Le C est un langage typé statiquement : chaque variable, chaque constante et chaque expression, a un type défini à la compilation. Le langage lui-même fournit des types permettant de manipuler des nombres (entiers, réels ou complexes) ou des caractères (eux-mêmes étant manipulés comme des entiers spéciaux), et permet de construire des types plus complexes à partir de ces premiers, par exemple en groupant des données de même type en tableaux, ou des données de types différents dans des structures. Dans ce chapitre, nous étudierons les types de base fournis par le C, l'étude des types complexes étant faite dans la suite du livre.

Entiers

Il y a cinq types de variables entières (« integer » en anglais) :

- `char` ;
- `short int`, ou plus simplement `short` ;
- `int` ;
- `long int`, ou `long` ;
- `long long int`, ou `long long` (ce type a été ajouté depuis la norme C99).

Comme évoqué en introduction, le type caractère `char` est particulier, et sera étudié en détail plus bas.

Les types entiers peuvent prendre les modificateurs `signed` et `unsigned` qui permettent respectivement d'obtenir un type signé ou non signé. Ces modificateurs ne changent pas la taille des types. Le langage ne définit pas exactement leur taille, mais définit un domaine de valeurs minimal pour chacun.

Représentation des entiers signés

La norme C tient compte des anciennes représentations des nombres signés telles que le **signe+valeur absolue** et le **complément à 1**. Ces deux représentations sont brièvement expliquées ci-dessous :

Signe + valeur absolue (SVA)

Un bit contient le signe du nombre (par ex : 0 pour +, 1 pour -), les autres bits sont utilisés pour la valeur absolue. On peut donc représenter 0 de deux manières : +0 (000...000) ou -0 (100..000). Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1}-1)$ *(111...111)* et $+(2^{N-1}-1)$ *(011...111)*.

Complément à 1 (CPL1)

Les bits des nombres négatifs sont inversés. On peut donc représenter 0 de deux manières : +0 (000...000) ou -0 (111..111). Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1}-1)$ *(100...000)* et $+(2^{N-1}-1)$ *(011...111)*.

Ces deux représentations peuvent représenter la valeur nulle de deux manières différentes.

La représentation moderne des nombres négatifs utilise le **complément à 2 (CPL2)** qui consiste à représenter les nombres négatifs comme le complément à 1 et en ajoutant 1. Sur un nombre fixe de bits, la valeur 0 n'a qu'une seule représentation : +0 (000...000) et -0 (111...111 + 1 = (1)000...000) ont deux représentations identiques. Sur N bits on peut donc représenter tout nombre entre $-(2^{N-1})$ *(100...000)* et $+(2^{N-1}-1)$ *(011...111)*. Cette représentation possède donc un domaine plus large.

Le tableau ci-dessous donne le domaine des valeurs quelle que soit la représentation utilisée (SVA, CPL1 ou CPL2) :

Domaines de valeurs **minimaux** des types entiers (C90 et C99) quelle que soit sa représentation (SVA, CPL1 ou CPL2)[1]

Type	Taille	Borne inférieure	Borne inférieure (formule)	Borne supérieure	Borne supérieure (formule)
`signed char`	≥ 8 bits	-127	$-(2^7-1)$	+127	2^7-1
`unsigned char`	≥ 8 bits	0	0	+255	2^8-1
`short`	≥ 16 bits	-32 767	$-(2^{15}-1)$	+32 767	$2^{15}-1$
`unsigned short`	≥ 16 bits	0	0	+65 535	$2^{16}-1$
`int`	≥ 16 bits	-32 767	$-(2^{15}-1)$	+32 767	$2^{15}-1$
`unsigned int`	≥ 16 bits	0	0	+65 535	$2^{16}-1$
`long`	≥ 32 bits	-2 147 483 647	$-(2^{31}-1)$	+2 147 483 647	$2^{31}-1$
`unsigned long`	≥ 32 bits	0	0	+4 294 967 295	$2^{32}-1$
`long long` *(C99)*	≥ 64 bits	-9 223 372 036 854 775 807	$-(2^{63}-1)$	+9 223 372 036 854 775 807	$2^{63}-1$
`unsigned long long` *(C99)*	≥ 64 bits	0	0	+18 446 744 073 709 551 615	$2^{64}-1$

Cette table signifie qu'un programme peut utiliser sans problème une variable de type `int` pour stocker la valeur $2^{15}-1$, quel que soit le compilateur ou la machine sur laquelle va tourner le programme.

Par contre, une implémentation C *peut* fournir des domaines de valeurs plus larges que ceux indiqués au-dessus :

- Les domaines indiqués pour les nombres signés dans le tableau précédent sont ceux d'une implémentation par **complément à 1**, ou par **signe et valeur absolue**. Pour **le complément à 2**, la borne inférieure est de la forme -2^{N-1}, ce qui autorise une valeur supplémentaire (ex : `int` de -2^{15} à $+2^{15}-1$, soit de -32 768 à +32 767),
- un `int` implémenté sur 32 bits pourrait aller de $-(2^{31}-1)$ à $2^{31}-1$, par exemple[2], et un programme tournant sur une telle implémentation peut alors utiliser ces valeurs, mais il perdrait en portabilité.

Les processeurs modernes supportent un grand nombre de bits, pouvant autoriser les compilateurs à fournir des types stockés sur un plus grand nombre de bits que le nombre minimal indiqué par la norme. Cependant des processeurs plus anciens sont encore utilisés de nos jours (coût réduit) sur des systèmes embarqués. Il est donc important de vérifier que les valeurs utilisées puissent être stockées dans les variables. Par exemple, cela permet l'utilisation d'une bibliothèque C (exemple : un décodeur vidéo) sur un système embarqué. Dans le cas où une bibliothèque assume une taille des types plus grande que supporté par le compilateur pour le processeur, il faudra effectuer des modifications du code source pour que la bibliothèque soit utilisable sur le système embarqué.

Domaines de valeurs des types entiers représentés **en complément à 2 (CPL2)**

Type	Borne inférieure	Borne inférieure (formule)	Borne supérieure	Borne supérieure (formule)
`signed char`	-128	$-(2^7)$	+127	2^7-1
`unsigned char`	0	0	+255	2^8-1
`short`	-32 768	$-(2^{15})$	+32 767	$2^{15}-1$
`unsigned short`	0	0	+65 535	$2^{16}-1$
`int`	-32 768	$-(2^{15})$	+32 767	$2^{15}-1$
`unsigned int`	0	0	+65 535	$2^{16}-1$
`long`	-2 147 483 648	$-(2^{31})$	+2 147 483 647	$2^{31}-1$
`unsigned long`	0	0	+4 294 967 295	$2^{32}-1$
`long long` *(C99)*	-9 223 372 036 854 775 808	$-(2^{63})$	+9 223 372 036 854 775 807	$2^{63}-1$
`unsigned long long` *(C99)*	0	0	+18 446 744 073 709 551 615	$2^{64}-1$

Par ailleurs, une relation d'ordre entre ces domaines de valeurs est garantie ; qui peut être exprimée ainsi :

```
domaine(char) ≤ domaine(short) ≤ domaine(int) ≤ domaine(long) ≤ domaine(long long)
```

Cela signifie que toutes les valeurs possibles pour une variable du type `char` sont aussi utilisables pour les autres types ; mais aussi que, par exemple, une valeur valide pour le type `int` peut ne pas être représentable dans une variable de type `short`.

Si vous ne savez pas quel type donner à une variable de type entier, le type `int` est par défaut le meilleur choix (à condition que votre donnée ne dépasse pas $2^{15}-1$) : ce type est la plupart du temps représenté au niveau matériel par un « mot machine », c'est-à-dire qu'il est adapté à la taille que la machine peut traiter directement (il fait usuellement 32 bits sur un PC 32 bits, par exemple). Cela permet un traitement plus rapide par le matériel. De plus, beaucoup de bibliothèques (que ce soit celle fournie par le langage C ou d'autres) utilisent ce type pour passer des entiers, ce qui fait que l'utilisation de ces bibliothèques sera plus aisée.

Par ailleurs, un utilisateur peut connaître les domaines de valeurs exacts de sa machine en utilisant l'en-tête `<limits.h>`.

Portabilité apportée par C99

L'incertitude sur l'intervalle de valeur de chaque type en fonction de la machine peut s'avérer extrêmement gênante, pour ne pas dire rédhibitoire. En effet, certains programmes peuvent nécessiter un type de données de taille fixe et cependant être destinés à être portables. Pour ces programmes, les types entiers du C ne sont pas suffisants. *Beaucoup* d'extensions ont été rajoutées pour définir explicitement des types entiers à intervalle fixe (8, 16, 32 bits...) à partir des types de base, avec une nomenclature loin d'être homogène d'un compilateur à l'autre (ce qui, loin de résoudre le problème, ne faisait que le déplacer).

La norme ISO C99 décide une bonne fois pour toute de définir, dans l'en-tête <stdint.h>, plusieurs nouveaux types où *N* représente un nombre entier définissant la taille requise en bit :

- des types implémentés optionnellement sur certaines architectures (à éviter ?) ;
  - entiers signés ou non et de longueur *N* exacte : `uint`*N*`_t` et `int`*N*`_t` ;
  - entiers pouvant contenir un pointeur : `intptr_t` et `uintptr_t` ;
- des types requis par toutes les architectures respectant la norme C99 ;
  - entiers devant être plus grand que *N* bits au moins : `int_least`*N*`_t` et `uint_least`*N*`_t` ;

- entiers rapides à calculer et plus grand que *N* bits au moins : `int_fastN_t` et `uint_fastN_t` ;
- plus grand entier : `intmax_t` et `uintmax_t`.

Cet en-tête définit aussi des constantes pour les valeurs minimales et maximales de chaque type.

L'include <inttypes.h> définit les constantes symboliques à utiliser pour imprimer ces nouveaux types avec les fonctions de la famille de printf (PRIxxx) et les lire avec celles de scanf (SCNxxx).

Constantes numériques entières

Il existe différentes suites de caractères qui sont reconnues comme étant des constantes numériques entières :

- un nombre en notation décimale : une suite de chiffres (0-9) ;
- le caractère « 0 » suivi d'un nombre en notation octale : une suite de chiffres compris entre 0 et 7 ;
- les caractères « 0x » suivi d'un nombre en notation hexadécimale : une suite de chiffres et des lettres a, b, c, d, e, f (ou A, B, C, D, E, F).

Par défaut, une constante numérique entière est de type `int` et, si sa valeur est trop grande pour le type `int`, elle prend celle du type « plus grand » suffisant. Comme les domaines de valeurs des types peuvent varier suivant la machine, le type effectif d'une constante peut lui aussi varier. Cela peut s'avérer problématique lors de passage de paramètres à des fonctions à nombre variable d'arguments, par exemple. À cause de cela, il est recommandé de forcer le type de la constante en le **postfixant** des attributs suivants :

- `U` : la constante est non-signée (voir la section promotion pour comprendre les implications) ;
- `L` : la constante est de type `long` au lieu de `int` ;
- `LL` est une nouveauté C99 pour les constantes de type `long long`.

`U` peut être combiné à `L` et `LL` pour obtenir les types `unsigned long` et `unsigned long long`, respectivement. Lorsqu'une constante est suffixée, mais que sa valeur est trop grande pour le type demandé, le même processus de recherche de type « assez grand » est utilisé[3].

Débordement

Sur une machine donnée, un type entier a un domaine de valeurs fixe. Considérons qu'on travaille sur un PC en 32 bits, en complément à deux : sur un tel ordinateur, le type `int` varie souvent de -2^{31} à $2^{31}-1$. Cela permet de manipuler sans problème des valeurs dans ce domaine. Par contre, si on utilise des valeurs hors du domaine, par exemple 2^{32}, et qu'on essaye de la stocker dans une variable de type `int` sur une telle machine, que se passe-t-il ?

La réponse dépend du type:

- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *signé* (comme dans l'exemple), la conversion n'est pas définie par le langage. Cela signifie que tout peut arriver.
- Si on essaye d'enregistrer une valeur hors domaine dans une variable de type *non signé* (`unsigned long`, par exemple), la conversion se fait modulo la valeur maximale représentable par ce type + 1.

Exemples : On suppose que le type `unsigned char` est codé sur 8 bits. Alors une valeur de ce type peut aller de 0 à 2^8-1, soit 255. Par la règle précédente, les conversions se feront donc modulo 255 + 1, soit 256. On considère le programme suivant :

```c
#include <stdio.h>

int main(void)
{
    unsigned char c = 300;
    /* %hhu sert à dire à printf() qu'on veut afficher un unsigned char */
    printf("La variable c vaut %hhu.\n", c);
    c = -5;
    printf("La variable c vaut %hhu.\n", c);
```

```
10      return 0;
11 }
```

Le résultat du programme est :

```
La variable c vaut 44.
La variable c vaut 251.
```

En effet, on a 300 - 256 = 44 et -5 + 256 = 251.

De même :

```c
#include <stdio.h>

int main(void)
{
    signed char c = 300;
    /* %hhd sert à dire à printf() qu'on veut afficher un signed char */
    printf("La variable c vaut %hhd.\n", c);
    c = -300;
    printf("La variable c vaut %hhd.\n", c);
    return 0;
}
```

Le résultat du programme peut alors être :

```
La variable c vaut 44.
La variable c vaut -44.
```

Sur une telle machine, les types signés sont traités de la même manière que les types non signés.

Si vous essayez de compiler ces deux programmes, votre compilateur pourra détecter les débordements et vous en avertir (GCC le fait).

Réels

Les nombres réels ne pouvant tous être représentés, sont approximés par des nombres à virgule flottante. Comme dans le cas des entiers, il existe plusieurs types de nombre à virgule flottante. En voici la liste triée par précision croissante :

- **float** ;
- **double** ;
- **long double**.

La norme C90 était assez floue concernant les nombres à virgule flottante, leurs représentations, la précision des opérations, etc., ce qui fait que c'était un des domaines où la conception de programmes utilisant les nombres flottants était chose peu aisée. Le C99 a clarifié les choses en précisant qu'une implémentation C devait respecter la norme **IEC 60559:1989** ***Arithmétique binaire en virgule flottante pour systèmes à microprocesseur***. Cette norme (dérivée de IEEE 754) définit des formats de données pour les nombres à virgule flottante, ainsi que des opérations et fonctions sur ces nombres. Elle garantit, entre autres :

- que certains types de données auront toujours le même format ;
- et que les calculs effectués sur un type de donnée donneront toujours le même résultat.

Elle définit de plus comment sont gérés les cas exceptionnels, comme les infinis, les *NaN* (pour *Not a Number*, résultant par exemple de la division 0/0), etc.

Les types flottants du C correspondent aux type IEC 60559 de la manière suivante :

- **float** : simple précision
- **double** : double précision
- **long double** : suivant l'implémentation, soit la double précision étendue, soit un type non-IEC 60559 (mais de précision au moins égale à **double**), soit double précision.

Constantes réelles

Une suite de caractères représente une constante à virgule flottante si :

- c'est une suite de chiffres séparée par un caractère « point », cette séparation pouvant s'effectuer à n'importe quel endroit de la suite (`0.0`, `.0` et `0.` représentent tous les trois la valeur 0 de type `double`) ;
- un *nombre* suivi d'un caractère « e » suivi d'un entier.

Dans le deuxième cas, le *nombre* peut être soit un entier, soit un réel du premier cas.

Les constantes sont de type `double` par défaut. Pour demander le type `float`, il faut la suffixer par `f` ou `F`, et pour le type `long double` par `l` ou `L`.

Arithmétique

Une attention particulière doit être portée sur la précision des types réels. Ces différents types ne font qu'approximer l'ensemble des nombres réels, avec une précision finie. Des erreurs d'arrondis sont à prévoir, ce qui est très problématique pour des domaines qui n'en tolèrent pas (notamment pour les applications financières, il est conseillé de ne pas utiliser de calcul en virgule flottante).

Le type `float` en particulier a une précision minimale, qui est bien souvent insuffisante. Voici un exemple classique d'erreur à ne pas faire qui illustre les problèmes liés à la précision des types flottants :

```c
#include <stdio.h>

int main(void)
{
    float i = 0;
    int j;
    for (j = 0; j < 1000; j ++)
    {
        i += 0.1;
    }
    printf("i = %f\n", i);
    return 0;
}
```

Le résultat est 99,999046, ce qui montre que la précision du type `float` est en général mauvaise, d'autant plus que le nombre 0,1 n'est pas représentable de manière exacte en binaire[4]. Il est ainsi conseillé d'utiliser le type `double` à la place de `float` autant que possible. Dans ce cas de figure, il est préférable d'éviter les accumulations d'erreurs infinitésimales, en réécrivant le code de la manière suivante :

```c
#include <stdio.h>

int main(void)
{
    float i;
    int j = 1000;

    i = j * 0.1;

```

```
10      printf("i = %f\n", i);
11      return 0;
12  }
```

Le résultat est alors 100.0, sans aucune erreur d'accumulation et d'arrondis.

Pour plus d'informations sur ce domaine, un wikilivre Arithmétique flottante est disponible.

Caractères

À l'origine, le type permettant de représenter un caractère est `char`. Même si un `char` n'est plus toujours suffisant aujourd'hui pour représenter un caractère quelconque.

Ce type est un peu plus particulier que les autres, d'une part parce que sa taille définit l'unité de calcul pour les quantités de mémoire (et donc pour les tailles des autres types du langage) et d'autre part son domaine de valeur peut grandement varier de manière relativement inattendue.

Par définition, la taille du type `char`, notée `sizeof(char)`, vaut toujours 1. Cependant, il faut faire attention : contrairement à ce qu'on pense souvent, un `char` au sens du C ne vaut pas toujours un *octet*. Il occupera au minimum 8 bits, mais il existe des architectures, relativement spécialisées il est vrai, ayant des `char` de 9 bits, de 16 bits, voire plus. Même si, dans une large majorité des cas, les compilateurs utilisent des `char` de 8 bits, à la fois par simplicité (les machines modernes fonctionnent généralement en 8, 16, 32 ou 64 bits) et pour éviter des problèmes de portabilité de code (beaucoup de codes C existants reposent sur l'hypothèse que les `char` font 8 bits, et risqueraient de ne pas marcher sur une autre architecture)[5]. Par simplification, nous utiliserons donc le terme *octet* la plupart du temps dans la suite de ce wikilivre.

Un autre piège de ce type est qu'il peut être **de base** (c'est-à-dire implicitement) `signed` ou `unsigned`, au choix du compilateur, ce qui peut s'avérer dangereux (c'est-à-dire difficile à maitriser). Considérez le code suivant :

```
/* Ce code peut ne pas fonctionner avec certains compilateurs */
char i;
for (i = 100; i >= 0; i --)
{
    /* ... */
}
```

Dans cet exemple, l'instruction `for` permet de faire itérer les valeurs entières de `i` de 100 à 0, incluses. On pourrait naïvement penser **optimiser** en utilisant un type `char`. Sauf que si ce type est implicitement `unsigned`, la condition `i >= 0` sera toujours vraie, et tout ce que vous obtiendrez est une boucle infinie. Normalement, tout bon compilateur devrait vous avertir que la condition est toujours vraie et donc vous permettre de corriger en conséquence, plutôt que perdre des heures en débugage.

Constantes représentant un caractère

Une constante représentant un caractère (de type char) est délimitée par des apostrophes, comme par exemple `'a'`. En fait le langage C n'a initialement pas été conçu pour utiliser tous les caractères. Le type char permet de représenter ni plus ni moins que des nombres entiers (ils sont de type `int`, mais leur valeur tiendra dans un type `char`), les deux étant parfaitement interchangeables. Cependant, comme dit plus haut, il ne s'agit généralement pas là des caractères Unicode que nous manipulons tous les jours, mais de caractère-octet dont la portée est réduite.

Un petit exemple :

```c
#include <stdio.h>

int main(void)
{
    printf("Sur votre machine, la lettre 'a' a pour code %d.\n", 'a');
    return 0;
}
```

Le programme précédent donnera le résultat suivant dans un environnement ASCII :

```
Sur votre machine, la lettre 'a' a pour code 97.
```

Il est tout à fait autorisé d'écrire `'a' * 2` ou `'a' - 32`.

Cette soustraction permet de convertir une minuscule ASCII en majuscule si le codage ASCII est utilisé, mais ne le permet pas si un autre codage est utilisé (comme ISO-8859-1 ou UTF-8).

Valeur des caractères-octets

La valeur représentée par cette constante est néanmoins dépendante des conventions de codages de caractères employées. À l'origine ces conventions étaient connues du système ; aujourd'hui, même si dans une écrasante majorité des cas, on se retrouvera avec un jeu de caractères augmentant l'ASCII et donc l'ISO-646. Le code ASCII définit 96 glyphes de caractères portant les numéros 32 à 126, bien loin des milliers de caractères nécessaires pour les logiciels fonctionnant sur la planète entière et dans de nombreuses langues[6]. Cet ensemble est à peine suffisant pour couvrir l'anglais alors que plusieurs langues latines étaient visées à l'époque où l'ASCII et l'ISO-646 ont été définis, si bien que de nombreuses extensions sont par la suite apparues.

Par exemple le caractère « œ » (ligature du o et du e) a pour valeur :

- 189 dans le jeu de caractères ISO-8859-15 (principalement utilisé pour les langues latines d'Europe, sous Unix) ;
- 156 sur certaines variantes du jeu de caractères Windows 1252 (principalement utilisé pour les langues latines d'Europe, sous Windows) ;
- 0xc5, 0x93 (deux octets, donc deux `char`) en UTF-8 ;
- 207 avec l'encodage Mac Roman (Mac OS 9 et antérieur) ;
- Et n'a pas d'équivalent ni en ISO-8859-1 ni en ASCII (le caractère 189 est le symbole « ½ », le 207 est le « Ï » et le 156 n'est pas utilisé).

Retenons simplement qu'Unicode est omniprésent sur Internet, mais que des logiciels plus anciens ou embarqués peuvent fonctionner avec des jeux de caractères huit bits, plus limités. Il s'agit en fait d'une problématique concernant des sujets plus vastes, comme l' *internationalisation*, la *portabilité* et l' *interopérabilité*.

Le langage C reste relativement abstrait à ce niveau : le type char contient généralement un octet et les chaînes, une simple suite d'octets terminée par 0. Il laisse au système et au développeur le soin d'interpréter et de traiter les octets comme il se doit. Ceci peut être fait à l'aide de bibliothèques appropriées. Si on trouve ici ou là de telles listes de bibliothèques [7], il est préférable de connaître le langage C avant de les utiliser.

Cet abstraction du langage C lui a permis de s'adapter aux nombreuses évolutions des conventions de codages des caractères.

Caractères non graphiques

Certains caractères ne sont pas graphiques. Par définition, on ne peut pas donc les écrire dans un code source de manière visible. Le langage C adopte la convention suivant pour désigner certains d'entre eux de manière littérale :

Constante	Caractère
`'\''`	une apostrophe
`'\"'`	un guillemet
`'\?'`	un point d'interrogation
`'\\'`	un backslash
`'\a'`	un signal sonore (ou visuel)
`'\b'`	un espace arrière
`'\f'`	saut au début de la page suivante
`'\n'`	saut de ligne
`'\r'`	un retour chariot
`'\t'`	une tabulation (pose problème avec gcc-4.8)
`'\v'`	une tabulation verticale (pose problème avec gcc-4.8)

De plus, on peut écrire n'importe quelle valeur de caractère avec les expressions suivantes :

- `'\x`*HH*`'`, où chaque *H* représente un chiffre hexadécimal correspondant au code du caractère. Par exemple, `'\x61'` représente le caractère `'a'` (minuscule) en ASCII (car 97 = 6 * 16 + 1) ;
- `"\xc5\x93"` représente le caractère œ en UTF-8.
- `'\`*000*`'`, où chaque *0* représente un chiffre octal correspondant au code du caractère.

Trigraphe

Le langage C date d'une époque où le standard de codage des caractères était l'ISO-646, un standard encore plus ancien et plus incomplet que l'ASCII. Les caractères absent de l'ISO-646 pouvaient être simulés par une séquence de trois caractères.

L'ASCII s'étant répandu, cette fonctionnalité tombée en désuétude est en général à éviter, car très souvent inutile, et peut causer des bugs incompréhensibles au programmeur non averti. Néanmoins, de très rares programmes peuvent utiliser ce genre de fonctionnalités. Un trigraphe est simplement une suite de trois caractères dans le code source qui sera remplacée par un seul.

Cette fonctionnalité a été ajoutée au C pour supporter les architectures (systèmes) dont l'alphabet ne dispose pas de certains caractères qui sont nécessaires dans la syntaxe du C, comme les dièses ou les accolades.

Les substitutions suivantes sont faites *partout* dans le code source (y compris les chaînes de caractères) :

Trigraphe	Caractère
`??=`	`#`
`??(`	`[`
`??)`	`]`
`??<`	`{`
`??>`	`}`
`??/`	`\`
`??'`	`^`
`??!`	`|`
`??-`	`~`

Voici une manière de rendre illisible un programme utilisant les trigraphes :

```
??=include <stdio.h>

int main(void)
??<
   puts("Bonjour !");
```

```
    return 0;
??>
```

Par défaut, la plupart des compilateurs désactivent les trigraphes, au cas où vous ne seriez pas encore dissuadé de les utiliser.

Chaîne de caractères

Une chaîne de caractère, comme son nom l'indique, est une suite de caractères avec la particularité d'avoir un caractère nul (0) à la fin. Une chaîne de caractère est en fait implémentée en C avec un tableau de type `char`.

Ce chapitre vous donne les rudiments sur les chaînes de caractère pour une première approche. Pour en savoir plus sur

- les rares fonctions de chaînes de caractères, voir Programmation C/Chaînes de caractères
- les tableaux & pointeurs : Programmation_C/Tableaux#Cha.C3.AEnes_de_caract.C3.A8res

```
Programmation_C/Tableaux#Chaînes_de_caractères
```

Structure

En mémoire, une chaîne de caractère est représenté dans le langage C comme un tableau d'octet dont les valeurs dépendent de l'encodage utilisé ici, Unicode.

Amélie en Unicode formes NFC et NFD

Caractère représenté	A	m	é		l	i	e	'\0'
Unicode NFC	0041	006d	00e9		006c	0069	0065	0
Unicode UTF-8 NFC	41	6d	c3 a9		6c	69	65	0
Unicode NFD	0041	006d	0065	0301	006c	0069	0065	0
Unicode UTF-8 NFD	41	6d	65	cc 81	6c	69	65	0
Unicode NFD	A	m	e	◌́	l	i	e	'\0'

Chaîne littérale

On appelle une chaîne littérale la manière dont est définie une constante chaîne dans un code source. Sous sa forme la plus simple, on déclare une chaîne comme une suite de caractères entre guillemets (double quote) :

```
"Ceci est une chaîne de caractère";
""; /* Chaîne vide */
```

Si la chaîne est trop longue, on peut aussi la couper sur plusieurs lignes :

```
"Ceci est une chaîne de caractère, " /* pas de ; */
"déclarée sur plusieurs lignes.";
```

Deux chaînes côtes à côtes (modulo les espaces et les commentaires) seront concaténées par le compilateur. De plus on peut utiliser le caractère barre oblique inverse (\) pour annuler la signification spéciale de certains caractères ou utiliser des caractères spéciaux (C.f liste ci-dessus).

```
"Une chaîne avec des \"guillemets\" et une barre oblique (\\)\n";
```

Il est aussi possible d'utiliser la notation hexadécimale et octale pour décrire des caractéres dans la chaîne, il faut néanmoins faire attention avec la notation hexadécimale, car la définition peut s'étendre sur plus de 2 caractéres. En fait, tous les caractères suivant le `\x` et appartenant à l'ensemble `"0123456789abcdefABCDEF"` seront utilisés pour déterminer la valeur du caractère (du fait que les caractères d'une chaîne peuvent faire plus qu'un octet). Cela peut produire certains effets de bords comme :

```
/* Ce code contient un effet de bord inattendu */
"\x00abcdefghijklmnopqrstuvzxyz"
```

Ce code n'insèrera pas un caractère 0 au début de la chaine, mais toute la séquence `"\x00abcdef"` servira à calculer la valeur du caractère (même si le résultat sera tronqué pour tenir dans un type `char`). On peut éviter cela en utilisant la concaténation de chaînes constantes :

```
"\x00" "abcdefghijklmnopqrstuvzxyz"
```

Enfin, les chaînes de caractères faisant appel aux concepts de pointeur, tableau et de zone mémoire statique, leur utilisation plus poussée sera décrite dans la section dédiée aux tableaux.

À noter la représentation **obsolète** des chaînes de caractères multilignes. À éviter dans la mesure du possible :

```
/* Ce code est obsolete et a eviter */
"Ceci est une chaîne de caractère,\
déclarée sur plusieurs lignes";
```

Booléens

Le langage (jusqu'à la norme C99) ne fournit pas de type booléen. La valeur entière 0 prend la valeur de vérité *faux* et toutes les autres valeurs entières prennent la valeur de vérité *vrai*.

La norme C99 a introduit le type `_Bool`, qui peut contenir les valeurs 0 et 1. Elle a aussi ajouté l'en-tête `<stdbool.h>`, qui définit le type `bool` qui est un raccourci pour `_Bool`, et les valeurs `true` et `false`[8]. Néanmoins, ces nouveautés du C99 ne sont pas très utilisées, les habitudes ayant été prises d'utiliser 0 et *différent de zéro* pour les booléens en C.

Nota : toute expression utilisant des opérateurs booléens (voir opérateurs), retourne 1 si l'expression est vraie et 0 si elle est fausse, ce qui rend quasiment inutile l'usage du type booléen.

Un booléen en un entier ont des règles de conversion différente, une valeur flottante comprise entre 0 et 1 non inclus, n'aura pas la même valeur selon que le récipient soit booléen ou entier. La valeur flottante NaN, étant différente de 0 sera vraie dans les deux cas.

Vide

En plus de ces types, le langage C fournit un autre type, `void` qui représente *rien*, le *vide*. Il n'est pas possible de déclarer une variable de type `void`. Nous verrons l'utilité de ce type lorsque nous parlerons de fonctions et de pointeurs.

Notes et références

1. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf
2. La norme C contraint les domaines de valeurs des types signés entre $-(2^{N-1}-1)$ et $2^{N-1}-1$, où N est un entier quelconque, et les types non signés entre 0 et 2^N-1.
3. La liste des types successifs utilisés pour déterminer le type d'une constante entière a changé entre C90 et C99. Cela n'a, quasiment tout le temps, aucune incidence, sauf pour le cas d'une constante de valeur trop grande pour le type `long` : en C90, le type `unsigned long` est utilisé, alors qu'en C99 le type `long long` sera utilisé. Une telle valeur sera alors signée dans un cas, et non signée dans l'autre. Dans ce cas, utiliser un suffixe adapté (`UL`, `LL` ou `ULL`) permet de s'assurer du type de la constante.
4. $0{,}1_{10} = 0.0001100110011..._2$
5. La confusion est surtout née de l'utilisation du mot *byte* pour désigner la taille du type *char* dans la norme, alors que ce mot est utilisé dans presque tous les autres domaines de l'informatique pour désigner un octet.
6. En occident, en-dehors des codages compatibles avec ASCII, on ne rencontre guère en pratique que la famille de jeux de caractères EBCDIC, utilisée sur des systèmes IBM et quelques mainframes.
7. Le site Unicode publie une liste de bibliothèque de gestion des caractères et chaînes Unicodes. http://www.unicode.org/resources/libraries.html
8. Ces trois expressions sont définies sous formes de macro pour le préprocesseur.
