Les patrons de fonctions

- Nous allons maintenant introduire une fonctionnalité très puissante de C++: les patrons, ou *template* en Anglais (ce cours utilisera indistinctement les deux appellations).
- Pour comprendre tout l'intérêt de ce concept, il faut se souvenir de la surcharge des fonctions.

  Si l'on voulait introduire une fonction *min* sur les entiers qui renverrait le plus petit de deux entiers passés en paramètres, on créerait cette fonction, mais on ne pourrait pas l'utiliser pour des *float* etc. Il faudrait créer une fonction par type de données que l'on veut comparer.

Un exemple

```
#include <iostream>
using namespace std;

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 1: code40.cpp

Dans cet exemple, on définit la fonction *minimum* une fois pour toute et on peut l'utiliser quel que soit le type passé en paramètre.

En fait, le compilateur va générer, de manière transparente pour l'utilisateur, autant de fonctions que nécessaire selon les types de paramètres qu'on passera à la fonction.

Il y a un seul bémol à cela en comparaison de la surcharge de fonction: l'algorithme ne varie pas en fonction du type des paramètres.

Un exemple

```
#include <iostream>
using namespace std;

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 1: code40.cpp

Concernant la syntaxe, on commence donc par le mot clé *template*. Ensuite, le contenu des chevrons définira le caractère générique de notre fonction.

Ici *typename T* définira donc le type générique *T* qu'on pourra utiliser au sein de notre fonction.

Paramètres expression

```cpp
#include <iostream>
using namespace std;

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 2: code41.cpp

Voici un exemple plus complexe.

On veut une fonction qui retourne le plus grand élément d'un tableau qu'on lui fournit en paramètre.

Il n'y a pas de raison de se limiter aux tableaux d'un type particulier.

En fait, tant qu'une relation de comparaison peut être définie entre deux éléments du tableau, notre algorithme peut fonctionner.

Paramètres expression

```
#include <iostream>
using namespace std;

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 2: code41.cpp

On définit donc notre fonction comme une fonction *template*, prenant un pointeur sur type en paramètre ainsi qu'un entier non signé qui contiendra le nombre d'éléments du tableau.

On remarque par ailleurs que des types non "templatés" peuvent entrer comme paramètres d'une fonction *template*, il s'agit de paramètres expression.

L'algorithme ensuite est classique, en prenant soin d'utiliser le type *template* quand c'est nécessaire.

Surdéfinition de fonctions template

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

template <typename T>
T maxtab(T* arr, T* arr2,
         unsigned int n, unsigned
             int n2)
{
  T tmp  = maxtab(arr, n);
  T tmp2 = maxtab(arr2, n2);
  return (tmp < tmp2) ? tmp2 : tmp;
}
```

Listing 3: code42_1.cpp

On peut surcharger une fonction *template* en faisant varier son nombre d'éléments ou le type de ceux-ci.

Ici nous avons surchargé la fonction *maxtab* en donnant la possibilité de renvoyer le plus grand élément de deux tableaux.

Spécialisation

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

string maxtab(string* arr, unsigned int n)
{
  string tmp = to_lower(arr[0]);
  for (unsigned int i=0; i<n; i++)
    {
      if (to_lower(tmp)< to_lower(arr[i]))
        tmp = arr[i];
    }
  return tmp;
}
```

Listing 4: code42_2.cpp

On peut également définir un *template* pour un algorithme général qui est valable quel que soit le type, mais aussi spécialiser une fonction, c'est-à-dire définir un algorithme pour un type particulier.

Ici, dans le cas d'un tableau de *string*, l'opérateur $<$, pour comparer deux *strings*, existe mais est sensible à la casse: on ne veut pas cela ici. On spécialise ici la fonction *maxtab* et on utilise une fonction non standard *to_lower* convertissant une chaîne de caractères en minuscules.

Pour finir

Enfin, il n'est pas forcément évident d'écrire et de spécifier une fonction template. En effet, il faut faire attention aux cas ambigüs - c'est-à-dire où le compilateur ne sait pas s'il doit utiliser une fonction plutôt qu'une autre car les deux conviennent.

Par ailleurs, la règle pour les patrons de fonctions est que le type doit convenir "parfaitement", c'est-à-dire qu'un *const int* n'est pas un *int* etc.

Les patrons de classes

- Il existe un mécanisme similaire aux fonctions template pour les classes.
- Bien que semblable aux patrons de fonctions sur de nombreux points, il existe des différences avec les templates de classe.
- On va voir que cela permet d'implémenter un code générique et réutilisable.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 5: Vector.hpp

On se souvient de la classe *Vector* que nous avions définie dans ce cours, dans la partie *Surcharge d'opérateurs*.

Celle-ci, bien que déclarant une surcouche "objet" à des tableaux d'entiers, n'était pas utilisable si nous voulions stocker d'autres types. Il aurait alors fallu tout refaire.

Perte de temps, d'énergie ...

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

En pratique, comme on le voit ci-contre, on définit les types dans l'entête de la déclaration de la classe, de la même façon que pour les fonctions template.

On remarquera aussi un point important: à l'inverse des classes que nous déclarons d'habitude, ici tout est dans le même fichier!

En effet, le code, ainsi que la déclaration de la classe ne sont, en somme, qu'une déclaration. Le code n'est vraiment généré qu'à la compilation, à partir de ce template, en fonction des besoins.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

En résumé, on adoptera les conventions suivantes dans ce cours:

- Déclaration d'une classe → fichier .hh
- Définition d'une classe → fichier .cpp
- Déclaration d'un template de classe → fichier .hpp

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

Comme on n'a, au final, déclaré qu'un template de classe, celui-ci ne se compile pas "séparément". Il ne sera compilé que s'il est inclus dans un fichier de code et que si ce code l'utilise!

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 5: main_Vector.cpp

Pour utiliser le patron de classe, on va devoir instancier le type, lors de la déclaration de notre variable.

Ainsi, on crée un objet *vect*, de type *Vector<double>*, soit un *Vector* dont le type sera des *double*.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 4: main_Vector.cpp

Et pour le compiler?

On ne compile que le fichier contenant le *main*, car c'est, au final, le seul fichier cpp de notre programme ici.

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4: Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 5: Point.hpp

```
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 6: main_Vector2.cpp

Dans cet exemple, on a ajouté la structure template *Point*, qui s'utilise comme une classe template, avec les particularités liées aux structures.

On peut ainsi définir un type *Point* dont le type générique est un entier.

On peut alors aussi définir un vecteur de *Point* d'entiers!

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 4 : Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 2 : Point.hpp

```
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 3 : main_Vector2.cpp

La structure *Point* est très simple, elle ne contient que deux variables de type T générique.

On notera aussi la surcharge de l'opérateur << pour pouvoir afficher un objet *Point*.

Comme *Point* est un type template, il est "logique" qu'une fonction l'utilisant soit aussi template ··· sinon quel type de *Point* le compilateur instancierait-il?

Pour finir sur les patrons de classe

- Les patrons de classe sont un outil très puissant et largement utilisés par les développeurs pour créer de nouvelles librairies.
- On verra, dans le cadre de ce cours, la librairie STL pour Standard Template Library, qui définit ainsi de nombreux outils rapides et puissants.
- La librairie Boost, célèbre également, est une librairie basée sur les template.
- En maths, par exemple, la librairie Eigen++ (https://eigen.tuxfamily.org/) est une librairie pour l'algèbre linéaire et contient des algorithmes très optimisés.

Pour finir sur les patrons de classe

- La notion de template est plus vaste que celle abordée dans ce cours.
- Ainsi, on n'a pas abordé les notions de type expression, ni la spécialisation de classe template, ou la spécialisation partielle. Nous ne parlerons pas non plus de meta-programmation ou de template récursifs.
- Bien qu'utiles et intéressantes, ces notions sortent de l'objectif de ce cours qui est une introduction au C++ et à la POO.