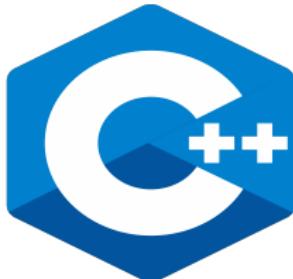


C++ : généricité et template

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Le concept objet

- Classe = { attributs } + { méthodes }
- **C++** : langage fortement typé ⇒ Les attributs ont **forcément** un nom et un type
- Le type d'attribut ne change pas pour toutes les instances d'une classe
- Les méthodes/fonctions dans un langage fortement typé :
 - pour un paramètre donné, elles ne peuvent accepter qu'un type (ou quelques autres qui lui sont compatibles)
 - idem pour la valeur de retour

Problématique

- Et si on voulait qu'une méthode effectue les mêmes opérations quel que soit le type d'attribut
 - **somme** pour **entiers** ou **réels**,
 - **concaténation** pour **chaînes de caractères**,
 - **ou logique** pour **booléens**...
 - ...
- **Impossible sans définir plusieurs classes (une pour chaque type) et une interface ou en imposant le type Object et en utilisant plusieurs cast...**

C++

C++11 nous offre une solution plus élégante

généricité + templates

C++11 nous offre une solution plus élégante

généricité + templates

Comment ça marche pour les fonctions ?

- Ne pas définir de type pour les paramètres et/ou la valeur de retour d'une fonction
- Définir un type générique qui ne sera pas précisé à la définition de la fonction
- À l'appel d'une fonction, on précise le(s) type(s) à utiliser
- **On peut donc choisir pour chaque appel un type différent**

Comment ça marche pour les classes ?

- Ne pas définir de type pour les attributs et/ou méthodes
- Définir un type générique qui ne sera pas précisé à la création de la classe
- À l'instanciation de la classe, on précise le(s) type(s) à utiliser
- **On peut donc choisir pour chaque objet un type différent**

Objectif

Avoir une fonction qui retourne

- la somme si elle reçoit deux paramètres de type nombre
- la concaténation si elle reçoit deux paramètres de type chaîne de caractère
- le résultat d'un ou logique si elle reçoit deux paramètres de type booléen

Dans le fichier main.cpp et avant la fonction principale, on déclare une fonction opPlus() qui utilise un template

```
template <typename T>
T opPlus(T a, T b)
{
    return a + b;
}
```

Dans le fichier `main.cpp` et avant la fonction principale, on déclare une fonction `opPlus()` qui utilise un template

```
template <typename T>
T opPlus(T a, T b)
{
    return a + b;
}
```

Le résultat est

```
int main()
{
    cout << opPlus<int>(2, 3) << endl;
    // affiche 5
    cout << opPlus<double>(2.5, 3.5) << endl;
    // affiche 6
    cout << opPlus<string>("bon", "jour") << endl;
    // affiche bonjour
    cout << opPlus<bool>(true, false) << endl;
    // affiche 1
    return 0;
}
```

Objectif

On veut avoir une classe `Point` avec deux attributs `abs` et `ord` qui doivent avoir le même type

- soit `short`
- soit `int`
- soit `double`
- ...

Dans le fichier `Point.h`, on définit la classe template dont toutes les méthodes sont inline (donc pas de fichier `Point.cpp`)

```
#ifndef POINT_H
#define POINT_H

template <typename T>
class Point
{
public:
    Point(T abs, T ord) : abs(abs), ord(ord) {}
    void setAbs(T abs) { this->abs = abs; }
    void setOrd(T ord) { this->ord = ord; }
    T getAbs() { return abs; }
    T getOrd() { return ord; }
    string toString() { return "(" + to_string(abs) + ", " + to_string(
        ord) + ")"; }

private:
    T abs;
    T ord;
};

#endif
```

Remarque

Les méthodes d'une classe template doivent être définis dans le .h

- soit en `inline`
- soit après la classe

C++

On peut maintenant tester avec des int et des float

```
int main()
{
    Point<int> p1(2, 5);
    cout << p1.toString() << endl;
    // affiche (2, 5)

    Point<float> p2(2.3, 5.8);
    cout << p2.toString() << endl;
    // affiche (2.300000, 5.800000)
    return 0;
}
```

On peut maintenant tester avec des int et des float

```
int main()
{
    Point<int> p1(2, 5);
    cout << p1.toString() << endl;
    // affiche (2, 5)

    Point<float> p2(2.3, 5.8);
    cout << p2.toString() << endl;
    // affiche (2.300000, 5.800000)
    return 0;
}
```

Les deux objets p1 et p2 sont de type différent car ils prennent deux paramètres templates différents.

Objectif

On veut avoir une classe Tableau qui crée un tableau selon une taille et un type précis.

C++

Contenu de Tableau.h

```
#ifndef TABLEAU_H
#define TABLEAU_H

template <typename T, int taille>
class Tableau
{
public:
    Tableau(T arr[])
    {
        for (int i(0); i < taille; i++)
        {
            data[i] = arr[i];
        }
    }
    T getMax()
    {
        T maxi = data[0];
        for (int i = 1; i < taille; i++)
        {
            maxi = max(data[i], maxi);
        }
        return maxi;
    }
private:
    T data[taille];
};

#endif
```

On peut maintenant tester en passant un type (`int`) et une valeur (taille)

```
int main()
{
    int tab[] = {2, 5, 7, 3};
    Tableau<int, 4> tableau(tab);
    cout << tableau.getMax();
    // affiche 7
    return 0;
}
```

Objectif

Supposant qu'on utilise souvent un type très long à écrire, il est possible de lui associer un alias.

C++

On commence par définir l'alias

```
using arrayInt4 = Tableau<int, 4>;
```

C++

On commence par définir l'alias

```
using arrayInt4 = Tableau<int, 4>;
```

On peut ensuite l'utiliser

```
int main()
{
    int tab [] = {2, 5, 7, 3};
    arrayInt4 obj (tab);
    cout << obj.getMax() << endl;
    // affiche 7

    int tab2 [] = {1, 6, 2, 3};
    arrayInt4 obj2 (tab2);
    cout << obj2.getMax() << endl;
    // affiche 6
}
```

Objectif

Comme pour les fonctions et les méthodes, on peut préciser des valeurs par défaut à nos paramètres template.

C++

Définissons les valeurs par défaut pour les paramètres de template

```
#ifndef TABLEAU_H
#define TABLEAU_H

template <typename T = int, int taille = 4>
class Tableau
{
public:
    Tableau(T arr[])
    {
        for (int i(0); i < taille; i++)
        {
            data[i] = arr[i];
        }
    }
    T getMax()
    {
        T maxi = data[0];
        for (int i = 1; i < taille; i++)
        {
            maxi = max(data[i], maxi);
        }
        return maxi;
    }
private:
    T data[taille];
};

#endif
```

On peut maintenant tester sans préciser les paramètres de template

```
int main()
{
    int tab[] = {2, 5, 7, 3};
    Tableau<> tableau(tab);
    cout << tableau.getMax();
    // affiche 7
    return 0;
}
```

Exercice

Nous voudrons créer une nouveau conteneur `map` appelé `Dictionary` qui

- fonctionne avec deux types génériques `T1` et `T2`
- stocke les clés dans un `vecteur<T1>` et les valeurs dans un `vecteur<T2>`
- possède une méthode `get (i)` qui retourne un tuple contenant la clé et la valeur situées à la position `i`
- considère que `int` est le type par défaut pour la clé et pour la valeur
- surcharge l'opérateur `+=` pour ajouter un nouveau couple à la fin
- accepte les doublons pour les clés et pour les valeurs