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

[Aller plus loin] Les nombres rationnels

Rappel sur la représentation des nombres rationnels

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.

Créer un nombre rationnel avec 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.

Optimisation à la compilation
En règle générale, le compilateur essayera au mieux d'optimiser les calculs à la 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 (mais vous ne le verrez pas, sauf à aller lire les fichiers binaires créés par le compilateur) :

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

C'est très intéressant, puisque tout ce qui est fait lors de l'étape de compilation ne sera fait qu'une seule fois et l'application sera un peu plus rapide (mais cela ne sera pas forcement perceptible par les utilisateurs. Dans l'exemple avec l'addition qui est remplacé par sa valeur, la différence de performances est de l'ordre de la nanoseconde probablement, donc non perceptible).

La classe `std::ratio` est un cas particulier, puisqu'elle ne peut être utilisée que lors de la compilation. C'est une forme de limitation, puisque cela n'est pas utilisable tout le temps. Mais dans certains cas, il est préférable d'avoir une solution qui est performante et qui ne fonctionne qu'à la compilation ou qu'à l'exécution, plutôt que d'avoir une solution plus généraliste, mais moins performante.

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 `==`).

Notez bien la distinction faite entre **l'opération** (calculer une addition, une soustraction, etc.) et les **opérateurs** (`+`, `-`, etc).
En général, une opération représente un concept, qui est défini de façon unique, souvent par une définition mathématique (comme c'est la cas par exemple ici avec l'addition entre deux nombres rationnels).

Au contraire, un opérateur est un moyen dans le code C++ de réaliser cette opération. Une opération peut être définie pour une classe, sans que l'opérateur habituel correspondant ne soit définie (comme c'est le cas ici avec `std::ratio`, qui permet de calculer une addition, mais sans utiliser l'opérateur `+`). Il est également possible d'avoir plusieurs syntaxes permettant de réaliser une opération (par exemple que l'on puisse utiliser `+` et `std::add` pour calculer une addition).

Il est important que les fonctionnalités proposées par une classe soient cohérentes et quelque soit la méthode utilisée pour calculer une addition par exemple, le résultat soit toujours le même.

La classe `std::ratio` utilise la notation avec chevrons, que vous avez déjà rencontré rapidement. Mais pas d'inquiétude, la syntaxe est simple, il suffit d'écrire : `std::ratio<numérateur, dénominateur>`. Pour écrire la fraction $ \frac{1}{3} $ par exemple, il faudra donc écrire :

```
#include <ratio>

std::ratio<1, 3>
```

Il existe un certain nombre de `std::ratio` prédéfinie dans la norme C++. La liste complète est donnée dans la page de documentation : std::ratio. Ces valeurs correspondent aux préfixes du système international d'unités (voir Wikipédia pour les détails). Par exemple :

- nano = `std::ratio<1, 1000000000>` ;
- micro = `std::ratio<1, 1000000>` ;
- milli = `std::ratio<1, 1000>` ;

- kilo = `std::ratio<1000, 1>` ;
- mega = `std::ratio<1000000, 1>` ;
- giga = `std::ratio<1000000000, 1>`.

Exercices

- Ecrire les fractions : 2/3, 1/2, 3/3.

Afficher un std::ratio

Si vous compilez le code précédent, vous obtiendrez l'avertissement suivant :

```
main.cpp:4:5: warning: declaration does not declare anything
[-Wmissing-declarations]
    std::ratio<1, 3>;
    ^~~~~~~~~~~~~~~~
1 warning generated.
```

Ce message indique en fait que la ligne contenant `std::ratio` ne fait rien. En effet, c'est le cas, le code déclare simplement un `std::ratio`, et ne fait rien avec.

Essayons d'afficher un `std::ratio` avec `std::cout` :

```
#include <iostream>
#include <ratio>

int main() {
    std::cout << std::ratio<1, 3> << std::endl;
}
```

Malheureusement, ce code ne fonctionne pas directement et produit une erreur :

```
main.cpp:5:35: error: expected '(' for function-style cast
or type construction
    std::cout << std::ratio<1, 3> << std::endl;
                 ~~~~~~~~~~~~~~~~ ^
```

```
1 error generated.
```

Le message est un peu plus complexe a comprendre, mais au final, cela veut dire qu'il ne comprend pas cette syntaxe. La raison est en fait très simple : la classe `std::ratio` ne contient pas de fonctionnalités pour être affichée. Il faut récupérer directement les valeurs du numérateur et du dénominateur et les afficher.

Pour cela, il faut utiliser `num` et `den` avec la syntaxe suivante :

```
std::ratio<1, 3>::num // numérateur
std::ratio<1, 3>::den // dénominateur
```

Au final, pour afficher un `std::ratio`, il faut donc écrire :

main.cpp

```
#include <iostream>
#include <ratio>

int main() {
    std::cout << std::ratio<1, 3>::num << std::endl;
    std::cout << std::ratio<1, 3>::den << std::endl;
}
```

affiche :

```
1
3
```

Une fraction est valide pour n'importe quelle paire d'entiers, avec un dénominateur non nul. Si on essaie de créer un `std::ratio` avec un dénominateur nul, on obtient une erreur assez explicite “denominator cannot be zero” :

main.cpp

```
#include <iostream>
#include <ratio>

int main() {
    std::cout << std::ratio<1, 0>::num << std::endl;
```

```
}
```

affiche :

```
In file included from main.cpp:2:
/usr/local/bin/../lib/gcc/x86_64-unknown-linux-gnu/5.2.0/../.
./../../include/c++/5.2.0/ratio:265:7: error:
static_assert failed "denominator cannot be zero"
      static_assert(_Den != 0, "denominator cannot be
zero");
      ^             ~~~~~~~~~
main.cpp:5:23: note: in instantiation of template class
'std::ratio<1, 0>' requested here
    std::cout << std::ratio<1, 0>::num << std::endl;
                      ^
1 error generated.
```

Un nombre rationnel peut être représenté par une infinité de paire d'entiers, en multipliant le numérateur et le dénominateur par un même nombre non nul. Par exemple, toutes les fractions suivantes sont des représentations du même nombre rationnel :

$$ \frac{2}{3} = \frac{4}{6} = \frac{6}{9} = \frac{12}{18} = \frac{22}{33} ... $$

La représentation utilisant un numérateur et un dénominateur qui ne possèdent pas de diviseurs commun est appelée forme standardisée. Par exemple, “4/6” n'est pas une forme standardisée, puisque “4” et “6” ont un diviseur commun (on peut diviser 4 et 6 par 2). Par contre, “2/3” est une forme standardisée. Pour chaque nombre rationnel, il existe une et une seule forme standardisée.

La classe `std::ratio` simplifie les fractions et utilise la forme standardisée. Lorsque vous utiliser `num` et `den`, cela correspond au numérateur et dénominateur de la forme standardisée.

```
#include <iostream>
#include <ratio>

int main() {
```

```
    std::cout << std::ratio<2, 3>::num << " / " <<
std::ratio<2, 3>::den << std::endl;
    std::cout << std::ratio<4, 6>::num << " / " <<
std::ratio<4, 6>::den << std::endl;
}
```

affiche

```
2 / 3
2 / 3
```

Les opérateurs arithmétiques de std:ratio

Les quatre opérations de base sont fournies avec la classe `std::ratio` : addition, soustraction, multiplication et division. Cependant, il n'est pas possible ici d'utiliser les opérateurs habituels `+`, `-`, `*` et `/` (pour des raisons techniques concernant la syntaxe de ces opérateurs, mais ce n'est pas important pour le moment). Ces opérations sont donc implémentées dans des fonctionnalités :

- `ratio_add` pour l'addition ;
- `ratio_subtract` pour la soustraction ;
- `ratio_multiply` pour la multiplication ;
- `ratio_divide` pour la divison.

Ces fonctionnalités utilisent aussi la syntaxe avec chevrons, il faut donc écrire :

```
std::ratio_add<première fraction, second fraction>
```

En remplaçant “première fraction” et “seconde fraction” par des appels à `std::ratio`. Donc, concrètement, si on veut additionner par exemple 2/3 et 3/4, il faudra écrire :

main.cpp

```
#include <iostream>
#include <ratio>
```

```
int main() {
    std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio
<3, 4>>::num << " / ";
    std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio
<3, 4>>::den << std::endl;
}
```

affiche :

```
17 / 12
```

Vous réalisez sans doute l'intérêt d'utiliser les opérateurs habituels `+`, `-`, `*` et `/` avec ce simple code. Même si le résultat est le même, l'utilisation des opérateurs habituels permet d'avoir une syntaxe plus simple et plus lisible.
Pour terminer, la classe `std::ratio` propose également les opérations de comparaison habituels, également avec des noms spécifiques au lieu des opérateurs :

- égalité = `ratio_equal` ;
- différence = `ratio_not_equal` ;
- infériorité = `ratio_less` ;
- infériorité ou égalité = `ratio_less_equal` ;
- supériorité = `ratio_greater` ;
- supériorité ou égalité = `ratio_greater_equal`.

Le résultat booléen est obtenu en utilisant `value` :

main.cpp

```
#include <iostream>
#include <ratio>

int main() {
    std::cout << "2/3 == 3/4 ? " << std::boolalpha;
    std::cout << std::ratio_equal<std::ratio<2, 3>, std::
ratio<3, 4>>::value << std::endl;

    std::cout << "2/3 == 4/6 ? " << std::boolalpha;
```

```
    std::cout << std::ratio_equal<std::ratio<2, 3>, std::
ratio<4, 6>>::value << std::endl;
}
```

affiche :

```
2/3 == 3/4 ? false
2/3 == 4/6 ? true
```

Exercices

- écrivez les opérations suivantes : 2/3+3/4, 2/3-3/4, 2/3*3/4, (2/3) / (3/4).

Chapitre précédent	Sommaire principal	Chapitre suivant