[Aller plus loin] Les nombres complexes

L'une des utilisations majeures du C++ est le calcul numérique intensif. Pour cela, il est possible d'utiliser des fonctionnalités du langage (comme par exemple les nombres à virgule flottante que vous avez vus précédemment) ou des outils, plus adaptés, apportés par des bibliothèques spécialisées ou non. Vous découvrirez certains de ces instruments dans les projets d'exemple.

La bibliothèque standard fournit également quelques outils mathématiques. Vous allez voir dans ce chapitre un exemple permettant de manipuler des nombres complexes. Le but ici n'est pas de présenter mathématiquement les nombres complexes, mais de donner un aperçu de ce qu'une bibliothèque peut fournir comme outils.

Rappels mathématiques

explications bof bof

Pour commencer, un petit rappel sur les nombres complexes. Pour ceux intéressés par les détails, vous pouvez consulter la page de Wikipédia correspondante (Nombre complexe) ou consulter un cours de mathématiques.

Les équations du second degré peuvent s'écrire de la façon suivante :

$$ ax^2 + bx + c = 0 $$

Si vous vous souvenez de vos cours de lycée, pour résoudre cette équation, on calcule le discriminant (réalisant ou $ \Delta $) donné par cette formule :

$$ \Delta = b^2 - 4ac $$

Si ce discriminant est positif, l'équation admet deux solutions réelles. S'il est nul, elle admet une solution réelle double. Le dernier cas, qui nous intéresse plus particulièrement ici, est que si le discriminant est négatif, cette équation n'admet pas de solutions réelles.

Cependant, on peut définir les nombres complexes de la façon suivante :

$$ z = x + i y $$

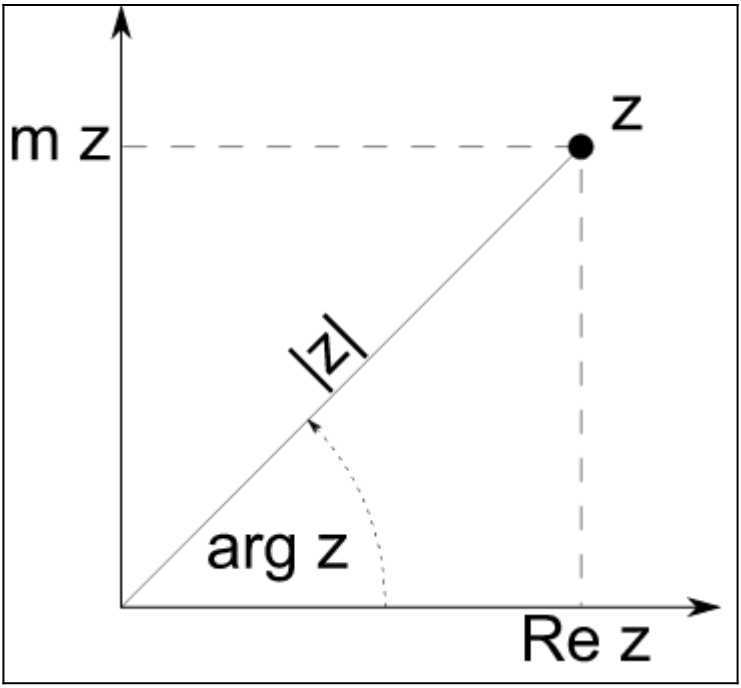
avec *x* et *y* réels et :

$$ i^2 = -1 $$

Dans ce cas, l'équation admet deux solutions complexes (conjuguées).

x est la partie réelle d'un nombre complexe et *y* est la partie imaginaire.

Il est classique de représenter les nombres complexes sur un plan, de la façon suivante :



On peut également définir un nombre complexe par l'angle entre l'axe des abscisses et la droite passant par l'origine et le point, que l'on appelle “argument” d'un nombre complexe, et la distance entre l'origine et le point, que l'on appelle “module”.

Nombres complexes en C++

Les nombres complexes sont fournis par la classe `std::complex` de la bibliothèque standard. Comme toujours, la documentation de cette classe se trouve sur le site cppreference. En consultant cette page, vous pouvez trouver au début le fichier d'en-tête à inclure : `<complex>`. Vous avez également des codes d'exemple à la fin.

Écrire une littérale d'un nombre complexe

Pour rappel, une littérale est une valeur écrite directement dans un code.

Pour écrire un nombre complexe en C++, une syntaxe a dû être définie pour cela. Une première approche peut être d'écrire directement un nombre complexe en suivant sa définition. Par exemple, pour le nombre :

$$ z = 2 + 3 i $$

On pourrait écrire :

```
2 + 3 * i;
```

Cette écriture est tout à fait valable. Cependant, sous cette forme, `i` correspond à l'écriture d'une variable (vous verrez cela par la suite), ce qui peut être limitant. Surtout que l'on a l'habitude en C++ d'utiliser la variable `i` comme indice (dans un tableau par exemple), d'où le risque de confusion. (Mais rien ne vous interdit par la suite, quand vous saurez créer une variable, de créer cette variable `i` avec le nombre complexe `z = 0 + 1 i`).

Pour éviter cette ambiguïté, une autre écriture a été choisie : un nombre imaginaire pur s'écrit sous forme d'une littérale réelle, suivie du suffixe `i`. Par exemple :

```
2.0 + 3.0i;
```

Dans ce code, la littérale `2.0` correspond à une littérale réelle et la littérale `3.0i` à un nombre imaginaire pur (en pratique, à une littérale réelle avec le suffixe `i`). Il est possible d'afficher directement un nombre complexe avec `std::cout` :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << "2+3i = " << (2.0 + 3.0i) << std::endl;
}
```

affiche :

```
2+3i = (2,3)
```

std::literals

Il existe différents types de préfixes et suffixes permettant de modifier une littérale. Certains sont définis dans le langage et ne nécessitent donc pas d'espace de noms. D'autres sont définis dans la bibliothèque standard, dans une bibliothèque externe ou par les utilisateurs. Ils peuvent dans ce cas être dans un espace de noms.

C'est le cas ici du suffixe `i`, qui se trouve dans l'espace de noms `std::literals`. Il faut donc avertir le compilateur que vous souhaitez utiliser cet espace de noms pour qu'il puisse utiliser `i`. (Essayez de compiler sans le `using`.)

Généralement, vous avez trois syntaxes possibles pour utiliser une fonctionnalité définie dans un espace de noms. Pour rappel :

```
using namespace std;
using std::cout;
std::cout;
```

L'utilisation d'un préfixe ou d'un suffixe nécessite qu'il n'y ait rien entre la littérale et le modificateur. Donc, il n'est pas possible d'écrire par exemple :

```
3.0std::literals::i  // erreur
```

C'est pour cette raison qu'il est plus pratique d'utiliser `using` pour travailler avec les modificateurs de littérales.

Vous voyez ici qu'un nombre complexe est affiché sous la forme `(partie réelle,partie imaginaire)`. On peut en particulier afficher `i` et vérifier que le carré de `i` vaut -1.

main.cpp

```
#include <iostream>
#include <complex>
```

```
int main() {
    using namespace std::literals;
    std::cout << "i = " << 1.0i << std::endl;
    std::cout << "i² = " << (1.0i * 1.0i) << std::endl;
}
```

affiche :

```
i = (0,1)
i² = (-1,0)
```

Le résultat affiché correspond bien aux valeurs attendues.

Notez bien que le nombre imaginaire `i` ne peut pas s'écrire directement `i` dans un code C++, puisque cela correspondrait à l'écriture d'une variable et non d'une littérale. Le `i` d'une littérale représentant un nombre imaginaire est un suffixe, il doit toujours suivre une littérale numérique.

Comparer des nombres complexes

Il est possible de comparer l'égalité (ou l'inégalité) des nombres complexes entre eux en utilisant les opérateurs `==` et `!=`, comme vous l'avez fait avec les nombres entiers et réels.

Par exemple :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((2.0 + 3.0i) == (2.0 +
3.0i)) << std::endl;
    std::cout << std::boolalpha << ((2.0 + 3.0i) != (3.0 +
2.0i)) << std::endl;
}
```

affiche :

```
true
true
```

Même si le calcul de i au carré donne 1, le résultat affiché correspond au nombre complexe `(-1,0)`. Mathématiquement, cela est correct :

$$ -1 + 0 i = -1 $$

Il n'est possible de comparer les nombres complexes que par égalité ou inégalité. Les comparaisons d'ordre (plus petit, plus grand, etc.) n'ont pas de sens pour les complexes.

Cependant, n'oubliez pas que même si deux valeurs sont mathématiquement identiques, le C++ est basé sur un typage fort et différencie les valeurs en fonction de leur type. Ainsi, même si "-1" (nombre entier) est égal à "-1.0" (nombre à virgule flottante) et à "(-1,0)" (nombre complexe), ce sont des valeurs différentes en C++.

Vous pouvez tester ces égalités en utilisant l'opérateur d'égalité `==`. Commençons par la comparaison d'un nombre complexe et d'un nombre à virgule flottante :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1.0)
<< std::endl;
}
```

affiche :

```
true
```

Dans ce cas, pas de problème, le résultat affiché est celui attendu. Si maintenant, vous testez avec un nombre entier :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1) <<
std::endl;
}
```

affiche :

```
main.cpp:6:51: error: invalid operands to binary expression
('complex<double>' and 'int')
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1) <<
std::endl;
                                   ~~~~~~~~~~~~~ ^  ~~
1 error generated.
```

Dans ce cas, le compilateur produit une erreur, indiquant qu'il ne sait pas comparer un nombre complexe et un nombre entier. Ce sont deux types différents pour lui.

Pour être plus précis, cela signifie que le compilateur connaît l'opérateur d'égalité `==` entre un complexe et un réel, mais qu'il n'en connaît pas entre un complexe et un entier.

La classe std::complex

Dans le message d'erreur précédent, vous pouvez remarquer que le compilateur interprète le calcul `1.0i * 1.0i` sous forme d'un type qui s'appelle `std::complex<double>`. Voyons plus en détail cela.

Une classe est un type, mais définie par un code C++ et non par le langage. Vous ne trouverez nulle part un fichier C++ qui définit `int` ou `float`, par contre le type `std::complex` est défini dans le fichier d'en-tête ”<complex>”.

Vous pourrez de la même façon créer vos propres types en créant des

classes, mais plus tard dans ce cours. (La création de classes a une importance particulière en programmation, on parle de “programmation orientée objet”.)

Cette classe est une **abstraction** représentant un nombre complexe :

- cela représente une version “manipulable par l'ordinateur” d'un concept mathématique et pas exactement ce concept mathématique (par exemple, comme vous l'avez vu, il n'est pas possible de comparer un entier avec un nombre complexe en C++, alors que cela peut avoir un sens en mathématique).
- vous n'avez pas besoin de savoir comment est écrit le code C++ définissant cette classe ou même comment l'ordinateur réalise les calculs. Tout ce qu'il vous faut connaître est l'interface publique, c'est à dire la partie de la classe accessible en dehors de la classe.

Cette notion d'abstraction est très importante à comprendre, puisque cela définit comment vous allez utiliser cette classe (interface publique) et ses limites (ce qui la différencie du modèle mathématique). En particulier pour les calculs numériques, n'oubliez pas que les nombres sur un ordinateur ont des limites (valeur minimale, valeur maximale, nombre maximal de chiffres après la virgule, etc.).

Pour terminer avec la notion `std::complex<double>` : vous avez vu que `std::complex` correspond donc au nom de la classe représentant un nombre complexe. Les nombres complexes sont représentés par deux nombres réels “x + y i”, donc la classe `std::complex` manipule également des nombres réels en interne. Pour le moment, vous n'avez pas vu à quel type correspond les nombres réels que vous avez écrit dans vos codes C++, mais sachez en fait que le C++ peut utiliser plusieurs types différents pour représenter des nombres réels.

Le type `double` est un de ces types, mais il en existe d'autres (`float`, `long double`, etc.). Les chevrons dans la définition de la classe `std::complex` permettent de préciser le type qui sera manipulé en interne par cette classe. Dit autrement, cela signifie que `std::complex` utilise le type `double` en interne lorsque vous écrivez

`std::complex<double>`, elle utilise `float` lorsque vous écrivez `std::complex<float>`, `MonType` lorsque vous écrivez `std::complex<MonType>`, et ainsi de suite.

(Notez bien que c'est toujours un type que vous devez mettre entre les chevrons et pas une valeur).

Pour créer une valeur de type `std::complex<double>`, vous avez vu que le plus simple est donc d'écrire une littérale numérique utilisant le suffixe `i`. Cependant, vous aurez besoin dans certains cas de créer un nombre complexe sans écrire de littérale. Par exemple, si vous souhaitez utiliser le résultat d'une expression pour calculer les parties réelle et imaginaire d'un nombre complexe :

$$ (2 * 3) + (4 * 5) i $$

Une première solution est de multiplier le résultat de l'expression de droite par le nombre imaginaire *i* (qui s'écrit donc `1.0i` en C++) :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << ((2.0 * 3.0) + (4.0 * 5.0) * 1.0i) << std::
endl;
}
```

affiche :

```
(6,20)
```

Une autre solution est d'appeler spécifiquement la classe `std::complex<double>` en passant les expressions entre parenthèses, sous la forme : `std::complex<double>(partie réelle, partie imaginaire)`. Concrètement, cela donne le code suivant :

main.cpp

```
#include <iostream>
```

```cpp
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::complex<double>(2.0 * 3.0, 4.0 * 5.0)
<< std::endl;
}
```

Ce qui affiche la même chose que précédemment.

Notez bien qu'il ne faut pas mettre dans cette écriture l'opérateur `+`, ni le nombre imaginaire *i*.

Cette syntaxe est nouvelle, donc pas forcément claire pour le moment. Mais pas d'inquiétude, vous verrez cela régulièrement, dans de nombreux codes. Retenez simplement l'idée générale : Le code `std::complex<double>(2.0, 3.0)` signifie que `std::complex` manipule en interne des nombres réels de type `double` et représente le nombre complexe *2 + 3i*.

Les opérations et fonctions sur std::complex

Pour terminer ce chapitre sur les nombres complexes, vous avez vu dans le chapitre sur les nombres réels que le C++ propose de nombreuses fonctions mathématiques pour les réels. C'est également le cas pour les nombres complexes. Cependant, toutes les fonctions sur les nombres réels ne sont pas forcément définies pour les nombres complexes.

Pour commencer, les nombres complexes définissent les opérations arithmétiques de base, comme l'addition et la soustraction entre complexes, ainsi que l'addition, la soustraction, la multiplication et la division entre un complexe et une nombre réel.

main.cpp

```cpp
#include <iostream>
#include <complex>
```

```
int main() {
    using namespace std::literals;

    std::cout << ((2.0 + 3.0i) + (4.0 + 5.0i)) << std::endl;
// addition
    std::cout << ((2.0 + 3.0i) - (4.0 + 5.0i)) << std::endl;
// soustraction

    std::cout << ((2.0 + 3.0i) + 4.0) << std::endl;
// addition
    std::cout << ((2.0 + 3.0i) - 4.0) << std::endl;
// soustraction
    std::cout << ((2.0 + 3.0i) * 4.0) << std::endl;
// multiplication
    std::cout << ((2.0 + 3.0i) / 4.0) << std::endl;
// division
}
```

affiche :

```
(6,8)
(-2,-2)
(6,3)
(-2,3)
(8,12)
(0.5,0.75)
```

En complément de ces opérations de base, les nombres complexes peuvent être utilisés avec différentes fonctions mathématiques. La syntaxe à utiliser est similaire à celle que vous avez vu pour les nombres réels :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;

    std::cout << real(2.0 + 3.0i) << std::endl;  // partie
réelle
```

```
    std::cout << imag(2.0 + 3.0i) << std::endl;  // partie
imaginaire
    std::cout << abs(2.0 + 3.0i) << std::endl;   // module
(valeur absolue en anglais)
    std::cout << arg(2.0 + 3.0i) << std::endl;   // argument
    std::cout << norm(2.0 + 3.0i) << std::endl;  // norme
    std::cout << conj(2.0 + 3.0i) << std::endl;  // conjugué
    std::cout << proj(2.0 + 3.0i) << std::endl;  //
projection
    std::cout << polar(2.0 + 3.0i) << std::endl; //
coordonnées polaires
}
```

affiche :

```
2
3
3.60555
0.982794
13
(2,-3)
(2,3)
((2,3),(0,0))
```

Il existe d'autres fonctions mathématiques sur les nombres complexes, qui s'utilisent de la même façon (voir la documentation pour la liste des fonctions : Documentation de std::complex) : fonctions exponentielles, puissances, trigonométriques et hyperboliques (voir la page de Wikipédia pour les explications sur ces fonctions mathématiques : Nombre complexe).

Chapitre précédent	Sommaire principal	Chapitre suivant