Patrons, Modèles & Généricité

Rendre des fonctions, objets paramétrables par des types et des valeurs

L‘idée simple des « Template (Patrons) »

- Une simple fonction « identité »

```
int identity(int x) { return x; }
string identify(string x) { return x; }
float identity(float x) { return x; }
```

- Cette fonction est là même pour tous les types. Cependant, nous devons écrire une fonction pour chaque type !
- Pourquoi ne pas déléguer cette tâche au compilateur ?

Principe des fonctions « Template »

- Idée : paramétrer la définition d'une fonction par un type :

```
template<class T>
T identity(T x) { return x; }
```

Ou

```
template<typename T>
T identity(T x) { return x; }
```

- La fonction `identity` devient une fonction qui prend un argument de type `T` et retourne une value de type `T`.

Que fait le compilateur C++ ?

```
int x = identity(3);
```

- Le compilateur cherche la fonction « `identity` » dans la table des fonctions.
- Le compilateur trouve un patron pour la fonction « `identity` » qui prend un paramètre de type `T`.
- Le compilateur identifie le type `T` avec le type de la valeur `3`, soit `int`.
- Le compilateur instantie le code de la fonction pour `identity` le `T=int` et crée le code :

```
int identity(int x) { return x; }
```

- Le compilateur appelle la fonction qu'il vient de créer.

Conséquences du processus de création des fonctions définies par des « templates »

- Le compilateur a besoin du code la fonction au moment où il rencontre l‘appel à la fonction
  - Le code de la fonction est présent dans le fichier .hpp et non pas .cpp.
- Le compilateur génère plusieurs fois la même fonction
  - L‘éditeur de liens doit supprimer les fonctions générées plusieurs fois.
- Le temps de compilation est augmenté par la génération du code des fonctions template.

La définition de classe paramétrée par des types

```
template<typename T>
class pair
{
   T x;
   T y;
   …
};
```

Paramètrez par des valeurs

```
template<typename T, size_t size>
class array
{
  T array[size];
  …
};
```

Crée un tableau de `type T` et de taille `size`.

TD --- Partie 1 & 2

- Création d‘une fonction de recherche templatée.
- Création d‘une classe templatée

Containeurs & Itérateurs

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

Séparer Structure de Données et Algorithmes

Idée : un algorithme de recherche d'un plus grand élément

```
template<class T>
T find_greatest_element(T theArray[], size_t theNumberOfElements)
{
    if (theNumberOfElements == 0)
        return T();
    T greatestElement = theArray[0];
    for (int index = 1; index < theNummberOfElements; index++)
    {
        if (theArray[index] > greatestElement)
            greatestElement = theArray[index];
    }
    return greatestElement;
}
```

Comment faire fonctionner cet algorithme sur d'autres structures de données que des tableaux ?

Rendre l‘algorithme générique

```
find_greatest_element(aCollection)
    si aCollection est vide:
        aucun plus grand élément.

    greatestElement <= premier élément de aCollection.
    tant qu'il existe un élément suivant de aCollection:
        nextElement <= élément suivant de aCollection
        si nextElement > greatestElement:
            greatestElement = nextElement;

    retourne greatestElement.
```

Input Iterator en C++

Un itérateur est une classe faisant référence à un élément appartenant à une séquence.

Un „input iterator“ offre les fonctions suivantes :

a == b	Test si deux itérateurs sont égaux, s‘ils sont égaux, c‘est qu‘ils font référence à le même élément
*a	Accède à l‘élément référencé par l‘itérateur
a++	Va à l‘élément suivant
a(b)	Crée un nouvel itérateur qui est la copie de l‘itérateur. Le nouvel itérateur créé référence le même élément que l‘itérateur initial
a = b	Affecte à un itérateur le contenu d‘une autre itérateur. a fait désormais référence au même élément que b.

La fonction „find_greatest_element“

```
template<class iterator>
iterator find_greatest_element(iterator itStart, iterator itEnd)
{
    if (itBegin == itEnd)
        return itEnd;
    iterator itGreatest = itStart;
    itStart ++;
    while(itStart != itEnd)
    {
        if(*itStart > *itGreatest)
            itGreatest = itStart;
        itStart ++;
    }
    return itGreatest;
}
```

Une structure de données « énumérable » en C++

- Doit fournir au moins deux fonctions

begin

Retourne un itérateur faisant référence à la première donnée présente dans la structure ou à l'itérateur désignant la fin des données.

end

Retourne l'itérateur désignant la fin des données de la structure.

Généralisation de l'usage

```
int test()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };
    std::vector<int> v(myints, myints + 8);
    int maxValue;
    std::vector<int>::iterator up =
      find_greatest_element(v.begin(), v.end(),
      maxValue);
    std::cout << "greatest value at position "
      << (up - v.begin()) << '\n';
    return 0;
}
```

TD --- Partie 3

- Utilisation d’itérateurs

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;
```

TD --- Partie 4

- Surcharge d‘opérateurs