syntaxe full compile time, risque de confusion à voir tout de suite les 2 syntaxes aussi en détails

Les nombres rationnels

Rappel sur la représentation des nombres

Les nombres rationnels sont des nombres qui s'écrivent sous la forme d'une division de deux nombres entiers : un numérateur divisé par un dénominateur non nul.

$$ \text{nombre rationnel} = \frac{\text{numérateur}}{\text{dénominateur}} $$

Comme pour les nombres complexes, le but de ce chapitre n'est pas d'entrer dans les détails mathématiques des nombres rationnels, mais de découvrir les outils fournis par le C++ pour les manipuler. Pour les détails, vous pouvez consulter la page de Wikipédia : Nombre rationnel.

L'ensemble des nombres rationnels $ \mathbb Q $ est inclut dans l'ensemble des nombres réels $ \mathbb R $, il est donc classique d'écrire un nombre rationnel sous une forme décimale limitée. Par exemple, le nombre $ \frac{1}{3} $ pourra être écrit : $ 0,33333... $. Cependant, cette représentation est problématique, puisque inexacte (il faudrait écrire une infinité de chiffre 3) : il n'existe de représentation décimale exacte pour les nombres rationnels.

Sur un ordinateur, la situation est encore plus problématique. Pour rappel, les nombres réels sont représentés en mémoire par un nombre fini d'octets, il n'est donc pas possible de représenter tous les nombres réels possibles, mais en général uniquement une valeur approchées (Il est donc tout à fait possible d'avoir deux nombres réels mathématiquement différents, mais qui auront la même représentation en mémoire et seront donc considérés égaux dans un programme C++).

Cela fait donc deux raisons qui limitent l'exactitude des calculs avec des nombres réels sur un ordinateur. Lorsque l'on réalise des calculs scientifiques, il est absolument nécessaire de prendre en compte ces erreurs.

Vous avez vu dans le chapitre précédent sur les nombres à virgule fixe une méthode pour représenter un nombre réels en utilisant un nombre entier et un diviseur fixe. Les nombres rationnels sont une généralisation de cette approche, mais au lieu d'utiliser un diviseur fixé dans le code, le diviseur pourra varier pour chaque nombre rationnel. Il ne faudra donc plus manipuler qu'un seul nombre, mais deux.

Remarque : n'oubliez pas que même si les nombres entiers n'ont pas de problème d'arrondi sur un ordinateur, ils sont quand même limites : ils ont une valeur maximale et une valeur minimale. Si vous utiliser des nombres trop grand ou trop petit, vos calculs seront faux.

La classe std::ratio

Si vous vous souvenez du chapitre sur les nombres complexes, vous avez déjà manipulé en C++ une classe (`std::complex`) qui permet de manipuler deux nombres réels. Vous pouvez donc imaginer que la classe `std::ratio` sera similaire à `std::complex` et que l'étude de cette classe n'apporte pas grand chose.

En fait, ce n'est pas du tout le cas !

Vous avez vu dans le chapitre Programme C++ minimal qu'il y avait deux étapes pour obtenir le résultat d'un programme : une première phase de compilation (*compile-time*), qui permet de générer un programme à partir du code source C++, et une seconde phase d'exécution (*runtime*), qui exécute le programme. Cette distinction est intéressante, puisque la phase de compilation sera généralement réalisée une seule fois, pour ensuite exécuter plusieurs fois le programme. Donc tout ce qui est fait lors de cette première étape sera du temps de gagné sur l'exécution du programme.

Et c'est bien la distinction majeure entre `std::complex` et `std::ratio`.

La première permet de réaliser des calculs lors de l'exécution, alors que la second travaille uniquement lors de la compilation. (Il est possible de créer une classe en C++ permettant de manipuler des nombres rationnels lors de l'exécution, vous ferez cela en exercice. Mais ce n'est pas l'approche utilisée par `std::ratio`).

La conséquence est que la syntaxe pour utiliser `std::ratio` est totalement différente de celle de `std::complex`. En particulier, il n'est pas possible d'utiliser les opérateurs que vous avez déjà utilisé pour les nombres et `std::complex` (comme `+` ou `==`).

Remarque : en règle générale (donc pas que pour std::ratio), quand possible, le compilateur fera les calculs à compile time, en fonction des options de compilation. Par exemple :

```
cout << (2+3) << endl;
```

Le compilateur peut calculer directement l'opération (2+3), il va donc remplacer le code par le résultat et compilera en fait :

```
cout << 5 << endl;
```

C'est très intéressant, puisque tout ce qui est fait au compile time sera fait qu'une seule fois = meilleures performance.

`std::ratio` uniquement en compile time, donc le résultat est évalué à la compilation.

http://en.cppreference.com/w/cpp/numeric/ratio

main.cpp

```
std::ratio<2, 3>

std::ratio_add<std::ratio<2, 3>, std::ratio<3, 4>>

std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio<3,
4>>::num << std::endl;
std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio<3,
4>>::den << std::endl;
```

Utilisation de using. Note : ancienne écriture : typedef

Opération arithmétiques : ratio_add, ratio_subtract, ratio_multiply, ratio_divide

Opérateur comparaison : ratio_equal, ratio_not_equal, ratio_less, ratio_less_equal, ratio_greater, ratio_greater_equal

Ratios communs : micro, milli, kilo, mega, etc.

Chapitre précédent	Sommaire principal	Chapitre suivant