Un outil pour la généricité

christophe.labourdette(at)cmla.ens-cachan.fr

Octobre 2016

Templates

Définition

Les templates permettent de définir des familles de classes, structures ou fonctions.

Les templates sont aussi appelés ”patrons”

- Il s’agit d’un des mécanisme les plus puissant du C++
- Il permet la généricité
- Il permet de faire de la méta-programmation
- Il permet d’avoir un code extensible
- Difficile à manipuler
- A la base de la STL

Précision

paramétrage

Les familles sont paramétrées par des types :

1. **abstraits** lors de la conception du template,
2. il ne prennent corps que lors de leur **instanciation**,
3. c'est à dire leur **utilisation**, à la compilation.

Il est possible de définir des templates :

- sur des structures
- sur des classes
- sur des fonctions

fonction

```
template <typename T>
inline T const& max (T const& a, T const& b)
{    return a<b ? b : a; }

int main()
{
  int i = 127;
  std::cout << "max(23,i)=" << ::max(23,i);
  std::cout << std::endl;
  double x1 = -7.4;
  double x2 = 89.45;
  std::cout << "max(" << x1 << "," << x2 << ")=";
  std::cout << ::max(x1,x2) << std::endl;
}
```

instanciation

Le remplacement des paramêtres template par des types concret est appelé **instanciation**
Tout se passe comme si on avait déclaré :

```
inline int const& max(int const&,int const&);
inline double const& max(double const&,double const
```

Autrement dit on a compilé deux fois.
Cela ne fonctionne que si la relation $<$ existe pour le type qui remplace *T* sinon il y a une erreur lors de la compilation. par exemple

```
std::complex<float> c1,c2;
...
::max(c1,c2);  // ERREUR
```

déduction des arguments

Dans l'exemple du max, les deux arguments doivent être identiques.

```
max(5,9);   //ok
max(8,5.7);   //erreur
```

On peut alors spécifier explicitement le type *T*

```
max<double>(5,9.7);   //ok
```

Ou alors préciser qu'il peut y avoir deux arguments.

fonction 2

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
  return a < b ? b : a;
}
...
max(3,3.7); // OK T1 est le type de retour
```

fonction 3

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b)
{
  return a < b ? b : a;
}
...
max<int,double,double>(3,3.7); // lourd
max<double>(3,3.7); // retourne un double
```

Surcharge

On peut surcharger une fonction template :

```
inline int const& max(int const& a, int const& b)
    return a < b ? b : a;
}

template < typename T>
inline T const& max(T const& a, T const& b) {
    return a < b ? b : a;
}

inline T const& max(T const& a, T const& b,
     T const& c) {
    return ::max (::max(a,b), c);
}
```

Surcharge (suite)

Voici quelques exemples d'utilisation :

```
int main() {
...
::max(5,87,23);   // appel du max avec 3 arguments
::max(5.0, 87.0); // appel max<double>
::max('j','a');   // appel max<char>
::max(5,87);      // appel nontemplate
::max<>(5,87);    // appel max<int> template
::max('j',31.5); // appel nontemplate pour 2 ints
...
}
```

Pile.hpp

L'exemple de la déclaration d'une classe implémentant une pile d'éléments.

```
#include <vector>

 template <typename T> Class Pile {
 private :
     std::vector<t> elts; // element
 public :
     void push(T const&); // push element
     void pop();          // pop element
     T top() const;       // retour element du haut
     bool vide() const { // teste si pile vide
         return elts.vide();
    }
 }
```

Pile.hpp (suite)

Pour être bien formée la classe doit également avoir.

```
template <typename T>
class Pile {
    ...
     Pile (Pile<T> const&);
     Pile<T>& operator= (Pile<T> const&);
    ...
};
```

Pile.cpp

```
template <typename T>
void Pile<T>::push (T const& elem) {
  elts.push_back(elem);
}

template<typename T>
void Pile<T>::pop ()
{
  if (elts.empty()) {
    std::cout << "Pile␣vide␣...\n";
    exit (1);
  }
  elts.pop_back();
}
```

Pile.cpp (suite)

```
template <typename T>
T Pile<T>::top () const {
    if (elts.empty())
    {
        std::cout << "Pile vide ...\n";
        exit (1);
    }
    return elts.back();
}
```

```cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "Pile.hpp"
int main()
{
    Pile<int>           intPile;    // Pile de ints
    Pile<std::string> stringPile; // Pile de strings
    // manipulate int Pile
    intPile.push(7);
    std::cout << intPile.top() << std::endl;
    // manipulate string Pile
    stringPile.push("hello");
    std::cout << stringPile.top() << std::endl;
    stringPile.pop();
    stringPile.pop();
}
```

utilisation suite

Une fois compilé on obtient l'affichage suivant :

```
7
hello
Pile vide ...
```

la compilation

Mais il y a un hic...
si j'ai construit mon programme en utilisant le modèle d'inclusion classique : Pile.hpp, Pile.cpp et main.cpp.
la compilation **g++ -o expile main.cpp Pile.cpp** donne :

```
/tmp/cc6v2a2D.o: In function `main':
exPile.cpp:(.text+0x37): undefined reference
to `Pile<int>::push(int const&)'
exPile.cpp:(.text+0x43): undefined reference
to `Pile<int>::top() const'
...
```

==> Supprimer Pile.cpp ou l'inclure

Spécialisation

Il est possible de spécialiser une classe template pour certains arguments template.

```
template<>
class Pile<std::string> {
...
};

void Pile<std::string>::push(std::string const& elem
{
    elts.push_back(elem);
}
```

Spécialisation partielle

Il est possible de spécialiser partiellement une classe template pour certains arguments template.

```
template <typename T1, typename T2>
class MaClasse { ...
```

Si on veut les mêmes types ...

```
template<typename T, typename T>
class MaClasse { ...
```

Si on veut le deuxième argument entier ...

```
template <typename T>
class MaClasse<T, int> { ...
```