**.
- Sous Mac OSX, l'application s'appelle **Terminal**.
- Sous Windows 10, vous pouvez installer **WSL** (Windows Subsystem for Linux) pour avoir un émulateur de terminal Linux.

Le C, un langage compilé

gcc

Par défaut, gcc active toutes les étapes de la compilation.
On le lance dans un terminal par la commande :

```
1  gcc [options] fichier.c [-llibraries]
```

Options

Les différentes options ne sont pas à connaître pour les concours ; nous en croiserons plusieurs au fil des TPs.

Les composants élémentaires du C

Les composants élémentaires du C

Composants élémentaires

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

- les identificateurs ;
- les mots-clés ;
- les constantes ;
- les chaînes de caractères ;
- les opérateurs ;
- les signes de ponctuation.

On peut ajouter à cette liste les commentaires, qui sont enlevés par le préprocesseur.

Les identificateurs

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`.

Les identificateurs

Syntaxe

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées) ;
- les chiffres ;
- l'underscore : _ (tiret du 8).

De plus, le premier caractère **ne** peut **pas** être un chiffre (et il est déconseillé d'utiliser _ comme premier caractère).

Exemple

- `var1`, `tab_23`, ou `_deb` sont des identificateurs valides.
- En revanche, `1i` et `i:j` ne le sont pas.

Les identificateurs

Remarques

- Les majuscules et minuscules sont différenciées.
- Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur.

$\hookrightarrow$ Cette limite peut dépendre du compilateur, mais elle sera toujours supérieure à 31 caractères.

Les mots-clés

Mots-clés

Un certain nombre de mots, appelés **mots-clés**, sont réservés pour le langage lui-même, et ne peuvent donc pas être utilisés comme identificateurs.

Aujourd'hui, le C compte 32 mots-clés, que l'on peut regrouper en catégories.

Les mots-clés

Les spécificateurs de stockages

`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`

divers

`return` `sizeof`

les instructions de contrôle

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

Les commentaires

Commentaires

Un commentaire est une partie du code source encadrée par `/* */`. Tout ce que vous écrivez entre les deux sera ignoré par le compilateur.

Les commentaires servent aux programmeurs à documenter ou expliquer une partie de leur code, pour s'y retrouver plus tard.

Exemple

```
/* Ceci est un commentaire */

/*
  Ceci est
  un commentaire
  sur plusieurs lignes
*/
```

Structure d’un programme C

Structure d'un programme C

Expression

Une **expression** est une suite de composants élémentaires syntaxiquement correcte.

Exemple 1

```
x = 0
```

Exemple 2

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

Structure d'un programme C

Instruction

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 { } pour former une **instruction composée** ou **bloc**, qui est syntaxiquement équivalent à une instruction.

Exemple

```
1 if (x != 0)
2 {
3   z = y / x;
4   t = y % x;
5 }
```

Structure d'un programme C

Déclaration

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

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

Exemple

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

Structure d'un programme C

Structure générale d'un programme C

```
1  [ directives au préprocesseur ]
2  [ déclarations de variables externes ]
3  [ fonctions secondaires ]
4
5  main()
6  {
7    déclarations de variables internes;
8    instructions;
9  }
```

Structure générale

Un programme C se présente de la façon ci-dessus.

Les **directives de préprocesseur** indiquent quels fichiers ou librairies doivent être incluses dans le programme.

On peut ensuite déclarer des **variables globales** et des **fonctions**.

Structure d'un programme C

Structure générale d'un programme C

```
1  [ directives au préprocesseur ]
2  [ déclarations de variables externes ]
3  [ fonctions secondaires ]
4
5  main()
6  {
7    déclarations de variables internes;
8    instructions;
9  }
```

La fonction `main`

Parmi les fonctions du fichier principal, il doit y avoir obligatoirement une fonction qui s'appelle **`main`**.
C'est la fonction principale de votre programme, c'est-à-dire celle qui sera lancée en premier lors de l'éxécution de votre programme compilé.

Structure d'un programme C

Structure générale d'un programme C

```
1  [ directives au préprocesseur ]
2  [ déclarations de variables externes ]
3  [ fonctions secondaires ]
4
5  main()
6  {
7    déclarations de variables internes;
8    instructions;
9  }
```

La fonction `main`

La fonction principale `main` peut avoir des paramètres.

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`.

Structure d'un programme C

Les fonctions secondaires

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale.

Une fonction secondaire peut se décrire de la manière suivante.

Exemple

```
1  type ma_fonction(type1 argument1, type2 argument2, ...)
2  {
3    déclarations de variables internes;
4    instructions;
5  }
```

Type de retour

Cette fonction retournera un objet dont le type sera `type` (à l'aide d'une instruction comme `return objet;`).

Structure d'un programme C

Les fonctions secondaires

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale.

Une fonction secondaire peut se décrire de la manière suivante.

Exemple

```
type ma_fonction(type1 argument1, type2 argument2, ...)
{
  déclarations de variables internes;
  instructions;
}
```

Arguments

Les arguments de la fonction obéissent à une syntaxe voisine de celle des déclarations : pour chaque argument, on précise son type et son nom, et on sépare les différents arguments par des virgules.

Structure d'un programme C

Exemple

La fonction suivante calcule le produit de deux entiers.

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

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

Les types prédéfinis

Les types prédéfinis

Les types

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

La 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 de 1.

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.

La mémoire

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels).

Ils sont désignés par les mots-clés suivants :

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

Le type caractère

Le type char

Le mot-clé `char` désigne une objet de type caractère (*character* en anglais).

Un `char` peut contenir n'importe quel élément du jeu de caractères de la machine utilisée.

Un objet de type `char` est codé sur un octet ; c'est l'objet le plus élémentaire en C.

Le jeu de caractère utilisé correspond généralement au codage ASCII sur 7 bits.

La table ASCII

	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

Le type caractère

Particularité

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.

Exemple

Si c est de type `char`, l'expression `c + 1` est valide : elle désigne le caractère suivant dans la table ASCII.

Ainsi, le programme suivant affiche le caractère 'B'.

```
1  main()
2  {
3    char c = 'A';
4    printf("%c", c + 1);
5  }
```

Les types entiers

Le type `int`

Le mot-clé désignant le type entier est `int`.

Un objet de type `int` est représenté en mémoire sur 32 bits (4 octets).

Attributs

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` est codé sur 16 bits (2 octets) ;
Un objet de type `long int` est codé sur 64 bits (8 octets).

Le mot-clé `sizeof`

`sizeof`

La taille d'un objet dans la mémoire n'est pas garantie d'être la même sur toutes les machines.

On peut obtenir le nombre d'octets utilisés par un objet dans la mémoire avec le mot-clé **`sizeof`**.

Les valeurs renvoyées par **`sizeof`** sont définies dans la librairie standard `limits.h`.

Le mot-clé sizeof

Exemple

```c
#include <stdio.h>

int main()
{
  int a;
  short int b;
  long int c;
  printf("a : %lu octets\n", sizeof(a));
  printf("b : %lu octets\n", sizeof(b));
  printf("c : %lu octets\n", sizeof(c));
  return 0;
}
```

Output

```
a : 4 octets
b : 2 octets
c : 8 octets
```

Les types entiers

Valeurs représentables

Puisqu'un entier n'est représenté que sur un nombre fini de chiffres en base 2, nous ne pouvons pas représenter tous les entiers.

Si un entier est codé sur n bits, on ne peut avoir que 2^n valeurs différentes.

Les types entiers

Valeurs négatives

C'est en général pratique de pouvoir travailler si on le souhaite avec des valeurs négatives.

Par défaut, les types entiers peuvent donc coder autant (à un près) de valeurs positives que négatives.

`unsigned`

Si on est sûr de ne jamais avoir de valeurs négatives, on peut demander à travailler avec des valeurs **non signées** à l'aide du mot-clé **`unsigned`**. On pourra alors aller deux fois plus loin dans les valeurs positives.

Les types entiers

Valeurs représentables

Voici les valeurs représentables pour les différents types entier :

`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`	$[-2^{63}; 2^{63}[$
`unsigned long int`	$[0; 2^{64}[$

Les types entiers

Programme MP2I

Puisque la taille en mémoire de certains des types précédents dépend de la machine ou du compilateur, des types (dont le nom est plus parlant) ont été rajoutés.
Leur taille en mémoire est clairement indiquée, et ne doit pas dépendre de la machine ou du compilateur utilisé.

Ce sont ces types entiers qu'il est conseillé d'utiliser dans le programme de MP2I.

- Types entiers signés : `int8_t`, `int32_t`, `int64_t`.
- Types entiers non signés : `uint8_t`, `uint32_t`, `uint64_t`.

Le programme de MP2I indique également : "Lorsque la spécification d'une taille précise pour le type n'apporte rien à l'exercice, on utilise les types signés `int` et non signés `unsigned int`."

Les types flottants

Les types flottants

Les types **float**, **double** et **long double** servent à représenter des nombres en virgule flottante.

Ils correspondent aux différentes précisions possibles :

float	32 bits
double	64 bits
long double	128 bits

Les types flottants

Exemple

```c
#include <stdio.h>

int main()
{
  float a;
  double b;
  long double c;
  printf("a : %lu octets\n", sizeof(a));
  printf("b : %lu octets\n", sizeof(b));
  printf("c : %lu octets\n", sizeof(c));
  return 0;
}
```

Output

```
a : 4 octets
b : 8 octets
c : 16 octets
```

Les constantes

Les constantes

Constante

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, caractère, énumération.

Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

Les constantes entières

Écriture décimale

Une suite de chiffres **ne** commençant **pas** par 0 est interprétée comme un entier écrit en base 10 (**décimale**).

Exemple

`0` et 2437348 sont des constantes entières décimales.

Les constantes entières

Écriture octale

Une suite de chiffres **commençant** par 0 est interprétée comme un entier écrit en base 8 (**octale**).

Exemple

- `00` vaut 0.
- `0377` vaut $3 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 255$.

Les constantes entières

Écriture hexadécimale

Une suite de chiffres **commençant** par `0x` est interprétée comme un entier écrit en base 16 (**hexadécimale**).

Les lettres de a à `f` sont utilisées pour représenter les nombres de 10 à 15.

Exemple

- `0x0` vaut 0.
- `0xc` vaut 12.
- `0xff` vaut $15 \times 16^1 + 15 \times 16^0 = 255$.

Les constantes entières

Type utilisé

Par défaut, une constante décimale est représentée avec le format le plus court permettant de la représenter parmi les format `int`, `long int`, et `unsigned long int`.

Les constantes réelles

Écriture scientifique

Une constante réelle peut être écrite tout simplement à l'aide d'une virgule, ou bien en écriture scientifique à l'aide de la lettre e.

Exemple

- `12.34` vaut 12.34.
- `12.3e-4` vaut 12.3×10^{-4}.

Les constantes caractères

Caractère

Pour désigner un caractère seul, il suffit de le mettre entre **apostrophes**.

Exemple

`'A'` ou `'!'`.

Les constantes caractères

Caractère d'échappement

L'antislash (\) est un caractère spécial, dit d'**échappement**.

Il permet d'utiliser des caractères spéciaux, en le combinant avec le caractère suivant.

Exemple

- `\'` désigne le caractère `'`.
- `\"` désigne le caractère `"`.
- `\\` désigne le caractère `\`.
- `\n` désigne un **retour à la ligne**.

Les constantes chaînes de caractères

Chaîne de caractère

Une **chaîne de caractères** est une suite de caractères entourés par des guillemets.

Exemple

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

Caractères spéciaux

Une chaîne de caractères peut contenir des **caractères spéciaux**.

Exemple

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

Les constantes chaînes de caractères

Code sur plusieurs lignes

Le caractère \ à la fin d'une ligne de code permet d'ignorer le passage à la ligne.

Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes.

Exemple

```
1 "Ceci est une longue longue longue longue longue longue longue longue longue \
2 longue chaîne de caractères"
```

Les opérateurs

L'affectation

Affectation

En C, l'**affectation** est un opérateur à part entière. Elle est symbolisée par le signe =.

Sa syntaxe est la suivante.

```
variable = expression
```

L'affectation

```
var = expr
```

Fonctionnement

Le terme de gauche d'une affectation peut être une variable simple ou un élément d'un tableau, mais pas une constante.

Cette expression d'affectation a pour effet :

- d'évaluer `expr`,
- et d'affecter la valeur obtenue à `var`.

De plus, cette expression d'affectation possède une valeur, qui est celle de `expr`.

Exemple

L'expression `i = 5` vaut 5.

L'affectation

Conversion implicite

L'affectation effectue une **conversion de type implicite** : la valeur du terme de droite est convertie dans le type du terme de gauche.

```
1  int main()
2  {
3    int i, j = 2;
4    float x = 2.5;
5    i = j + x;
6    x = x + i;
7    printf("x : %f\n",x);
8    return 0;
9  }
```

Exemple

Le code ci-dessus affiche `x : 6.5`, car à la ligne 5 l'expression `j + x` a été convertie en entier $(4.5 \to 4)$.

Les opérateurs arithmétiques

Opérateurs arithmétiques

Les **opérateurs arithmétiques** classiques sont :

- l'opérateur **unaire** - (changement de signe) ;
- les opérateurs **binaires** :
  - `+` addition ;
  - `-` soustraction ;
  - `*` multiplication ;
  - `/` division ;
  - `%` reste de la division euclidienne (modulo).

Les opérateurs arithmétiques

Division

Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière (quotient de la division euclidienne) et la division entre flottants.

- Si les deux opérandes sont de type entier, l'opérateur / effectue la division entière.
- En revanche, il renverra une valeur flottante dès que l'un des opérandes est un flottant.

Exemple

```c
float x;
x = 3 / 2; // x vaut 1

float y;
y = 3 / 2.; // y vaut 1.5
```

Les opérateurs arithmétiques

Modulo

L'opérateur `%` ne s'applique qu'à des opérandes de type entier.

Il est **fortement conseillé** de ne pas l'utiliser avec des entiers négatifs, car le signe du résultat peut varier selon le compilateur utilisé.

Les opérateurs relationnels

Opérateurs relationnels

Les **opérateurs relationnels du** C sont :

- `>` strictement supérieur ;
- `>=` supérieur ou égal ;
- `<` strictement inférieur ;
- `<=` inférieur ou égal ;
- `==` égal ;
- `!=` différent.

Les opérateurs relationnels

```
1  expression1 op expression2
```

Fonctionnement

Dans la syntaxe ci-dessus, les deux expressions sont évaluées puis comparées.

La valeur renvoyée est un entier : 1 si le test est vrai, 0 s'il est faux.

Les opérateurs relationnels

```c
#include <stdbool.h>
```

Remarque

Par défaut, il n'y a pas de type booléen en C.

On peut en importer un avec l'instruction ci-dessus.

On obtient alors un type `bool`, et deux constantes `true` (qui vaut 1) et `false` (qui vaut 0).

Les opérateurs relationnels

Attention

Attention à ne pas confondre l'opérateur d'égalité == avec l'opérateur d'affectation = !

```
1  main()
2  {
3    int a = 0;
4    int b = 1;
5    if (a = b)
6    {
7      printf("a et b sont egaux\n");
8    }
9    else
10   {
11     printf("a et b sont differents\n");
12   }
13 }
```

Output

```
a et b sont egaux
```

Les opérateurs logiques booléens

Opérateurs logiques

Les opérateurs logiques en C sont :

- `!` négation logique
- `&&` “et” logique
- `||` “ou” logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un `int` qui vaut 1 si le test est vrai, ou 0 sinon.

En l’absence de parenthèses, la priorité des opérateurs est : `!` $>$ `&&` $>$ `||`.

Mais si vous avez le moindre doute, je vous conseille de parenthéser vos expressions !

Les opérateurs logiques booléens

```
expr1 op1 expr2 op2 ... exprN
```

Évaluation

Dans une expression comme celle ci-dessus, l'évaluation se fait de gauche à droite, et s'arrête dès que le résultat final est déterminé.

Exemple

```
int i;
int j;
...

if ((i >= 0) && (i <= 9) && !(j == 0))
...
```

Exemple

Dans le code ci-dessus, la dernière clause `!(j == 0)` ne sera pas évaluée si `i` n'est pas entre 0 et 9.

Les opérateurs d'affectation composée

Affectation composée

Pour tout opérateur op, l'expression :

```
expr1 = expr1 op expr2
```

peut être remplacée par :

```
expr1 op= expr2
```

On parle alors d'**affectation composée**.

Exemple

```
1  i += 3; // même chose que i = i + 3;
2  i -= 3; // même chose que i = i - 3;
3  i *= 3; // même chose que i = i * 3;
4  i /= 3; // même chose que i = i / 3;
```

Les opérateurs d'incrémentation et de décrémentation

Incrémentation

En informatique, il arrive très souvent qu'on ait besoin d'ajouter 1 à une variable entière : on parle d'**incrémentation**.

Son opération inverse s'appelle la **décrémentation**.

Une syntaxe existe pour effectuer ces opérations en C. Elle est hors programme, mais je vous la signale tout de même car très classique en programmation C.

```
i++; // augmente la valeur de i de 1
i--; // diminue la valeur de i de 1
```

L’opérateur de conversion de type

Conversion de type

L’opérateur de **conversion de type** (appelé **cast** en anglais) permet de convertir un objet dans un autre type.

La syntaxe est la suivante :

```
(nouveau_type) objet_a_convertir
```

Exemple

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

Output

```
valeur : 1.5
```

La programmation impérative

Instructions de contrôle

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.

Les instructions de branchement conditionnel

Instruction de branchement conditionnel

Il arrive souvent d'avoir besoin d'exécuter une partie du code uniquement si un test est vérifié (et d'en exécuter éventuellement un autre sinon).

C'est ce qu'on appelle une **instruction de branchement conditionnel**.

Les instructions de branchement conditionnel

```
1   if (expression1)
2   {
3     instruction1;
4     ...
5   }
6   else if (expression2)
7   {
8     instruction2;
9     ...
10  }
11  ...
12  else if (expressionN)
13  {
14    instructionN;
15    ...
16  }
17  else
18  {
19    instruction_else;
20    ...
21  }
```

Les instructions de branchement conditionnel

Propriétés

- On peut utiliser autant de **else if** qu'on veut (potentiellement zéro).
- Le dernier **else** est facultatif.

Ainsi, si on ne veut rien faire lorsque le premier test n'est pas vérifié, on peut écrire tout simplement :

```
1  if (expression)
2  {
3    instructions;
4    ...
5  }
```

Les boucles

Boucles

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

Boucle while

boucle while

```
while (expression)
{
  instructions;
  ...
}
```

while

Il arrive souvent qu'on ne sache pas à l'avance combien de fois certaines instructions doivent être répétées, mais qu'on connaisse une condition nous indiquant d'arrêter.
Dans ce cas, on utilise une boucle **while**, dont la syntaxe est présentée ci-dessus.

Boucle `while`

boucle while

```
1 while (expression)
2 {
3   instructions;
4   ...
5 }
```

`while`

Lorsqu'on arrive sur une instruction **`while`** :

- si **`expression`** n'est pas vraie, le contenu de la boucle **`while`** n'est pas exécuté, et on saute directement après le bloc d'**`instructions`** ;
- si **`expression`** est vraie (non nulle), les **`instructions`** sont sont **`toutes`** exécutées une fois, puis on retourne à la ligne du **`while`**.

Boucle `while`

boucle while

```
while (expression)
{
  instructions;
  ...
}
```

Remarques

- Si **`expression`** est fausse au départ, le contenu de la boucle **`while`** n'est donc jamais exécuté.
- Si jamais **`expression`** est vrai en haut de la boucle, puis devient faux au milieu des **`instructions`**, le reste des **`instructions`** continue quand même de s'exécuter avant de retester la valeur de l'**`expression`**.

Boucle `while`

Exemple

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

Exemple

Le programme ci-dessus imprime les entiers de 1 à 9.

Boucle `do ... while`

do ... while

```
1 do
2 {
3   instructions;
4   ...
5 }
6 while (expression);
```

`do ... while`

Il peut arriver que l'on veuille effectuer le **test de continuation** qu'après avoir exécuter une première fois les **`instructions`**.

Dans ce cas, on peut utiliser la boucle **`do ... while`**, dont la syntaxe est présentée ci-dessus.

En utilisant cette syntaxe, les **`instructions`** seront bien exécutées une première fois, avant de tester la valeur d'**`expression`**.

Boucle do ... while

Exemple

```
1  int a;
2
3  do
4  {
5    printf("\n Entrez un entier entre 1 et 10 : ");
6    scanf("%d",&a);
7  }
8  while ((a<=0) || (a > 10));
```

Exemple

Le code ci-dessus demande à l'utilisateur de saisir au clavier une valeur entre 1 et 10, et continue de demander une valeur tant qu'elle n'est pas dans $[1;10]$.

Boucle for

boucle for
```
1  for (initialisation; condition; modification)
2  {
3    instructions;
4    ...
5  }
```

boucle while équivalente
```
1  initialisation;
2  while (condition)
3  {
4    instructions;
5    ...
6    modification;
7  }
```

for

Il arrive souvent que la répétition nécessaire à notre problème soit très simple. Dans ce cas, on préfère utiliser une boucle `for` pour alléger notre code, à l'aide de la syntaxe ci-dessus (à gauche).

Cela est équivalent à la boucle `while` de droite.

Boucle for

Exemple

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

Exemple

Le programme ci-dessus imprime les entiers de 1 à 9.

Boucle for

Exemple

```
int factorielle(int n)
{
  int i;
  int fact = 1;

  for (i = 1; i <= n; i++)
  {
    fact *= i;
  }

  return fact;
}
```

Exemple

La fonction ci-dessus calcule $n!$.

Les instructions de branchement non conditionnel

`break`

À l'intérieur d'une boucle, l'instruction **`break`** permet d'interrompre la boucle, et de passer directement à la première instruction qui suit la boucle.

En cas de boucles imbriquées, **`break`** interrompt uniquement la boucle la plus interne.

Les instructions de branchement non conditionnel

Exemple

```c
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);
}
```

Output

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

Les instructions de branchement non conditionnel

`continue`

À l'intérieur d'une boucle, l'instruction **`continue`** permet et de passer directement en haut de la boucle sans finir d'exécuter le tour de boucle courant.

Les instructions de branchement non conditionnel

Exemple

```
1  main()
2  {
3    int i;
4    for (i = 0; i < 5; i++)
5    {
6      if (i == 3)
7      {
8        continue;
9      }
10     printf("i = %d\n",i);
11   }
12   printf("valeur de i a la sortie de la boucle : %d\n",i);
13 }
```

Output

```
i = 0
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle : 5
```

Les fonctions d’entrées/sorties classiques

Les fonctions d'entrées/sorties classiques

Entrées/sorties

Les fonctions d'**entrées/sorties** servent au programme à interagir avec l'utilisateur.

Une fonction d'**entrée** sert à acquérir de l'information (par exemple, l'utilisateur saisit une information au clavier).

Une fonction de **sortie** sert à fournir une information à l'utilisateur (par exemple, afficher du texte à l'écran).

Les fonctions d’entrées/sorties classiques

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

stdio

Les fonctions d’entrées/sorties classiques sont regroupées dans la librairie standard **`stdio.h`** (pour ***standard input/output*** en anglais).

On peut importer cette librairie avec l’**instruction de préprocesseur** ci-dessus.

La fonction d'écriture `printf`

```
1  printf("chaîne de contrôle", expr1, ..., exprN);
```

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 celle ci-dessus.

La fonction d'écriture `printf`

```
1   printf("chaîne de contrôle", expr1, ..., exprN);
```

printf

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'impressions pour la fonction printf

format	type	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 signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale signée
%lx	unsigned long int	hexadécimale non signée

Les formats d'impressions pour la fonction printf

format	type	écriture
%f	float ou double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	float ou double	décimale notation scientifique
%le	long double	décimale notation scientifique
%g	float ou double	décimale, la plus courte parmi %f et %e
%lg	long double	décimale, la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

Les formats d’impressions pour la fonction `printf`

Remarques

Par défaut, un flottant sera affiché avec 6 chiffres après la virgule.

On peut modifier ce nombre de chiffres : par exemple, `%.12f` affichera le flottant avec 12 chiffres après la virgule.

De même, `%.10s` n’affichera que les 10 premiers caracteres d’une chaîne de caractères.

la fonction printf

Exemple

```c
#include <stdio.h>

main()
{
  int i = 23674;
  int j = -23674;
  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",j,j);
  printf("\nimpression de x: \n");
  printf("%f \t %e \t %g",x,x,x);
  printf("\n%.2f \t %.2e",x,x);
  printf("\n%.20f",x);
  printf("\n%.20e",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");
}
```

Output

```
impression de i:
23674    23674   56172   5c7a
impression de j:
-23674   4294943622
impression de x:
1000.000000      1.000000e+03    1000
1000.00          1.00e+03
1000.00000001000000793283
1.00000000001000000793e+03
impression de c:
A        65
impression de chaine:
chaine de caracteres     chaine de
```

La fonction `scanf`

```
1 scanf("chaîne de contrôle", adresse1, ..., adresseN)
```

scanf

La fonction **scanf** permet de saisir des données au clavier, et de les stocker aux **adresses** spécifiées par les arguments de la fonction.
La **chaîne de contrôle** indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères.

La fonction scanf

```
1  scanf("chaîne de contrôle", adresse1, ..., adresseN)
```

scanf

Comme pour **printf**, les **conversions de format** sont spécifiées par un caractère précédé su signe %.
Les données à entrer au clavier doivent être séparées par des espaces ou des **<RETURN>**, sauf s'il s'agit de caractères.

Attention

Les formats valides pour la fonction **scanf** diffèrent légèrement de ceux de la fonction **printf**.

La fonction `scanf`

Adresse d'une variable

Il ne faut pas donner une variable à **`scanf`**, mais l'**adresse** de la variable dans la mémoire.

On peut obtenir l'**adresse mémoire** d'une variable avec l'esperluette : **`&`**.

Exemple

L'adresse mémoire de la variable **`var`** est : **`&var`**.

La fonction scanf

Exemple

```
1  #include <stdio.h>
2
3  main()
4  {
5    int i;
6    printf("entrez un entier sous forme hexadecimale : ");
7    scanf("%x", &i);
8    printf("valeur decimale : %d\n", i);
9  }
```

Output

```
entrez un entier sous forme hexadecimale :
```

Exemple

En exécutant le programme suivant, la première ligne s'affiche dans la console jusqu'aux "`:`".
Le programme se met alors en pause, et attend que l'utilisateur saisisse une valeur au clavier.

La fonction scanf

Exemple

```
1  #include <stdio.h>
2
3  main()
4  {
5    int i;
6    printf("entrez un entier sous forme hexadecimale : ");
7    scanf("%x", &i);
8    printf("valeur decimale : %d\n", i);
9  }
```

Output

```
entrez un entier sous forme hexadecimale : 1a
valeur decimale : 26
```

Exemple

Après avoir saisi **`1a`** puis appuyé sur **`<ENTER>`**, le programme continue et affiche la ligne 2.

Les formats de saisie pour la fonction `scanf`

format	type	représentation
%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

Les formats de saisie pour la fonction scanf

format	type	représentation
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation scientifique
%le	double	flottante notation scientifique
%Le	long double	flottante notation scientifique
%g	float	flottante au choix
%lg	double	flottante au choix
%Lg	long double	flottante au choix
%c	char	caractère
%s	char*	chaîne de caractères

Les conventions d’écriture d’un programme C

Les conventions d’écriture d’un programme C

Conventions d’écriture

Il n’existe que très peu de contraintes concernant l’écriture d’un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles.

Ainsi, il existe un certain nombres de conventions, respectées par tout programmeur digne de ce nom.

Les conventions d'écriture d'un programme C

Conventions d'écriture

- 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.

Les conventions d'écriture d'un programme C

Conventions d'écriture

- On laisse un espace :
  - entre les mots-clés `if`, `while`, `do` et la parenthèse ouvrante qui suit ;
  - après une virgule ;
  - de part et d'autre d'un opérateur binaire.
- On ne met pas d'espace :
  - entre un opérateur unaire et son opérande ;
  - entre les deux caractères d'un opérateur d'affectation composée.

Les conventions d’écriture d’un programme C

Conventions d’écriture

- Le code doit être indenté afin que toutes les instructions d’un même bloc soient alignées.

Le mieux est d’utiliser l’indentation automatique de votre IDE.