utilisation de sizeof ?

priorité de l'opérateur modulo

Les nombres entiers

Dans le chapitre précédent, vous avez appris à utiliser `cout` pour afficher des messages. Les messages sont des chaînes de caractères encadrées par des guillemets droits ". Une valeur écrite directement dans le code C++ est appelée une littérale. Il existe d'autres types de littérales :

- les nombres entiers : `0`, `1`, `2`, `-1`, `-2`, etc.
- les nombres réels : `1.0`, `2.1`, `-5.12`, `1.457`, etc.
- les booléens, qui représente une valeur à deux états : `true` (vrai) et `false` (faux).
- les caractères : 'a', 'b', 'c', etc.
- et les chaînes de caractères que vous avez déjà vu : "hello, world", "salut tout le monde", etc.

Vous pouvez afficher ces littérales directement en utilisant `cout`, comme vous l'avez vu pour les chaînes de caractères.

Écrire des nombres entiers

Les nombres entiers ne devraient pas vous poser de problème, ce sont les nombres que vous utiliser pour compter depuis l'école maternelle. La forme la plus simple pour écrire un nombre entier est d'écrire une série continue de chiffres entre 0 et 9. Les nombres ne doivent pas commencer par zéro.

main.cpp

```
#include <iostream>

int main() {
```

```
    std::cout << 1 << std::endl; // affiche le nombre 1
    std::cout << 2 << std::endl; // affiche le nombre 2
    std::cout << 3 << std::endl; // affiche le nombre 3
    std::cout << 4 << std::endl; // affiche le nombre 4
}
```

Pour écrire un nombre entier négatif, vous devez ajouter le signe moins devant le nombre.

main.cpp

```
#include <iostream>

int main() {
    std::cout << -1 << std::endl; // affiche le nombre -1
    std::cout << -2 << std::endl; // affiche le nombre -2
    std::cout << -3 << std::endl; // affiche le nombre -3
    std::cout << -4 << std::endl; // affiche le nombre -4
}
```

Lorsque vous écrivez de très grands nombres, il est difficile de le lire. Par exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 123456789123456789 << std::endl;
}
```

La raison est que le cerveau humain est capable de reconnaître des groupes de quelques caractères, mais pas un seul bloc de 18 caractères. Il n'arrive donc pas, en un coup d'oeil, à identifier si ce nombre est de l'ordre du million, du milliard ou autre.

Pour faciliter la lecture, il est donc possible d'ajouter un caractère guillemet droit ' pour séparer un grand nombre en plusieurs groupes. Par habitude, on séparera en groupes de trois chiffres :

main.cpp

```
#include <iostream>
```

```
int main() {
    std::cout << 123'456'789'123'456'789 << std::endl;
}
```

Vous pouvez écrire des nombres très grand de cette manière, mais il existe une limite. Si vous écrivez un nombre trop grand, vous aurez un message d'erreur signalant que ce nombre est trop grand.

main.cpp

```
#include <iostream>

int main() {
    std::cout << 123456789123456789 << std::endl; // affiche
le nombre 123456789123456789
}
```

Produira l'erreur :

```
main.cpp:4:18: error: integer constant is larger than the
largest unsigned integer type
    std::cout << 123456789123456789123456789123456789 <<
std::endl; // affiche le nombre 123456789123456789
                 ^
1 error generated.
```

L'existence d'une valeur maximale limite est liée à la représentation des nombres dans la mémoire des ordinateurs et à la notion de type de données. Vous verrez cela dans les prochains chapitres.

Il est bien sûr possible d'utiliser des nombres avec autant de chiffre que vous souhaitez, mais il ne sera pas possible d'utiliser dans ce cas les nombres entiers tel que définit dans ce chapitre. Vous apprendrez à créer dans un travail pratique dans un prochain chapitre comment créer ce type de nombres entiers.

Les nombres décimaux, hexadécimaux, octaux et binaires

Les nombres entiers que vous utilisez habituellement s'écrivent à partir de dix chiffres (0 à 9). C'est pour cette raison que l'on parle de système décimal (du latin “decimus”, qui signifie “dixième”) et l'on parle de base 10. Mais ce n'est pas la seule façon d'écrire les nombres et il existe d'autres systèmes numériques.

Imaginons par exemple que l'on souhaite écrire les nombres en utilisant que huit chiffres (0 à 7). Dans ce cas, nous pouvons compter de la façon suivante : 0, 1, 2, 3, 4, 5, 6, 7. Arrivé au huitième chiffre, nous ne pouvons pas écrire “8”, puisque ce chiffre n'est pas autorisé dans ce système décimal. Donc, il faut passer à un nombre à deux chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, etc.

Vous verrez dans les exercices comment convertir un nombre écrit selon une base dans une autre base.

En C++, il est possible d'écrire et d'afficher des nombres écrit selon des bases différentes de 10. Pour des raisons historiques, les ordinateurs sont habitués à manipuler les nombres en base 2 (binaire), 8 (octal), 16 (hexadécimal). Pour écrire un nombre en base différente de 10, il faut commencer le nombre par le chiffre 0 puis un caractère pour spécifier la base : rien pour octal, x ou X pour l'hexadécimal et b pour le binaire. Les chiffres autorisés pour écrire un nombre dépendent également du mode : 0 à 7 pour l'octal, 0 à 9 et a à f (ou A à F) pour l'hexadécimal et 0 et 1 pour le binaire.

Le code suivant permet d'afficher la valeur de 10 selon la base :

main.cpp

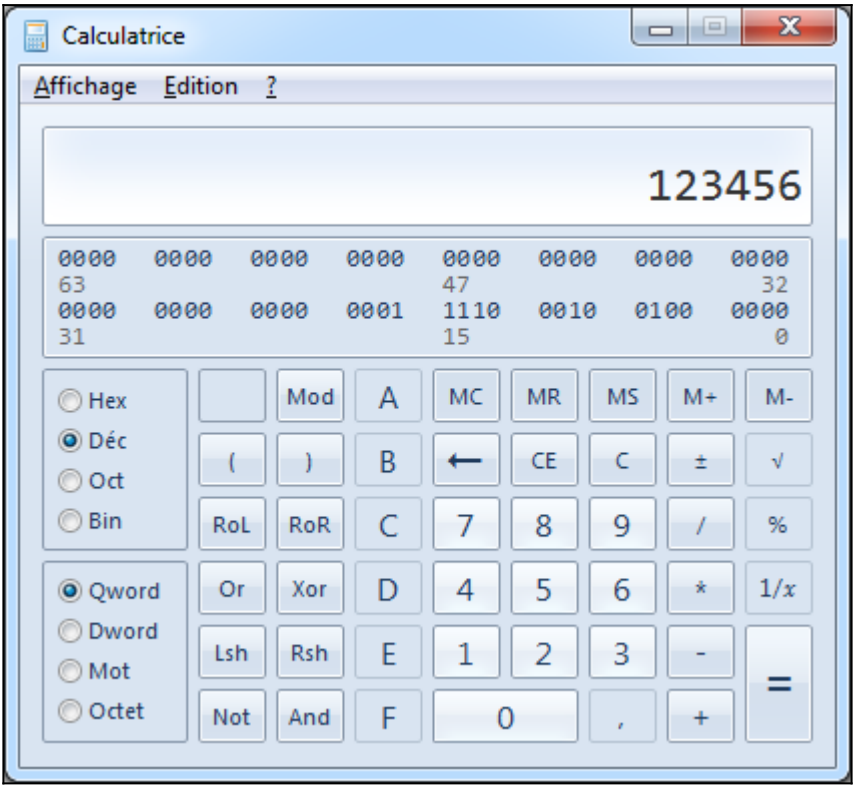
```
#include <iostream>

int main() {
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
```

```
}
```

affiche :

```
10
8
16
2
```

Il est possible calculer les conversions à la main, mais c'est plus simple d'utiliser un convertisseur. Il existe des convertisseurs en ligne, vous pouvez également utiliser la calculatrice de Windows en sélectionnant le mode “Programmeur” dans le menu “Affichage”.



Il faut donc bien faire attention à la base utilisée lorsque l'on écrit un

nombre.

Base	Système	Préfixe	Chiffres	Exemple
2	binaire	0b	0 et 1	`0b01`
8	octal	0	0 à 7	`001234567`
10	décimal		0 à 9	`1234567890`
16	hexadécimal	0x	0 à 9 et a à f	`0x0123456789abcdef`
		0X	0 à 9 et A à F	`0X0123456789ABCDEF`

suffixe ? u l et ll

Comme vous l'avez vu dans les codes précédents, les nombres sont affichés après conversion en base 10. C'est le mode de fonctionnement par défaut de `cout`, mais il est possible de modifier ce comportement. Les directives suivantes permettent de modifier la base utilisée par `cout` pour afficher les nombres :

- `std::oct` pour afficher en utilisant la base 8 ;
- `std::dec` pour afficher en utilisant la base 10 ;
- `std::hex` pour afficher en utilisant la base 16.

Il n'existe pas de directive permettant d'afficher les nombres en binaire, vous verrez dans un exercice dans un prochain chapitre comment afficher des nombres en binaire. Par défaut, `cout` affiche les nombres en utilisant la base 10.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::hex;
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
}
```

affiche :

```
a
8
10
2
```

Comme vous le voyez, une directive continue de s'appliquer sur toutes les lignes suivantes, tant que vous ne donner pas une directive modifiant une nouvelle fois l'affichage. Vous pouvez écrire les directives sur une ligne différente (comme dans le code précédent), sur la première ligne ou à chaque ligne, cela ne change rien au résultat.

Si vous manipuler et afficher plusieurs bases, il peut être difficile de savoir exactement ce que vous afficher, avec quelle base. Il est dans ce cas possible d'afficher la base utilisée pour l'affichage avec la directive `std::showbase`. Pour ne plus afficher la base, vous pouvez utiliser la directive `std::noshowbase`.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;

    std::cout << std::hex << std::showbase;
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
}
```

affiche :

```
10
8
16
2
0xa
```

```
0x8
0x10
0x2
```

Les opérations arithmétiques sur les entiers

Pouvoir écrire et afficher des nombres entiers est un début, mais il est possible d'aller plus loin et de réaliser des calculs dessus. Pour cela, le C++ propose plusieurs opérateurs naturels de l'arithmétique, comme l'addition `+`, la soustraction `-` ou la multiplication `*`.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "123 + 456 = " << 123 + 456 << std::endl;
    std::cout << "123 - 456 = " << 123 - 456 << std::endl;
    std::cout << "123 * 456 = " << 123 * 456 << std::endl;
}
```

affiche :

```
123 + 456 = 579
123 - 456 = -333
123 * 456 = 56088
```

Il est également possible de faire des divisions entières avec l'opérateur division `/` et de calculer le reste d'une division entière avec l'opérateur modulo `%`. Pour rappel, la division entière permet de calculer le résultat d'une division en utilisant uniquement des nombres entiers (c'est le premier type de division que vous avez appris à l'école). Ainsi :

- avec une division entière : 11 divisé par 4 donne 2 et reste 3 ;
- avec une division réelle : 11 divisé par 4 donne 2,75.

main.cpp

```cpp
#include <iostream>

int main() {
```

```
    std::cout << "10 / 2 = " << 10 / 2 << std::endl;
    std::cout << "10 / 3 = " << 10 / 3 << std::endl;
    std::cout << "10 / 4 = " << 10 / 4 << std::endl;

    std::cout << "10 % 2 = " << 10 % 2 << std::endl;
    std::cout << "10 % 3 = " << 10 % 3 << std::endl;
    std::cout << "10 % 4 = " << 10 % 4 << std::endl;
}
```

affiche :

```
10 / 2 = 5
10 / 3 = 3
10 / 4 = 2
10 % 2 = 0
10 % 3 = 1
10 % 4 = 2
```

Les opérations mathématiques suivent la sémantique habituelle, en particulier pour l'ordre d'évaluation et l'utilisation des parenthèses. Pour rappel, pour évaluer une expression mathématique (une série de plusieurs opérations), l'ordre d'évaluation est le suivant :

- en premier, évaluer les expressions entre parenthèse ;
- ensuite, évaluer les multiplications et les divisions ;
- pour terminer, évaluer les additions et les soustractions.

Ainsi, l'expression ci-dessous s'évalue de la façon suivante :

```
  5 - (2 + 3) / 4
= 5 -    5    / 4
= 5 -         1
=   4
```

Écrivez le code C++ correspondant pour évaluer cette expression et vérifier que le résultat est correct.

De la même façon, les opérateurs du C++ suivent les règles d'arithmétiques habituelles :

- la commutativité (sauf pour la division) : a + b = b + a ;

- l'associativité : a + (b + c) = (a + b) + c ;
- la distributivité : a * (b + c) = a * b + a * c.

On peut vérifier facilement ces règles avec un programme C++ :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Commutativité :" << std::endl;
    std::cout << "2 + 3 = " << 2 + 3 << std::endl;
    std::cout << "3 + 2 = " << 3 + 2 << std::endl;

    std::cout << "Associativité :" << std::endl;
    std::cout << "2 + (3 + 4) = " << 2 + (3 + 4) << std::
endl;
    std::cout << "(2 + 3) + 4 = " << (2 + 3) + 4 << std::
endl;

    std::cout << "Distributivité :" << std::endl;
    std::cout << "2 * (4 + 3) = " << 2 * (4 + 3) << std::
endl;
    std::cout << "2 * 4 + 2 * 3 = " << 2 * 4 + 2 * 3 << std
::endl;
}
```

affiche :

```
Commutativité :
2 + 3 = 5
3 + 2 = 5
Associativité :
2 + (3 + 4) = 9
(2 + 3) + 4 = 9
Distributivité :
2 * (4 + 3) = 14
2 * 4 + 2 * 3 = 14
```

Un dernier point pour terminer : essayez de diviser un nombre entier par zéro :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "1 / 0 = " << (1 / 0) << std::endl;
}
```

affichera un avertissement à la compilation (*warning*) vous informant que la division par zéro n'est pas définie :

```
main.cpp:4:20: warning: division by zero is undefined
[-Wdivision-by-zero]
    std::cout << (1/0) << std::endl;
                   ^~
1 warning generated.
```

On parle en C++ de “comportement indéfini”, *undefined behavior* en anglais, que l'on abrège parfois avec les initiales “UB”. Cela signifie qu'un programme utilisant un tel code aura un comportement aléatoire et faire n'importe quoi : provoquer une erreur, donner un résultat quelconque ou encore sembler fonctionner correctement. Ce type d'erreur est difficile à diagnostiquer dans un programme complexe puisque cela ne produit pas forcement un message clair sur la ligne de code correspondant au problème.

Ainsi, l'exécution du programme précédent se déroule sans message d'erreur. Par contre, la valeur affichée par `cout` est aléatoire et n'a pas de sens en termes mathématique :

```
1 / 0 = 4196864
```

Le C++ étant un langage permissif, il n'y a aucune vérification faite par le langage pour éviter ce type d'erreur dans le code. C'est de votre responsabilité de vérifier que le code ne provoque pas de comportement indéfini. Pour cela, voici quelques aides :

Respectez les bonnes pratiques de codage. Comme le C++ est permissif, le développeur a le droit d'écrire du code correct ou du code incorrect. Cela peut paraître évident, mais il vaut mieux le dire : il est

préférable d'écrire du code correct que du code incorrect ! Il y a souvent plusieurs façon en C++ de résoudre un même problème, certaines approches posent plus de problèmes que d'autres.

Ce cours de C++ n'a pas pour vocation à vous apprendre tout le C++. Il se focalise, volontairement, sur les syntaxes les plus récentes et les plus sûres. Et même en utilisant ces approches modernes, il y aura des précautions à prendre pour éviter les problèmes. Vous apprendrez à écrire du code le plus sûr possible.

Faites vous aider par le compilateur. Celui-ci peut faire certaine vérifications, comme dans le code précédent, et vous avertir lorsque vous écrivez un code qui pose manifestement problème. Tous les compilateurs ne font pas forcement les mêmes vérifications et les vérifications faites dépendent des options utilisées. Les options `-Wall -Wextra -pedantic` activent les principales vérifications, mais il en existe d'autres. Il peut être intéressant de compiler un code avec plusieurs compilateurs pour avoir un maximum de messages d'avertissement.

Utilisez les outils de vérifications statique et dynamique. Ils ne permettent pas simplement de vérifier le respect des règles du langage C++, mais également le respect de bonnes pratiques de codage. Dans l'objectif d'écrire des programmes C++ modernes, il ne faut pas hésiter à utiliser ce type d'outils et vous apprendrez dans la suite de cours à les utiliser.

Exercices

Conversion en nombre de n'importe quelle base

http://fr.wikipedia.org/wiki/Syst%C3%A8me_d%C3%A9cimal

opérations à faire pour les calculs ?

Calculer l'indice de colonne et de ligne d'une table

utilisation de / et %

Résumé

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++