



De mars à juin
dans toute
la France

Pour vous inscrire,
cliquez-ici

Visual Studio 2005
bêta 2 et SQL Server
2005 bêta 3 OFFERTS



programmation

Cours

Livres

Recherche

Arcane

Progworld

Forum



w o r l d

► **Accueil**

Accueil

Page des cours

Bibliothèque

News

**Sommaire
rubrique**

Cours précédent

**Langage C : Hello
World C !**

Cours suivant

**Langage C : Les
entrées et sorties
en C**



Les types de données

Nous allons apprendre dans ce chapitre à manipuler les types de données fondamentaux du langage C. Nous examinerons de même la façon dont le C conserve ces données en mémoire.

Sommaire

1. La mémoire d'un ordinateur
2. Les types
3. Où se trouvent les données manipulées par C ?
4. Présentation des différents types du C
5. Les types entiers
6. unsigned et signed : explication
7. Le type short et unsigned short
8. Le type int et unsigned int
9. Le type long et unsigned long
10. Le type char
11. Les types à virgule flottante
12. Le type float
13. Le type double
14. Les identificateurs
15. Savoir écrire un identificateur
16. Les mots clés
17. Les constantes
18. Constantes non nommées
19. Constantes non nommées entières



**SQL Server
2005**

**Plus de
fonctionnalités,
plus de
productivité.**

**Ce sont
vos mains
qui vont
être contentes !**

Inscrivez-vous aux
conférences gratuites
pour découvrir toutes
les nouveautés
SQL Server 2005.

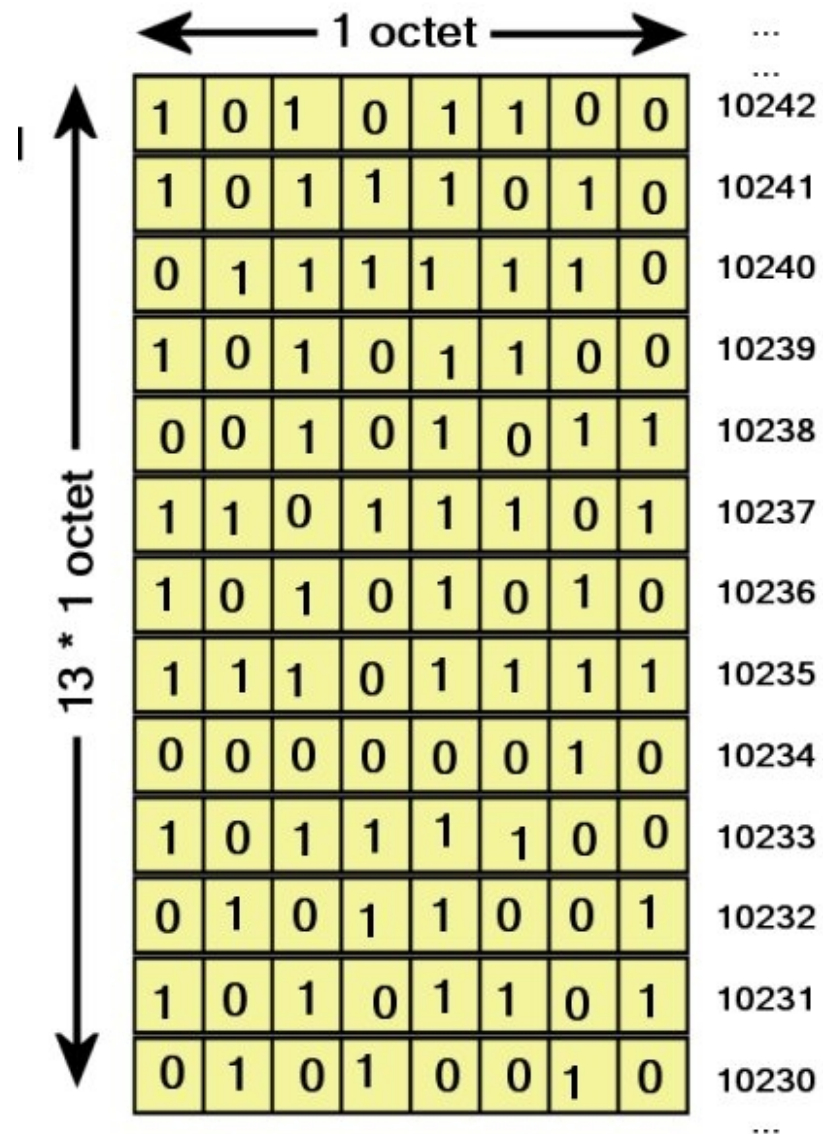
Cliquez ici !



- 20. [Constantes non nommées réelles](#)
- 21. [Constantes caractères](#)
- 22. [Les séquences d'échappement](#)
- 23. [Constantes string \(chaînes de caractères\)](#)
- 24. [Constantes nommées](#)
- 25. [Les variables](#)
- 26. [Initialisation](#)
- 27. [Affectation](#)
- 28. [Portée des identificateurs de variables ou de constantes](#)
- 29. [Annexe](#)

La mémoire d'un ordinateur

La mémoire d'un ordinateur est un ensemble "d'états binaires" qui se nomment "bits" (8 bits donnent un octet). Un bit peut indifféremment prendre 0 ou 1 comme valeur. Il va de soi qu'un ordinateur peut mémoriser bien plus qu'un simple bit d'information. Il manipule généralement les données par tranches de format fixe qu'on appelle mots. La taille d'un mot varie d'un ordinateur à l'autre. Sur les plus anciens, elle était de 8 ou 16 bits (soit un ou deux octets) mais on arrive aujourd'hui à 32 bits (soit quatre octets). Les ordinateurs ont besoin de stocker de grandes quantités d'informations, telles que les instructions d'un programme ou les données que ces instructions manipulent. La mémoire centrale d'un PC est ainsi très importante. Elle contient environ 128.10^6 cellules. Une cellule mémoire est une petite unité dont la taille est d'un octet. Pour des raisons pratiques, ces cellules sont numérotées consécutivement à partir de 0 (0, 1, 2, 3, 4, 5...). Ces numéros sont appelés adresses et chaque cellule a une adresse unique. L'adresse 0 est dite null. C'est dans ces cellules que seront stockées les données et les instructions d'un programme.



Une portion de 13 octets de la mémoire d'un ordinateur. Chaque octet a son adresse.

Les types

Comment un ordinateur peut savoir, avec une mémoire composée de milliards de 0 et de 1, où commence un nombre et où il finit ? De même, comment peut-il savoir si ce nombre est entier, flottant, ou s'il s'agit d'un caractère ? C'est là le rôle d'un type. Indiquer au compilateur où se trouve une variable, comment celle-ci est codée en mémoire, et la taille de la portion qu'elle y occupe. Pour stocker une donnée dans une cellule il sera nécessaire de fournir à la mémoire une adresse ou de stocker l'information en plus de la donnée elle-même. De nombreux types de données peuvent tenir en un seul mot. D'autres, plus petits, n'en utilisent qu'une partie. Ne vous inquiétez pas, cette opération de stockage ou de lecture est extrêmement simple.

Où se trouvent les données manipulées par C ?

Nos programmes peuvent manipuler des données se trouvant dans trois endroits différents. C'est la façon dont elles seront utilisées qui déterminera leur emplacement. Les données peuvent se situer tout d'abord dans une zone de la mémoire spéciale que l'on appelle la pile ou stack en anglais. Il s'agit d'une structure où les éléments fonctionnent comme une pile d'assiettes dans un placard ; la dernière assiette posée sur la pile est aussi la première à en être retirée. La première assiette mise sous la pile est la dernière à être retirée. Ce genre de données sert principalement pour l'exécution de tâches rapides, c'est pourquoi ce sont elles qui sont créées à l'intérieur des méthodes. Le C peut aussi manipuler des données dans une zone de la mémoire nommée tas ou heap dont la taille peut varier dynamiquement. C'est dans celle-ci que sont placés les données globales (accessibles par n'importe quelle fonction d'un programme). Elle s'étend au fur et à mesure des besoins du programme (dans les limites de l'espace mémoire). A la différence des langages comme le C# ou le Java, c'est au programmeur de gérer cette mémoire. Ceci implique donc forcément un risque d'erreur plus grand. Enfin, évidemment les données peuvent être stockées de manière permanente lorsqu'elles se trouvent sur le disque dur ou sur des zones de mémoire morte. Ces données persistent après l'arrêt de l'ordinateur.

Présentation des différents types du C

En C, les types sont soit prédéfinis, soit définis par le programmeur lui-même. Les types prédéfinis sont appelés types de données fondamentaux. Les types créés par le programmeur peuvent être soit des pointeurs, des énumérations, des tableaux (arrays), ou bien évidemment des structures. Il existe en fondamental plusieurs sortes de types :

- Les entiers dont les spécificateurs sont int, short, long, char, unsigned int, unsigned short int, unsigned long int.
- Les flottants (nombres à virgules) dont les spécificateurs sont float, double, long double.



Astuce

Avant de commencer notre étude des différents types fondamentaux existant en C, nous ne saurions trop vous conseiller de vous rendre dans la partie "Introduction aux systèmes numériques" afin de lire le cours sur le système binaire.

Les types entiers

Pour la représentation des nombres entiers, c'est-à-dire ceux dépourvus de décimales, le C permet l'utilisation des types int, short, long, char, unsigned short int, unsigned int, unsigned long int. Le préfixe unsigned indique que les nombres peuvent être mémorisés sans signe. Il est possible de stocker les entiers et les caractères en mémoire, avec ou sans signe. Il suffit de faire précéder le spécificateur de la clause signed (pour signé) ou unsigned (pour non signé). Par défaut, le compilateur estime que toutes les données sont signées. Il est donc équivalent d'écrire :

- "signed int" et "int"
- "signed short int" et "short int"
- "signed char" et "char"
- "signed long int" et "long int"

Il existe aussi d'autres équivalences que la [première annexe](#) montre. Il est important de lire cette annexe pour comprendre que pour le compilateur par exemple, signed long int est aussi équivalent à signed long mais aussi à long. Si au contraire nous voulons une donnée sans signe nous écrirons : unsigned int, unsigned long... Cette donnée sera alors une valeur uniquement positive. Voyons tout de suite l'intérêt relatif des signes.

unsigned et signed : explication

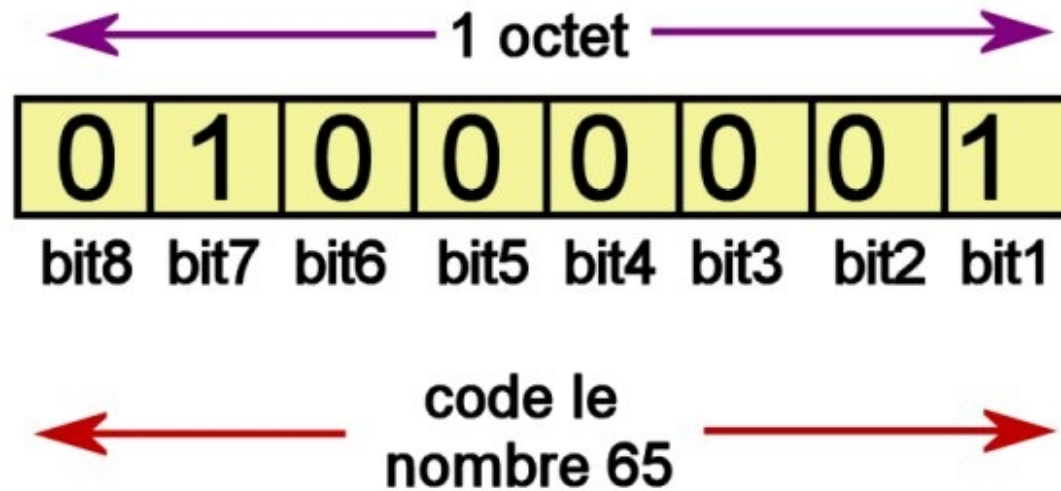
Nous allons tenter de comprendre l'utilité des mots clés signed et unsigned en prenant des exemples avec le type char. Une donnée de type char occupe un octet (ou byte en anglais) en mémoire soit 8 bits, elle est utilisée pour stocker un entier faible compris entre 0 et 255 (c'est-à-dire les chiffres compris entre 0 et 2^8-1) soit un nombre positif exclusivement. Ce nombre peut servir à coder un caractère (nous reviendrons sur ce principe plus loin). Contrairement au type char que nous allons voir après, la mémoire occupée par un type unsigned char ne dépend pas de l'existence d'un signe. Le nombre codé en binaire est codé avec la totalité des 8 octets. Il n'est pas nécessaire de réserver un emplacement pour coder le signe. D'un point de vue mathématique, ce nombre sera positif, pour l'ordinateur il n'aura simplement pas de signe. Chacun de ces bits peut prendre la valeur 0 ou 1, c'est-à-dire deux états possibles huit fois de suite, soit 2^8 (256) combinaisons possibles, la première étant :

```
0000 0000
```

ce qui correspond au nombre 0, et la dernière étant :

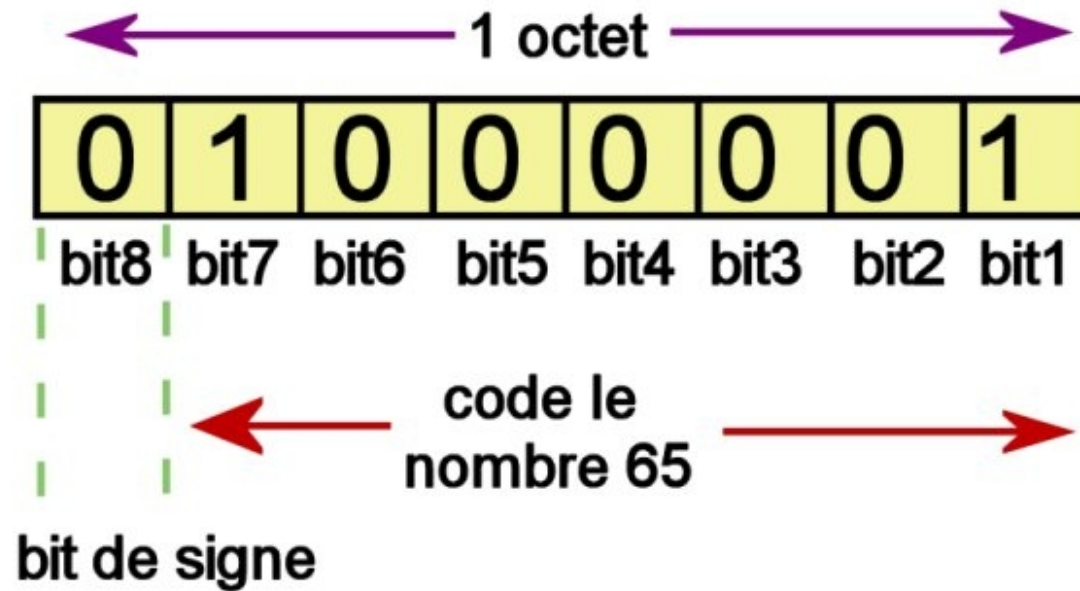
```
1111 1111
```

c'est-à-dire le nombre 255.



Codage du nombre 65 en décimal (ou 01000001 en binaire).

A l'inverse du type unsigned char, char permet le stockage de données sur 8 bits signées. Ces données sont des nombres codés sur 7 bits, le huitième bit étant utilisé pour contenir une information de signe.



Structure du type unsigned char en mémoire.

Le nombre en mémoire est considéré comme négatif ou positif selon la valeur de ce huitième bit. Si celui-ci est égal à 1 alors le nombre est considéré comme négatif, s'il vaut 0 alors le nombre est positif. Il ne reste donc plus pour les nombres positifs que 128 valeurs représentables soit de 0 à 127 (de 00000000 à 01111111). A l'opposé, les combinaisons 10000000 à 11111111 représentent les nombres négatifs -1 à -128.

binaire	décimal	
0 000 0000	0	positif
0 000 0001	1	
0 000 0010	2	
0 000 0011	3	
...	...	
0 111 1100	124	
0 111 1101	125	
0 111 1110	126	
0 111 1111	127	
<hr/>		
1 000 0000	-128	negatif
1 000 0001	-127	
1 000 0010	-126	
1 000 0011	-125	
...	...	
1 111 1100	-4	
1 111 1101	-3	
1 111 1110	-2	
1 111 1111	-1	

Représentation des nombres signés 8 bits.

La représentation des nombres négatifs est appelée méthode du complément à deux. Le complément à deux d'un nombre binaire b est le nombre b' tel que

$$b + b' = 2^n$$

n représente le nombre de bits du nombre binaire. Dans le cas du type unsigned char, n vaut 8. Le complément à deux d'un nombre binaire s'obtient en remplaçant chaque 0 de la combinaison binaire le représentant par 1, et chaque 1 par 0, puis en ajoutant 1 au résultat. Calculons le complément à deux de 1 (qui est -1) Le nombre 1 en char est codé ainsi en mémoire :

```
00000001
```

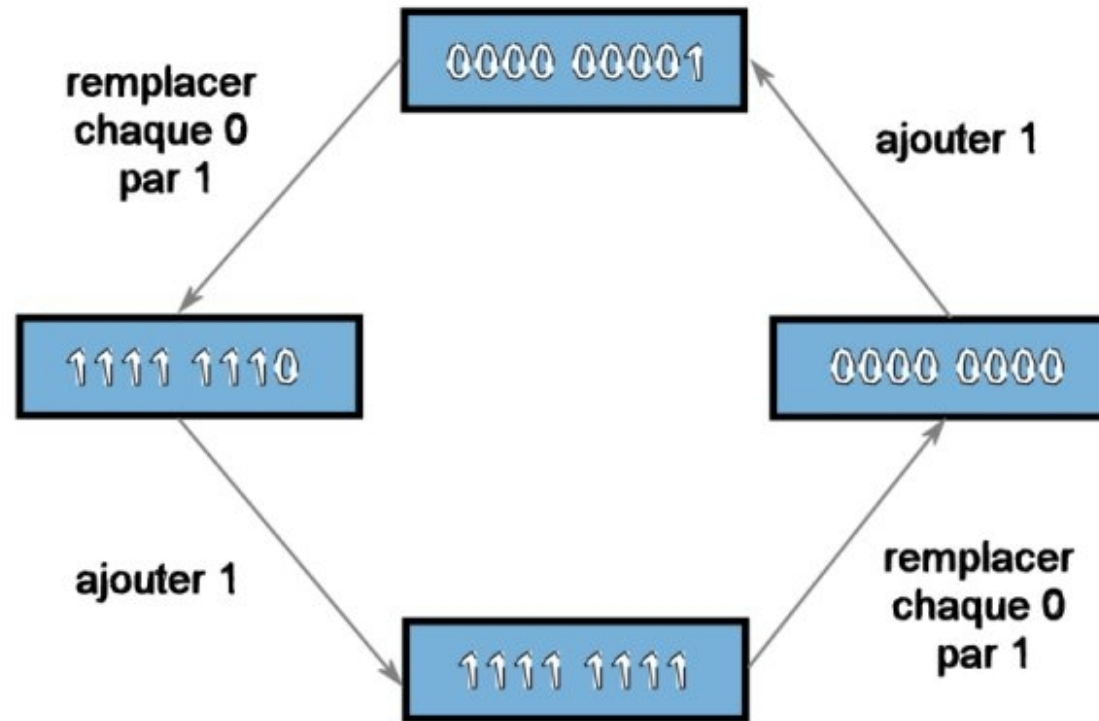
Si on change chaque bits 0 en 1 et vice et versa on obtient :

```
11111110
```

on lui rajoute alors 1 :

```
11111111
```

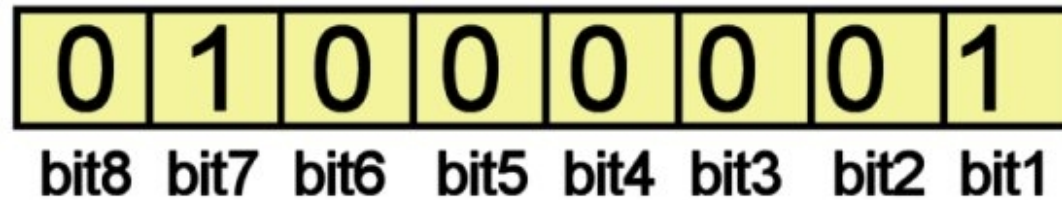
pour obtenir la combinaison de bits représentant le nombre -1. Si vous cherchez le complément à deux de la suite binaire 11111111 vous obtiendrez évidemment 00000001. En additionnant 11111111 à 00000001 on obtient le nombre 256 qui est égal à 2^8 .



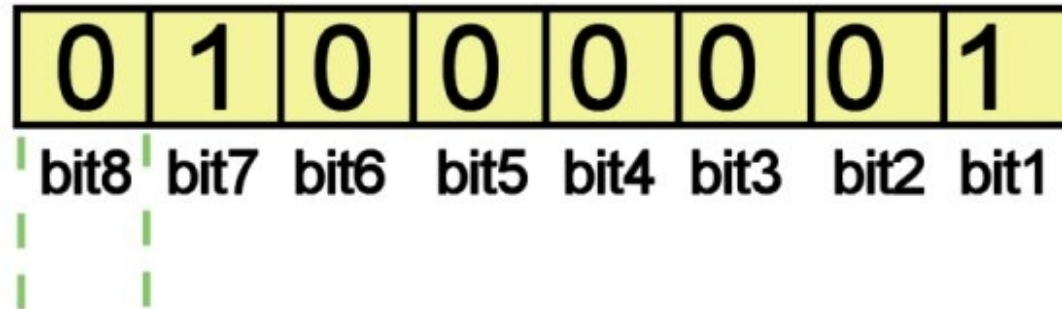
Passage d'un nombre positif à son opposé négatif et inversement.

Il convient de rester prudent quant au stockage de nombres avec le type unsigned char. Voyons le stockage de deux nombres 65 et 227 avec les types char et unsigned char.

65 en type char



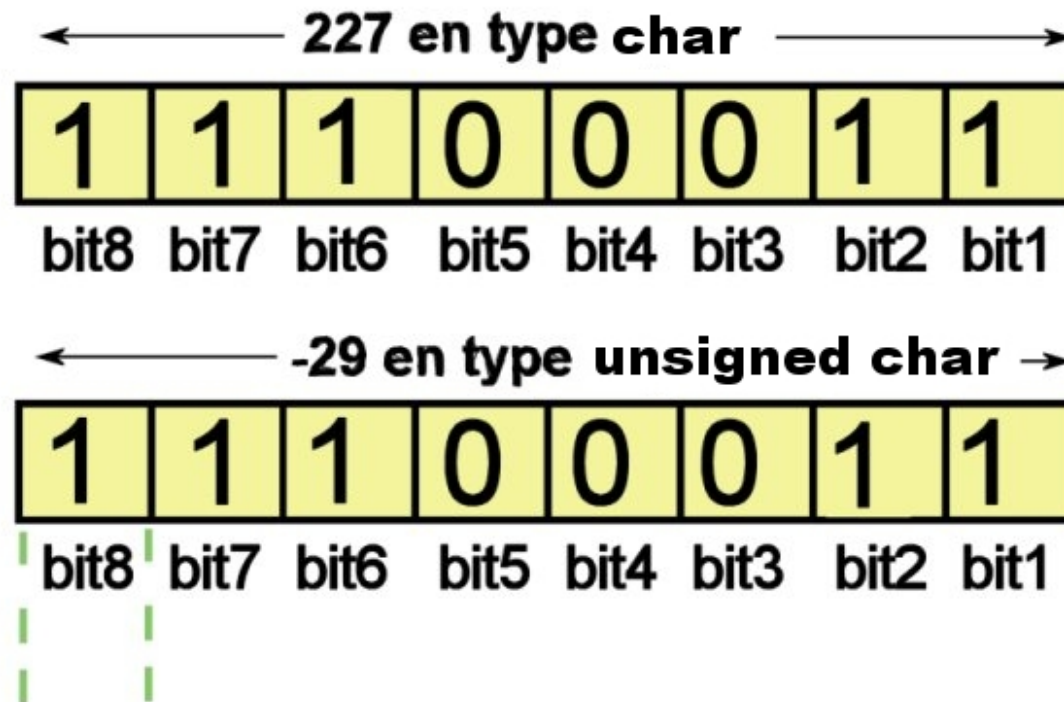
65 en type unsigned char



bit de signe

Le codage de 65 avec les types char et unsigned char ne pose pas de problème.

65 étant inférieur à 127, il est représentable sur 7 bits et ne modifie pas le huitième bit de signe. Celui-ci ne servant pas reste à 0, laissant le nombre en positif. Pourtant, si on prend un nombre qui ne peut être codé que sur 8 bits du fait de sa grandeur comme 227, il va se poser un problème :



bit de signe

Le codage de 227 avec les types char et unsigned char pose problème.

En unsigned char on obtient 227 car le huitième bit sert au codage du nombre mais pas à donner le signe. En char, ce bit va servir à indiquer le signe du nombre codé par les sept autres bits. La combinaison binaire qui était égale à 227 en unsigned char va être lue comme -29 par le C (le négatif venant du huitième bit à 1).

Le type short et unsigned short

unsigned short et short occupent en mémoire 2 octets soit 16 bits, c'est-à-dire 2^{16} (65536) nombres différents. Le type unsigned short étant non signé, permet de représenter les nombres dans l'intervalle 0 à 65535, short permet de représenter les nombres positifs par l'intermédiaire du bit de signe (le seizième):

0000 0000 0000 0000

à

```
0111 1111 1111 1111
```

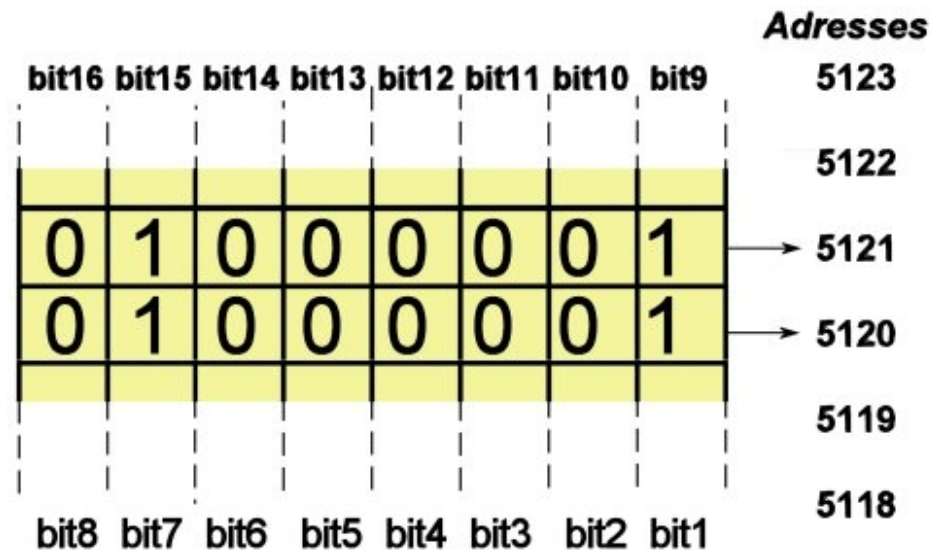
C'est-à-dire les nombres 0 à 32767 et les nombres négatifs -1 à -32768 :

```
1111 1111 1111 1111
```

et

```
1000 0000 0000 0000
```

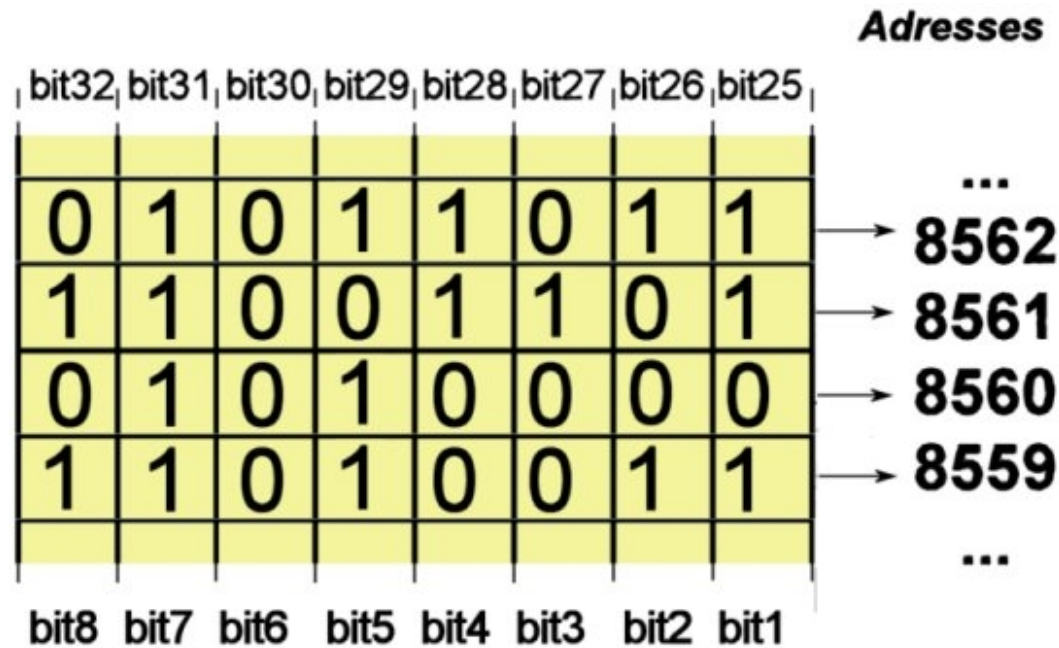
short permet donc de mémoriser un nombre dans un intervalle compris entre -32768 et 32767.



Représentation d'un short ou unsigned short en mémoire ; ici les deux octets du type ont pris arbitrairement les valeurs 5120 et 5121.

Le type int et unsigned int

Le type int et son équivalent non signé unsigned int représentent les types entiers standards qui sont le plus souvent utilisés par les programmeurs. Ils occupent en mémoire 4 octets soit 32 bits. Ils fonctionnent sur le même principe que short et ushort, mais la plage des nombres représentables est beaucoup plus importante puisqu'il est possible de combiner 2^{32} (4294967296) entiers. uint représentera les entiers entre 0 et 4294967295, et int les entiers positifs entre 0 et 2147483647 et négatifs entre -1 et 2147483648.



Représentation d'un int ou unsigned int en mémoire.



Question

Sur certains systèmes anciens le type int occupe uniquement 2 octets, il convient donc de faire attention. Le programmeur C peut néanmoins être certain que le type short occupera 2 octets quel que soit le système, et que le type long occupera 4 octets quel que soit le système.

Le type long et unsigned long

Les types long et unsigned long peuvent stocker des nombres sur 4 octets soit la même capacité que le type int vu précédemment.

Le type char

Le type de données char (de l'anglais caractère) occupe une place de 1 octet en mémoire. Il sert à manipuler les caractères par l'intermédiaire du code ASCII. Ce dernier est un système 16 bits permettant de représenter 128 (2^7) caractères pour ASCII simple et 256 (2^8) caractères pour l'ASCII étendu.



Remarque

Code ASCII Les 128 premiers caractères d'Unicode sont ceux du code ASCII

Chaque nombre de cet intervalle correspond à un caractère ou à un idéogramme permettant de coder les signes de toutes les langues du monde. Par exemple la lettre A possède la valeur 65 dans le code Unicode/ASCII.



Astuce

La page située [ici](#) contient tous les codes ASCII et ASCII étendu.

Les types à virgule flottante

Les types à virgules flottantes (appelés également réels ou tout simplement flottants) permettent de stocker en mémoire les nombres qui possèdent des décimales comme 3.14 ou -0.01258. En C la virgule est représentée par un point décimal. Ce point peut se déplacer en n'importe quel endroit du nombre :

45.123

est équivalent à

4.5123*10¹

mais aussi à

451.23*10⁻¹

C'est la multiplication par différentes puissances de dix qui fait que ces diverses représentations sont une seule et même valeur. En C la représentation des nombres flottants est différente des nombres entiers. L'emplacement mémoire occupé par un nombre flottant est divisé en trois parties : le signe, l'exposant et la mantisse :



Représentation des nombres réels en mémoire.

La partie signe fonctionne de la même manière que le bit de signe des types entiers, soit à 1 pour les nombres négatifs et à 0 pour les positifs. La mantisse est la suite de chiffres qui constitue le nombre flottant. Dans le nombre 45.123 la mantisse serait ainsi égale à 45123. Enfin l'exposant est la puissance nécessaire pour écrire le nombre dans la forme voulue. Cette forme voulue étant :

$0.\text{mantisse} * 10^{\text{exposant}}$

c'est-à-dire en forme binaire :

$0.\text{mantisse-codée-en-binaire} * 2^{\text{exposant}}$

notre nombre 45.123 s'écrit ainsi en décimal :

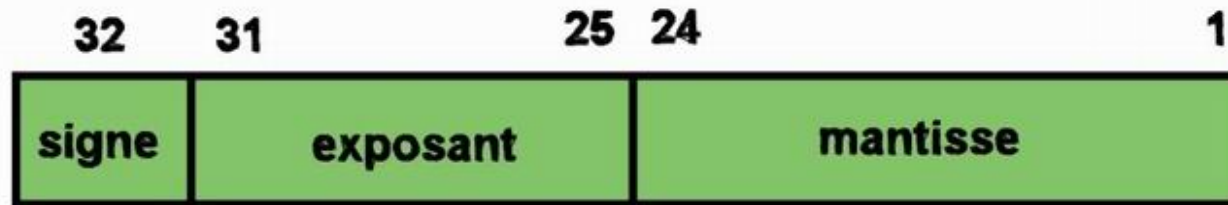
$0.45123 * 10^2$

et en binaire

$0.0101101 * 2^5$

Le type float

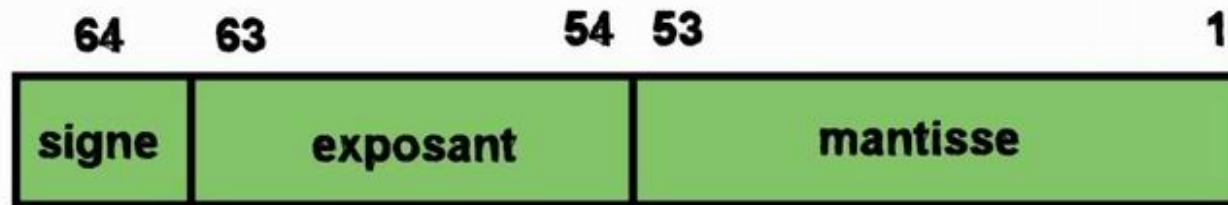
Le type float occupe 4 octets en mémoire, divisés en trois parties énoncées ci-dessous. Les 32 bits sont divisés en 24 bits pour coder la mantisse, 7 bits pour coder l'exposant et 1 bit de signe. Le type float peut mémoriser des nombres compris entre $3.4 \cdot 10^{-38}$ et $3.4 \cdot 10^{38}$ avec une précision de 7 digits.



Représentation d'un flottant en mémoire.

Le type double

Le type double occupe 8 octets en mémoire avec une mantisse de 53 bits et un exposant de 10 bits assurant une couverture des nombres de $1.7 \cdot 10^{-308}$ à $1.7 \cdot 10^{308}$ offrant une précision de 15 à 16 digits.



Représentation d'un double en mémoire.

Les identificateurs

Il est possible d'attribuer à une donnée un identificateur. Cet identificateur doit cependant respecter certaines règles édictées par le langage. Celles-ci s'appliquent aussi bien aux noms attribués aux données qu'à ceux des handles.

Savoir écrire un identificateur

En principe un identificateur doit être constitué d'un ou plusieurs caractères. Ceux-ci peuvent être des lettres, des chiffres, ou bien l'underscore (_) ; le premier caractère de l'identificateur devant être soit une lettre soit un underscore mais jamais un chiffre.

```

pepette           //bon
pepette2          //bon
2pepette          //mauvais : commence par un chiffre
MA_VARIABLE       //bon
MES_2variables    //bon
fichier_13        //bon
#_de_participants //mauvais : utilisation du caractère #
nombre de participants //mauvais : utilisation d'un caractère d'espacement
nombre-1          //mauvais : utilisation du caractère -
    
```

Le compilateur fait aussi une distinction entre les majuscules et les minuscules, ce qui est souvent source d'erreurs au moment de la compilation.

```

pepette
    
```

est ainsi différent de

```

Pepette
    
```

Les mots clés

Les identificateurs sont aussi soumis à une autre forme de restriction ; ils ne doivent surtout pas correspondre à un mot clé du langage C. Ainsi, il serait faux d'utiliser Main ou return en tant qu'identificateurs. Le C dispose de 34 mots clés ; le tableau ci-dessous les énumère :

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

nous y retrouvons évidemment les types float, int, long, etc... dont nous avons parlé précédemment. La signification des mots clés que contient le tableau ci-dessus sera évidemment expliquée par la suite.

Les constantes

Les données ne se caractérisent pas uniquement par leur type (ce que nous venons de voir) ou leur valeur, mais aussi par les constantes et les variables. Ces deux derniers éléments vont nous permettre de modifier leur valeur. Il y a deux catégories de constantes, les non nommées qui ont une valeur et un type, et les constantes nommées qui possèdent en plus un nom pour les manipuler (à la manière des handles).

Constantes non nommées

Les constantes non nommées sont soit entières pour affecter une valeur à un entier, soit flottantes pour les flottants, soit chaînes de caractères pour les strings.

Constantes non nommées entières

Ces constantes sont une suite de chiffres décimaux ou hexadécimaux (ou octaux mais plus rarement).

Constantes décimales Les constantes décimales sont évidemment exprimées dans la base de 10 (c'est-à-dire avec des chiffres compris entre 0 et 9).

1
45
1256489
895

Constantes hexadécimales Les constantes hexadécimales sont exprimées en base 16, c'est-à-dire avec les digits 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f. Les six dernières lettres (majuscules et minuscules sont ici équivalentes) correspondent aux valeurs décimales 10 à 15. Pour indiquer au compilateur que la valeur est hexadécimale on fait précéder le nombre du symbole 0x ou 0X. Voici les chiffres de l'exemple sur les constantes décimales en hexadécimal :

```
0x1
0x2D
0x132C29
0x37F
```

Constantes négatives Pour rendre une constante décimale ou hexadécimale négative, il suffit de la précéder du signe moins (-).

```
-1
-0x132C29
-01577
```

Définir les constantes entières Le type d'une constante dépend de sa valeur. Nous avons vu que les valeurs représentables par les types dépendent de la place occupée en mémoire. Une constante de type unsigned short ne pourra pas contenir des nombres plus grands que 2^{16} . Ainsi, si une constante a une valeur trop importante pour un unsigned short, elle se verra attribuer le type int, si celui-ci est encore trop petit ce sera unsigned int, puis long, jusqu'à unsigned long. Cette règle s'applique quelle que soit la base du nombre. Il est possible pour le programmeur d'indiquer lui-même le type d'une constante dans un programme, en terminant la valeur par une lettre. Par exemple :

```
22L
```

force le compilateur à donner le type long à cette constante grâce à la lettre L la terminant, alors qu'elle tenait fort bien dans un short. Une constante littérale suit les règles suivantes :

- S'il n'y a pas de suffixe dans la constante, alors le premier de ces types sera pris dans l'ordre de grandeur de la valeur : int, unsigned int, long, unsigned long.
- Si le suffixe est la lettre L ou l alors le premier de ces types sera pris dans l'ordre de grandeur de la valeur : long, unsigned long

Constantes non nommées réelles

Les constantes à virgules flottantes se composent d'une partie entière (c'est-à-dire les chiffres situés avant le point décimal), d'une partie décimale (soit les chiffres après le point) et éventuellement d'un exposant. Il se trouve toujours exprimé en base 10. **Représentation** La partie entière et les décimales dans une constante réelle peuvent être omises. La présence du point décimal est elle aussi facultative. L'exposant doit s'exprimer en base 10 avec l'aide de la lettre E ou e suivie d'un nombre entier (positif ou négatif). Par exemple :

```
45.12E-1
```

et

```
45.12e-1
```

représentent le nombre 4.512 c'est-à-dire :

```
45.12*10^-1
```

Nous aurions aussi pu écrire ce nombre ainsi :

```
0.4512e1
```

Lorsque la partie entière est nulle comme ici elle peut être omise :

```
.4512e1
```

Si c'est la partie décimale qui est nulle, nous pouvons écrire :

```
4512.e-3
```

qui est équivalent à

4512E-3

Les réels négatifs s'obtiennent tout simplement en rajoutant le préfixe moins (-).

Définir les constantes réelles La gestion des types suivant la valeur de la constante réelle est identique aux constantes entières : Si aucun suffixe n'est spécifié, le type de la constante est double sinon le type est choisi par le compilateur en suivant les règles suivantes :

- Un réel suffixé par un F ou un f est de type float par exemple : 1f, 1.5F ou 456E-3F sont de type float.
- Un réel suffixé par L ou l est de type long double par exemple : 1l, 1.5L ou 1e10l sont tous de type long double.

Constantes caractères

Une constante caractère se compose d'un caractère placé entre une paire d'apostrophes. Exemple :

```
'A'      //caractère majuscule A
'1'      //caractère 1 (à ne pas confondre avec la valeur numérique)
```

Les constantes de caractères sont de type char et pourraient se classer dans les constantes entières comme nous l'avons vu lors de l'étude du type char. Ces constantes représentent les codes entiers du jeu de caractères ASCII. Pour l'ordinateur la constante 'A' équivaut à la valeur numérique 65 et '1' à la valeur numérique 49 (et non 1 comme on pourrait le penser). A chaque nombre de 0 à 255 correspond un caractère imprimable par l'ordinateur.



Astuce

Vous trouverez dans la rubrique langage les 255 premiers caractères ASCII et leur valeur numérique.

Il faut bien comprendre que les caractères '1', '2' ou '3' ne sont pas équivalents aux entiers 1, 2 et 3. Les premiers correspondent aux nombres 49, 50 et 51 et les seconds aux valeurs 1 à 3. Une question peut nous venir à l'esprit : comment représenter le caractère apostrophe puisque celui-ci sert déjà à délimiter les caractères ? La réponse à cette question est "avec les séquences d'échappement".

Les séquences d'échappement

Certains caractères ne peuvent pas être représentés sous une forme imprimable. L'apostrophe est un bon exemple. Pour le compilateur elle représente un séparateur.

Pour représenter l'apostrophe, il faut la faire précéder d'une barre oblique inversée (\) ou backslash :

```

'\''

```

Les caractères introduits par un backslash s'appellent les séquences d'échappement.

Caractère	Séquence d'échappement	nom	code ASCII hexadécimal
'	\'	Single quote	0x0027
"	\"	Double quote	0x0022
\	\\	Backslash	0x005C
caractère nul(NUL)	\0	Null	0x0000
signal sonore(Bel)	\a	Alert	0x0007
retour arrière	\b	Backspace	0x0008
saut de page	\f	Form feed	0x000C
saut de ligne	\n	New line	0x000A
retour chariot	\r	Carriage return	0x000D
tabulation horizontale	\t	Horizontal tab	0x0009
tabulation verticale	\v	Vertical tab	0x000B
nombre hexadécimal	\xhhh	--	hhh prend la valeur hexadécimale voulue

Le guillemet n'a pas besoin d'un backslash pour être représenté en simple caractère

```

" "

```

Par contre à l'intérieur d'une constante string (chaîne de caractères) la séquence d'échappement est obligatoire

```

printf("et Cesar dit : \" les dés sont jetés\" ");

```

A l'inverse, si le caractère apostrophe nécessite obligatoirement une barre oblique inversée en tant que caractère, à l'intérieur d'une string il peut être affiché directement.

```

printf("l'alibi d'la donzelle l'a protégé");

```

Il existe un grand nombre de caractères non imprimables mais très souvent utilisés par les programmes C. Ils ont le code décimal 0 à 31 dans le code ASCII ; le signal sonore (\a) permet d'émettre un Bip, le caractère Backspace (\b) permet de faire reculer d'un caractère la position du curseur, etc... Ainsi l'instruction :

```
printf("le cours sur le langage D\bC est bien ");
```

affichera :

```
le cours sur le langage C est bien
```

La tabulation et le saut de ligne permettent de modifier la présentation de ce que nous écrivons à l'écran

```
printf("1ere ligne\nligne\n3emeline vive le C");
```

La séquence d'échappement \f permet d'effectuer des sauts de pages pour l'impression de données. Par exemple copiez ce code dans un fichier que vous nommerez texte.c

```
//texte.c
/*****
/*
/*          affiche 3 pages          */
/*
/*
/*****

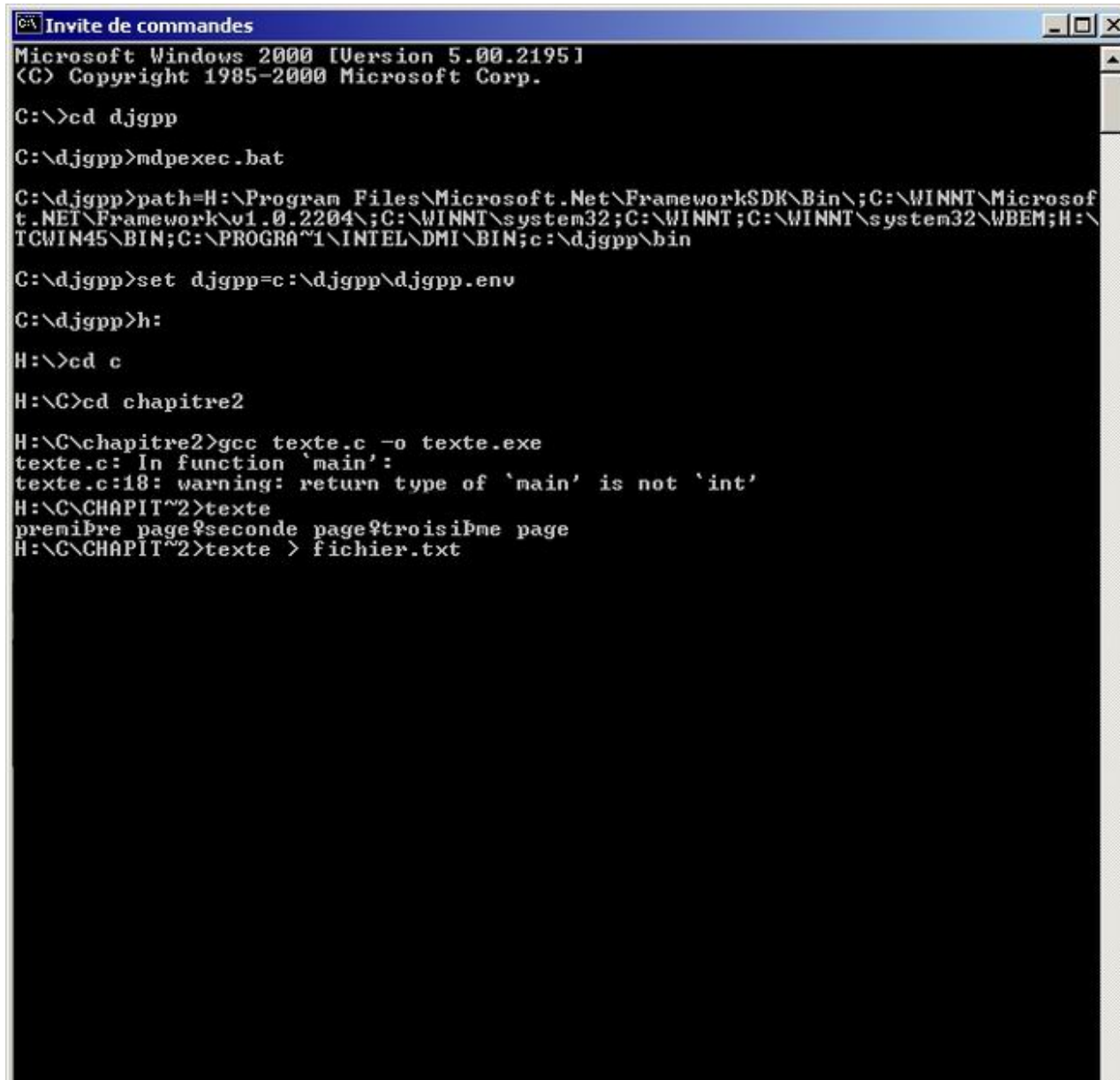
//inclut la bibliothèque
#include <stdio.h>

/**
 * fonction principale du programme
 */
void main()
{
    printf("première page");
    printf("\f");
    printf("seconde page");
    printf("\f");
    printf("troisième page");
```



```
}
```

Si vous l'exécutez, vous obtiendrez une sortie avec des caractères incompréhensibles. En redirigeant (par l'intermédiaire du caractère supérieur) la sortie vers un fichier, vous obtiendrez un document de trois feuilles avec en haut de chacune d'elles écrit "première page", "seconde page" et "troisième page".



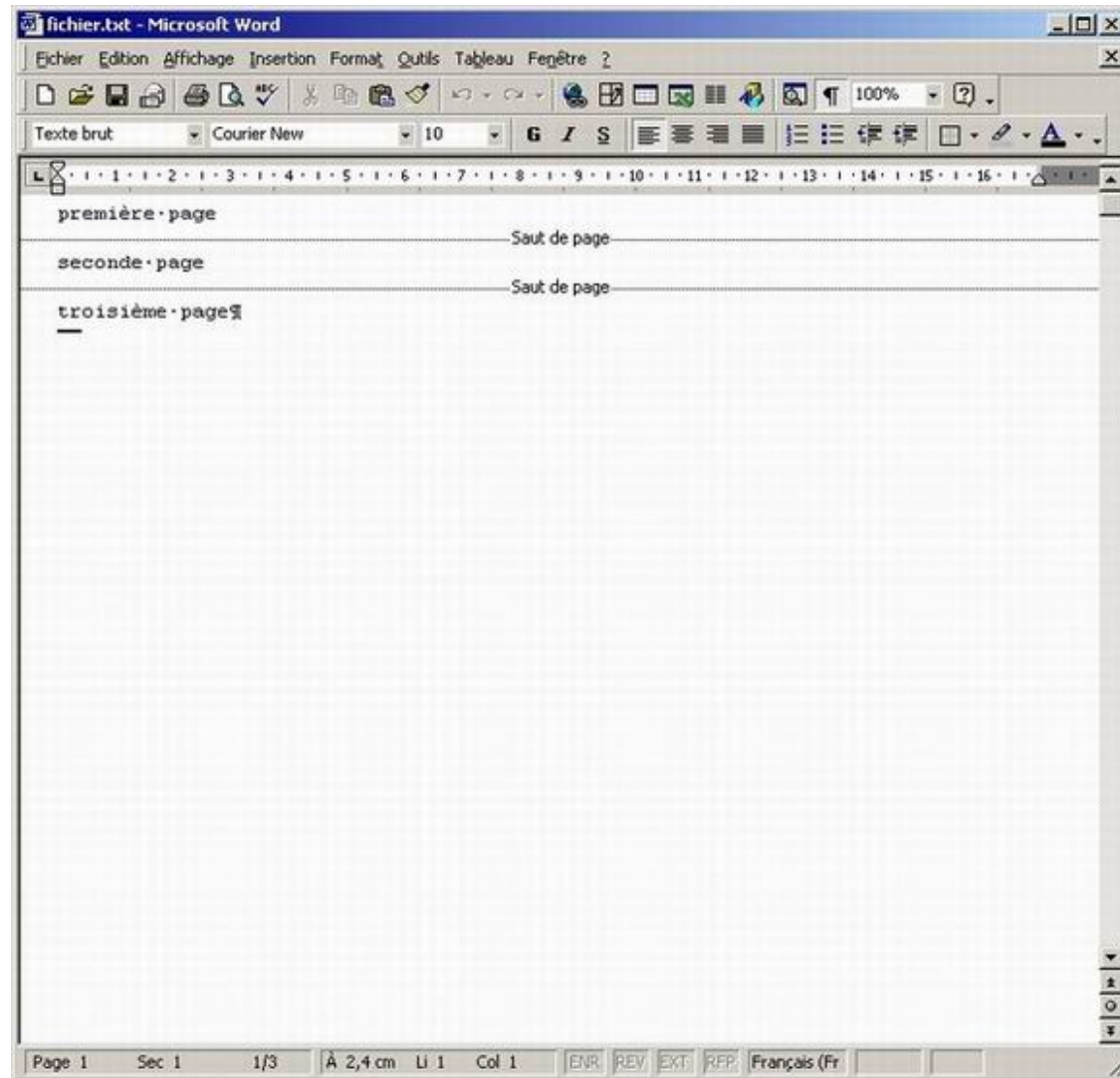
```

C:\>cd djgpp
C:\djgpp>mdpexec.bat
C:\djgpp>path=H:\Program Files\Microsoft.Net\FrameworkSDK\Bin\;C:\WINNT\Microsof
t.NET\Framework\v1.0.2204\;C:\WINNT\system32;C:\WINNT;C:\WINNT\system32\WBEM;H:\
TCWIN45\BIN;C:\PROGRA~1\INTEL\DMI\BIN;c:\djgpp\bin
C:\djgpp>set djgpp=c:\djgpp\djgpp.env
C:\djgpp>h:
H:\>cd c
H:\C>cd chapitre2
H:\C\chapitre2>gcc texte.c -o texte.exe
texte.c: In function 'main':
texte.c:18: warning: return type of 'main' is not 'int'
H:\C\CHAPIT~2>texte
premiere page%seconde page%troisieme page
H:\C\CHAPIT~2>texte > fichier.txt
  
```



Un affichage incompréhensible....

En effet si nous ouvrons le fichier fichier.txt créé comme dans l'image ci-dessus avec un traitement de texte comme ABIWORD sous QNX, KWord sous Linux ou bien Word sous Windows nous obtenons la sortie :



Un résultat tout à fait convenable.

Terminons notre étude des caractères d'échappement en étudiant ceux qui comportent un nombre hexadécimal. Tous les caractères ASCII peuvent être représentés avec leur code hexadécimal. Les séquences d'échappement hexadécimales sont constituées d'une barre oblique inversée suivie d'une valeur en base 16. Le caractère # est codé en hexadécimal par la valeur 23. Ainsi l'instruction :

```
printf("langage C\x23");
```

donnera à la sortie :

```
langage C#
```

Constantes string (chaînes de caractères)

Les constantes chaînes de caractères permettent l'affichage de texte. Elle sont constituées d'une suite de caractères placés entre guillemets. Notre premier programme contenait ainsi la string :

```
"Bonjour de la part de C"
```

Comme nous l'avons vu précédemment les string peuvent contenir des caractères ordinaires mais aussi des séquences d'échappement.

Séquences d'échappement hexadécimales dans les string Utiliser les séquences d'échappement à l'intérieur des chaînes de caractères peut s'avérer dangereux. Imaginons que nous voulions utiliser le point d'exclamation en l'écrivant avec son code hexadécimal :

```
\x21
```

Employée dans un printf seul, cette séquence ne posera pas de problème :

```
printf("\x21");
```

Et donnera la sortie :

```
!
```

Mais si nous voulions afficher la sortie "!exclamation" une erreur risque de se produire :

```
printf( "\x21exclamation");
```

donne la sortie

```
?xclamation
```

Ce qui n'est pas ce que nous attendions. En réalité le compilateur n'a pas lu la séquence \x21 mais \x21e qui représente un caractère tout autre sur le point d'exclamation en ASCII. Ceci est souvent pour le programmeur une source d'erreurs incompréhensible et pourtant facilement débuggable...

Constantes nommées

Intéressons-nous dès maintenant aux constantes nommées. Celles-ci sont des variables auxquelles il est impossible de changer une valeur. Contrairement aux constantes non nommées, les constantes de ce type disposent d'un identificateur auquel est associé une valeur dans le code source. Elles nécessitent une déclaration dans votre code source.

La déclaration d'une constante Une déclaration de constante s'avère pratiquement identique à une déclaration de handle ou de variable (comme nous le verrons plus loin) :

```
const type_de_la_constant nom_de_la_constant = valeur_de_la_constant  
[ nom_de_la_constant = valeur_de_la_constant, .];
```

Les crochets indiquent le facultatif : une instruction peut définir plusieurs constantes. Par exemple, l'instruction :

```
const int jours = 365;
```

définit une constante de type int appelée jours qui a pour valeur 365. L'instruction :

```
const short age= 23, taille = 185;
```

permet la création, en une instruction, de deux constantes de type short nommées age et taille, en leur donnant respectivement la valeur 23 et 185. Notons que si, dans les deux derniers exemples, nous omettions le mot clé const, nous déclarerions alors une variable. Nous avons dit qu'une constante gardait sa valeur à vie, contrairement à une variable. Il est donc nécessaire de lui donner une valeur au moment de sa déclaration et de ne jamais tenter de modifier cette valeur par la suite. Ainsi :

```
const short x;  
  
x = 3;
```

Provoquera une erreur ou un warning au moment de la compilation :

```
texte.c:21: warning: assignment of read-only variable `x'
```

Où déclarer ?

Une déclaration de constante, tout comme une déclaration de variable peut se faire n'importe où à l'extérieur d'une fonction et uniquement au début de fonction :

```
//constante.c  
/*****  
/*  
/*    troisième programme d'apprentissage du C    */  
/*  
/*  
/*  
/*****  
  
//inclut la bibliothèque  
#include <stdio.h>  
  
/**  
 * fonction principale du programme  
 */  
void main()  
{  
  
    const char lettre = 'a';  
    const short x = 3;  
    const float PI = 3.14F;  
    const double EURO = 6.75566;  
    const unsigned int NOMBRE_ALBUMS = 1750000;  
  
    printf("bonjour à tous");  
  
}
```

par contre le code suivant :

```
//constante.c
/*****
/*
/*      troisième programme d'apprentissage du C      */
/*      (seconde version)                             */
/*
/*
*****/

//inclut la bibliothèque
#include <stdio.h>

/**
 * fonction principale du programme
 */
void main()
{
    printf("bonjour à tous");

    const char lettre = 'a';
    const short x = 3;
    const float PI = 3.14F;
    const double EURO = 6.75566;
    const unsigned int NOMBRE_ALBUMS = 1750000;

}
```

est faux car nous avons mis une instruction avant les déclarations de constantes. Les constantes nommées permettent une sécurité du code en interdisant le programmeur d'en modifier la valeur par inadvertance, et en rendant le code source un peu plus lisible grâce au remplacement des valeurs numériques par des noms explicites.



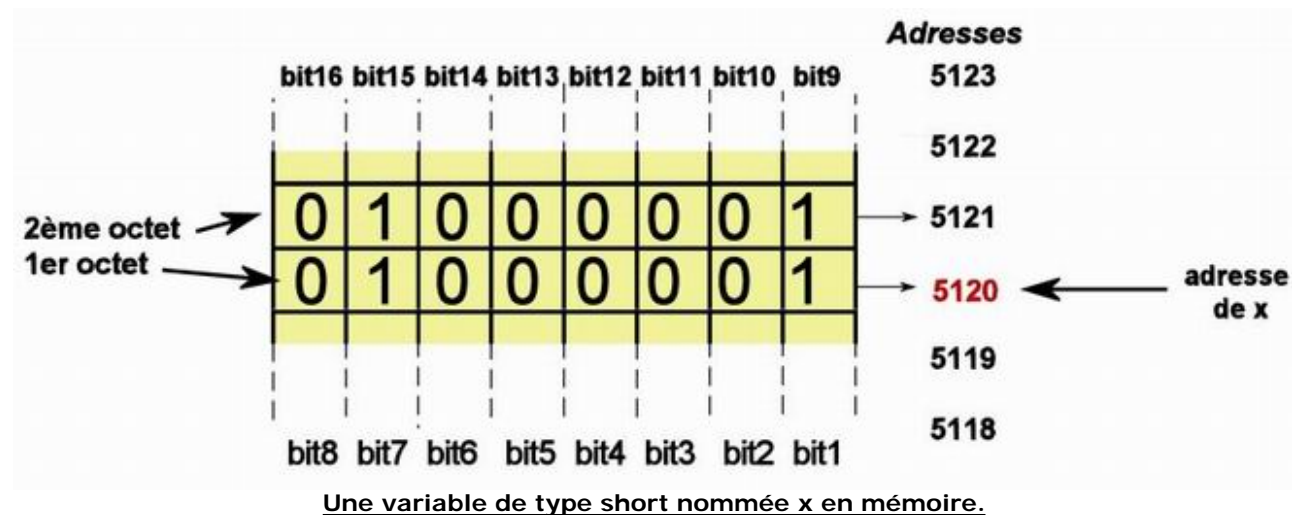
Remarque

Nous apprendrons à afficher ces variables et constantes dans le prochain chapitre.

Lorsque la constante primitive est déclarée à l'extérieur d'une méthode, c'est une constante globale ou une variable globale.

Les variables

Si les constantes sont un atout pour rendre un code clair en créant des données de références dont la valeur ne peut changer, elles souffrent d'un grave défaut : elles sont constantes ! Heureusement le programmeur peut aussi faire appel à des variables qui, elles, peuvent contenir au cours de leur vie un grand nombre de valeurs (qui seront toujours du type de la variable). Une variable est un emplacement mémoire d'un ou de plusieurs octets (suivant le type) qui fixe la manière dont ces octets sont à interpréter. Ainsi pour un emplacement de quatre octets, c'est en connaissant le type de la variable que nous saurons s'il s'agit d'un int ou d'un float ou d'un long. Pour que la variable soit localisable en mémoire centrale, on lui donne une adresse. Cette adresse nous importe peu pour l'instant car avec le nom de la variable nous allons pouvoir la manipuler dans nos programmes.



Ici notre variable a pour adresse 5423. L'emplacement d'une variable est totalement imprévisible, il dépend de l'état de la mémoire au moment de l'exécution du programme. La déclaration d'une variable est identique à celle d'une constante nommée à laquelle on enlève le mot clé const :

```
type_de_la_constant nom_de_la_constant [= valeur_de_la_constant]
[ nom_de_la_constant = valeur_de_la_constant, .];
```

Comme on le voit ici, une variable ne doit pas nécessairement être initialisée lors de sa déclaration. Ainsi l'instruction :

```
long population;
```

déclare une variable de type long nommée population. L'instruction :

```
int resultat1, resultat2, resultat3;
```

Déclare trois variables nommées respectivement resultat1, resultat2 et resultat3 de type int. Sachant qu'un int occupe quatre octets en mémoire, nous réservons donc $4 * 3$ octets soit 12 octets.

Initialisation

En l'absence d'initialisation d'une variable, la valeur prise par celle-ci est tout à fait aléatoire et dépend de ce qu'il y avait en mémoire là où le programme a placé une variable.



Danger

IL EST DONC IMPORTANT DE NE JAMAIS UTILISER UNE VARIABLE NON INITIALISEE !!!

L'initialisation se fait ici aussi à l'aide du signe égal lors de la création de la variable :

```
long population = 3;
```

En initialisant la variable population, comme ci-dessus, la compilation se déroulera sans aucun problème. Dans le cas de plusieurs déclarations dans une même instruction nous ferons :

```
int resultat1 = 15, resultat2 = 12, resultat3 = 6;
```

Affectation

La valeur d'une variable peut être modifiée après sa déclaration. Qu'il y ait eu ou non initialisation. On utilise ici aussi l'opérateur égal = pour affecter à la variable une nouvelle valeur.

```
unsigned short poids;
```

```
//...  
poids = 423;
```

Dans le code ci-dessus, après la déclaration de la variable, on affecte la valeur 423 à la variable poids. Si poids possédait déjà une valeur, elle aurait été remplacée par la nouvelle. Nous aurions pu évidemment écrire :


```
unsigned short poids = 423 ;
```

Mais il s'agirait ici d'une initialisation et non d'une affectation. L'initialisation d'une variable se fait au moment de sa déclaration, alors que l'affectation se fait à n'importe quel moment et dans un nombre indéfini de fois. L'opérande située à droite de l'opérateur = n'est pas forcément une constante non nommée. Avec les déclarations suivantes :

```
int a = 1, b = 7;  
const int c = 12;
```

il est possible de faire :

```
b = a;
```

mais aussi

```
a = c;
```

mais pas :

```
c = b;
```

car c est une constante ! Nous transférons tout d'abord la valeur contenue par a dans b. Cette dernière est maintenant égale à la valeur de a (soit 1). Dans la deuxième instruction, nous transférons la valeur de la constante nommée c dans a. La variable a prend la valeur 12.

Portée des identificateurs de variables ou de constantes

Pour faire référence à un objet, une variable ou une constante nommée, nous utilisons un identificateur. Cet identificateur va nous permettre de manipuler une variable ou une constante. Ces identificateurs n'ont pas de durée proprement dite mais une visibilité. Les primitives ne sont visibles que dans certaines parties de notre programme et invisibles dans d'autres. On appelle portée (ou scope en anglais) l'étendue du programme dans lequel un identificateur existe. Nous savons que la majeure partie d'un programme se déroule à l'intérieur de blocs (délimités par une accolade ouvrante { et une accolade fermante }). La portée d'un identificateur s'étend de l'endroit où il est créé, jusqu'à la fin du bloc (en incluant les blocs imbriqués). L'image ci-dessous nous montre la portée des identificateurs dans un programme.

void main()

```
{
  int a;
  ...
  {
    int b;
    ...
  }
  int c;
  ...
  {
    int d;
    ...
    {
      int e;
      ...
    }
  }
}
```

Portée des différents identificateurs de la fonction main ; à chaque zone de gris correspond une

portée.

Voyons cela concrètement en compilant le programme suivant :

```
//texte.c

//inclut la bibliothèque
#include <stdio.h>

/**
 * fonction principale du programme
 */
void main()
{
    int x = 3;

    {
        int b = 5;

    }

    b = x;

}
```

La compilation échoue :

```
H:\C\chapitre2>gcc portee.c -o portee.exe
portee.c: In function `main':
portee.c:25: `b' undeclared (first use in this function)
portee.c:25: (Each undeclared identifier is reported only once
portee.c:25: for each function it appears in.)
portee.c:18: warning: return type of `main' is not `int'
```

En effet l'instruction :

```
b = x;
```

utilise la variable b alors que celle-ci n'est pas visible (elle a été déclarée dans un bloc de niveau supérieur). De même, le programme suivant :

```
//portee.c

//inclut la bibliothèque
#include <stdio.h>

/**
 * fonction principale du programme
 */
void main()
{

    int b = x;
    int x = 3;

}
```

ne peut se compiler car l'instruction :

```
int b = x;
```

utilise la variable x alors que celle-ci n'a pas été déclarée.

Annexe

Les types du langage C

Par *Lord Asriel*



Copyright ©1998-2003



| Toute reproduction interdite | Contact: Webmaster@ProgrammationWorld.com | Confidentialité