

Cours C++

Patrons de fonctions et de classes

Notions de patrons ou template

- ▶ la **surdéfinition de fonctions** permet de donner un nom unique à plusieurs fonctions dont le contexte d'appel, c'est-à-dire les arguments de la fonction, diffère,
- ▶ la **redéfinition** intervient entre classes héritées et permet de spécialiser une fonction membre suivant les buts de la classe dérivée,
- ▶ les **patrons ou template de fonction** permettent d'écrire une seule fois la définition d'une fonction afin que le compilateur puisse l'adapter au type des arguments d'entrée ou de valeur de retour.

Notions de patrons ou template

- ▶ la **surdéfinition de fonctions** permet de donner un nom unique à plusieurs fonctions dont le contexte d'appel, c'est-à-dire les arguments de la fonction, diffère,
- ▶ la **redéfinition** intervient entre classes héritées et permet de spécialiser une fonction membre suivant les buts de la classe dérivée,
- ▶ les **patrons ou template de fonction** permettent d'écrire une seule fois la définition d'une fonction afin que le compilateur puisse l'adapter au type des arguments d'entrée ou de valeur de retour.

Notions de patrons ou template

- ▶ la **surdéfinition de fonctions** permet de donner un nom unique à plusieurs fonctions dont le contexte d'appel, c'est-à-dire les arguments de la fonction, diffère,
- ▶ la **redéfinition** intervient entre classes héritées et permet de spécialiser une fonction membre suivant les buts de la classe dérivée,
- ▶ les **patrons ou template de fonction** permettent d'écrire une seule fois la définition d'une fonction afin que le compilateur puisse l'adapter au type des arguments d'entrée ou de valeur de retour.

Illustration de l'utilisation de fonction patron

```
int min(const int a_, const int b_)  
{  
    return a_ < b_ ? a_ : b_;  
}
```

```
float min(const float a_, const float b_)  
{  
    return a_ < b_ ? a_ : b_;  
}
```

```
char min(const char a_, const char b_)  
{  
    return a_ < b_ ? a_ : b_;  
}
```

Illustration de l'utilisation de fonction patron

```
template<typename T> T min(const T & a_, const T & b_)
{
    return a_ < b_ ? a_ : b_;
}
```

- lors de la compilation, suivant le type des arguments fournis, le compilateur **enregistre** c'est-à-dire implémente le mécanisme adéquat

```
int main()
{
    int i1 = 2, i2 = 7;
    float f1 = 3.4, f2 = 5.6;
    char c1 = 'd', c2 = 'z';

    cout << "min(i1,i2)  = " << min(i1,i2) << endl;
    cout << "min(f1,f2)  = " << min(f1,f2) << endl;
    cout << "min(c1,c2)  = " << min(c1,c2) << endl;
}
```

Illustration de l'utilisation de fonction patron

```
template<typename T> T min(const T & a_, const T & b_)
{
    return a_ < b_ ? a_ : b_;
}
```

- lors de la compilation, suivant le type des arguments fournis, le compilateur **enregistre** c'est-à-dire implémente le mécanisme adéquat

```
int main()
{
    int i1 = 2, i2 = 7;
    float f1 = 3.4, f2 = 5.6;
    char c1 = 'd', c2 = 'z';

    cout << "min(i1,i2)  = " << min(i1,i2) << endl;
    cout << "min(f1,f2)  = " << min(f1,f2) << endl;
    cout << "min(c1,c2)  = " << min(c1,c2) << endl;
}
```

Illustration de l'utilisation de fonction patron

- C++ permettant la surcharge des opérateurs, la fonction patron `min` peut donc s'utiliser avec des objets plus complexes

```
class point {  
public:  
    double norme() const;  
private:  
    double m_x;  
    double m_y;  
};  
  
bool operator<(const point & p1_, const point & p2_)  
{  
    return p1_.norme() < p2_.norme();  
}  
  
...  
point my_point1(1,1), my_point2(2,2);  
cout << "min(p1,p2) =" << min(my_point1, my_point2) << endl;
```


Utilisation de patron de classe

La classe n'étant qu'une généralisation de la notion de type, il est par conséquent possible de définir des patrons de classe

```
template<typename T> class point
{
public:
    point(T abs_ = 0, T ord_ = 0) : m_x(abs_), m_y(ord_) {}
private:
    T m_x;
    T m_y;
};

int main()
{
    point<double> my_point1(5.0, 3.0);
    point<char>   my_point2('a', 'b');
}
```