# Opérateurs, flux et autres abstractions

Nouveaux paradigmes de manipulation des données offerts par les langages à objets

# La surchage des opérateurs

Une classe de nombre complexes :

```
class Complex
{
private:
    double mRealPart;
    double mImaginaryPart;
public:
    Complex();
    ~Complex();
};
```

# Ajouter les opérations par des méthodes ?

```
class Complex
{
private:
    double mRealPart;
    double mImaginaryPart;
public:
    Complex();
    ~Complex();
    Complex Add(const Complex& aRightValue)
    {
        Complex result = *this;
        result.mRealPart += aRightValue.mRealPart;
        result.mImaginaryPart += aRightValue.mImaginaryPart;
        return result;
    }
};
```

- Ajoute bien les fonctionnalités mais pour faire une addition, on doit écrire :

```
Complex myResult = myInitialValue.Add(myAdditionalValue);
```

# Surcharcher les opérateurs ?

```
class Complex
{
private:
    double mRealPart;
    double mImaginaryPart;
public:
    Complex();
    ~Complex();
    Complex operator + (const Complex& aRightValue)
    {
        Complex result = *this;
        result.mRealPart += aRightValue.mRealPart;
        result.mImaginaryPart += aRightValue.mImaginaryPart;
        return result;
    }
};
```

- La syntaxe devient bien plus sympathique :

```
Complex myResult = myInitialValue + myAdditionalValue;
```

# Effectuer des conversions automatiques.

- On aimerait pouvoir écrire :

```
Complex myResult = 3.0;
```

- Comment faire ?

```
class Complex
{
private:
    double mRealPart;
    double mImaginaryPart;
public:
    Complex();
    Complex(double aValue): mRealPart(aValue), mImaginaryPart(0)
    {}
    …
};
```

- Le constructeur `Complex(double aValue)` est un constructeur avec un seul paramètre, sans « `explicit` », donc il peut-être appelé automatiquement pour convertie un nombre flottant en nombre « `Complex` ».

# Effectuer des conversions automatiques.

On aimerait pouvoir écrire :

```
Complex myComplex;
…
double realPart = (double)myComplex;
```

Pour ce faire, il faut introduire un opérateur de conversion :

```
class Complex
{
    …
public:
    Complex();
    ~Complex();
    Complex(double aValue): mRealPart(aValue), mImaginaryPart(0)
    {}
    operator double() const {
        return mRealPart;
    }
    …
};
```

# Limite des opérateurs de conversions définies dans la classe

```
Complex operator + (const Complex& aRightValue)
{
    Complex result = *this;
    result.mRealPart += aRightValue.mRealPart;
    result.mImaginaryPart +=
        aRightValue.mImaginaryPart;
    return result;
}
```

- L'élément à gauche est obligatoire de l'opération a obligatoirement le type « `Complex` ».

# Limite des opérateurs de conversions définies dans la classe

- Comment faire si on veut faire une addition entre un entier et un complexe ?
  - Solution 1 :
    - Convertir l'entier en « `Complex` »
    - Faire l'addition entre deux « `Complex` ».
  - Solution 2 :
    - Définir un nouvel opérateur + qui prend un entier comme premier argument et une classe « `Complex` » comme deuxième argument.

# Opérateurs définis en dehors de la classe

```
class Complex
{
    friend Complex operator +(int, const Complex&);
public:
    Complex();
    ~Complex();
};

Complex operator +(int aLeftValue,
    const Complex& aRightValue)
{
    Complex result = aRightValue;
    result.mRealPart += aLeftValue;
    return result;
}
```

# TD --- Partie 1

- Implanter les opérateurs pour les nombres complexes.
- Manipuler des nombres complexes.

# La notion de flux

- Un flux :
  - Permet d'écrire une liste de données de manière séquentielle vers l'extérieur.
  - Permet de lire une liste de données de manière séquentielle vers l'intérieur.
- Quelques flux que vous connaissez

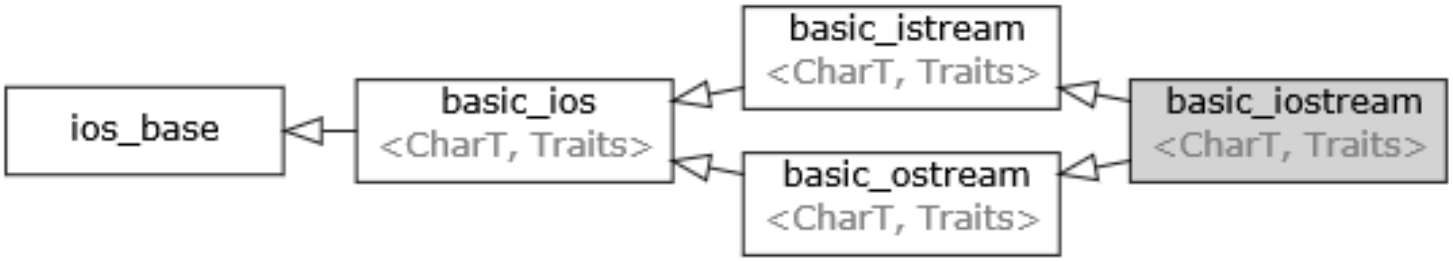
| Type de flux | Nom | Description |
|---|---|---|
| Entrant | std::cin | Flux standard d'entrée |
| Sortant | std::cout | Flux standard de sortie |
| Sortant | std::cerr | Flux standard pour les erreurs (non-bufferisé) |
| Sortant | std::clog | Flux standard pour les erreurs (bufferisé) |

# Les différentes catégories de flux

- Flux paramétrés par le type de caractères

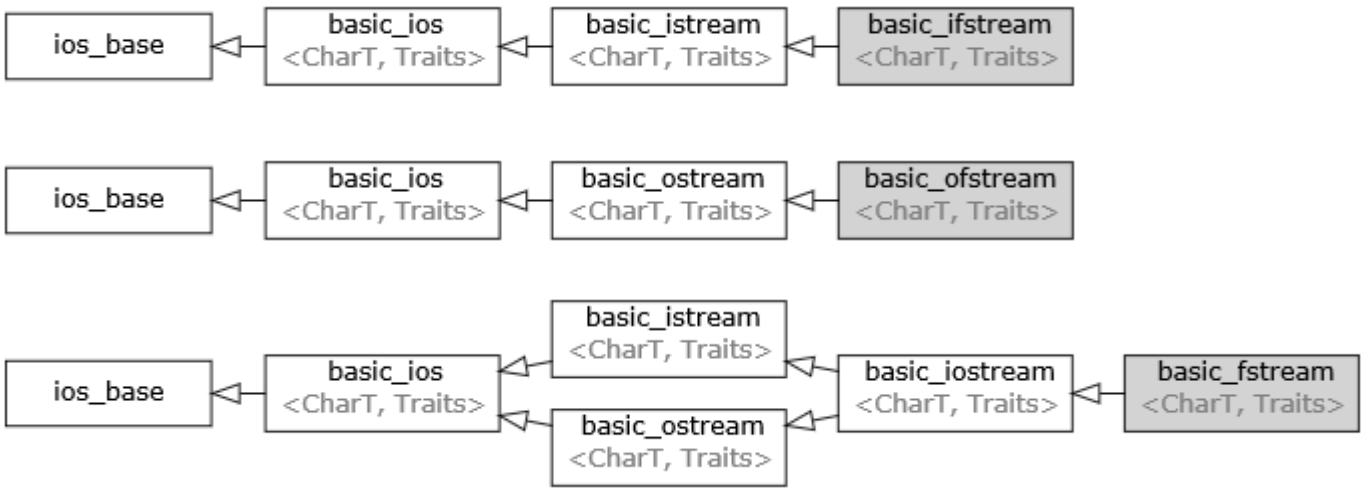
```
template<class charT,
     class traits =
std::char_traits<charT>> class basic_ios
```

- Direction du flux
  - Entrant : dérive de std::basic_istream
  - Sortant : dérive de std::basic_ostream
  - Entrant/Sortant : dérive de std::basic_iostream



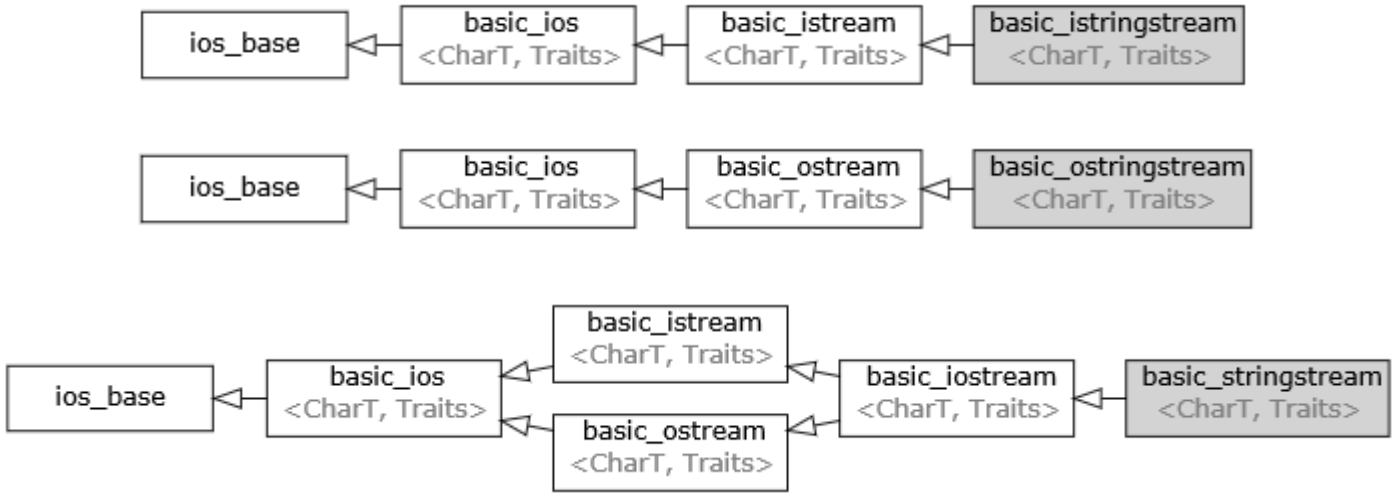
# Les différents types de flux par support

- Fichiers



# Les différents types de flux par support

- Les écritures vers ou à partir des chaînes de caractères



# L’idée : rendre les opérations de lecture/écriture indépendante de leur support

- Deux opérateurs surchargés :

Un opérateur de lecture:

```
std::cin >> myValue;
```

Un opérateur d’écriture:

```
std::cout << myValue;
```

# Chaque type défini des opérateurs surchargés

- Définition d'un opérateur pour l'écriture :

```
template<class charT, class traits = std::char_traits<charT>>
std::basic_ostream<charT, traits> operator <<
    (std::basic_ostream<charT, traits>& theStream,
    const Complex& theNumber) {...}
```

- Définition d'un opérateur pour la lecture :

```
template<class charT, class traits = std::char_traits<charT>>
std::basic_istream<charT, traits> operator >>
    (std::basic_istream<charT, traits>& theStream,
    const Complex& theNumber) {...}
```

# Comment définir des attributs de lecture/écriture

- Idée : utiliser des modificateurs

```
#include <iostream>
void print()
{
std::cout << true << " et " << false << std::endl;
//Affichage "normal"
std::cout << std::boolalpha; //On applique le manipulateur
std::cout << true << " et " << false << std::endl;
//Affichage "modifié"
}
```

- Qui affiche

1 et 0
true et false

# TD --- Partie 2

- Ajouter les opérations d'écriture et de lecture des nombres complexes.
- Définir un opérateur de flux qui définit si le chiffre est écrit comme au format x+iy ou au format $(\rho, \theta)$. Ce modificateur sera polar ou imaginary.