

## Chapitre 10

### Patrons (Templates) de fonctions

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 10 : Patrons (Templates) de fonctions

188

### 1. Généralités

Modèle que le compilateur utilise pour créer des fonctions au besoin.

Exemple: Nous désirons écrire une fonction qui permet d'inter changer deux valeurs ...

\* deux int

```
void exchange(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

\* deux double

```
void exchange(double& a, double& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

\* deux char

```
void exchange(char& a, char& b){  
    char temp = a;  
    a = b;  
    b = temp;  
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

On constate que nous sommes en présence du même code, seul le type traité qui diffère,  
=> Solution: définir un patron pour cette fonction.

```
template <class T>
void exchange(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

T est un paramètre qui prend la place d'un type, n'importe quel type (int, double, char etc.)

2. Instanciation

\* cas de int

```
int x=10,y=20;
```

// x et y sont de type int, le compilateur instancie T à int et génère la fonction: void exchange(int&, int&)

exchange(x,y);

\* cas de double

```
double v=20.5,w=10.8;
```

Chapitre 10 - Patrons (Templates) de fonctions

// v et w sont de type double, le compilateur instancie T à double et génère la fonction:

```
void exchange(double&, double&)
```

exchange(v,w);

\* cas de char

```
char c1='a',c2='h';
```

// c1 et c2 sont de type char, le compilateur instancie T à char et génère la fonction:

```
void exchange(char&, char&)
```

exchange(c1,c2);

\* cas de la classe compte (voir chapitre 7 "Construction, destruction, initialisation et recopie")

```
compte c(120.98), d(345.86);
```

// c et d sont de type compte, le compilateur instancie T à compte et génère la fonction:

```
void exchange(compte&, compte&)
```

exchange(c,d);

Puisque la fonction exchange contient les deux lignes suivantes:

```
T temp = a; // appel du constructeur de recopie.
```

```
a = b; // appel de l'opérateur d'affectation.
```

Faire attention aux cas où la classe contient un membre de la partie donnée alloué dynamiquement, si c'est le cas il faudra définir le constructeur de recopie ainsi que l'opérateur d'affectation. Agit comme la surdéfinition de fonctions, sauf que nous somme en présence d'une seule définition qui va être valable pour tous les types: {int, double, char etc.} C'est le compilateur qui se charge d'instancier le patron de fonction au besoin, ce qui entraîne moins de risques d'erreurs et permet d'éviter tous les types de pléonasme.

3. Exemple affichage d'un tableau générique

```
template <class Z>
void affiche (Z tab[], int taille) {
    for (int i=0;i<taille;i++) cout << tab[i] << endl;
}

int main() {
    double tab1[3] = {1.5,2.4,5.8};

    // Créé une version pour les double, affiche toutes les valeurs du tableau tab
    affiche(tab1,3);
    char tab2[4] = {'a','b','c','d'};

    // Créé une version pour les char et n'affiche que les deux valeurs du tableau tab
    affiche(tab2,2);
    compte C1(345.68);
    compte C2(999.99);
    compte tab3[2]= {C1,C2};

    // Pour que ça marche, il faut surcharger dans la classe compte, l'opérateur <<
    affiche(tab3,2);

    return 0;
}
```

4. Répartition des déclarations de patrons de fonctions entre .h/.cpp

Les modèles de fonctions doivent être déclarés dans le fichier \*.h

```
// Fichier affiche.h

#include <iostream>

// ifndef/define/endif sont la pour empêcher de multiples inclusions du fichier d'en-tête.
#ifdef T_AFF
#define T_AFF
#else
    template <class Z>
    void affiche (Z tab[], int taille) {
        for (int i=0;i<taille;i++) cout << tab[i] << endl;
    }
#endif

// fichier compte.h

#include <iostream>

#ifdef H_COMP
#define H_COMP
#else
    class compte {
        double solde;
    public:
        compte(double m):solde(m) {}
        friend ostream& operator<<(ostream&, const compte&);
    };
#endif
```

```
ostream& operator<<(ostream& out,const complexe a) {
    cout << "solde: " << a.solde ;
    return out;
}

#endif
// Fin fichier compte.h

// Fichier prgtest.cpp
// Le fichier qui contient la fonction main

#include "affiche.h"
#include "compte.h"
using namespace std;
int main() {
    double tabl[3] = {1.5,2.4,5.8};
    affiche(tabl,3);

    char tab2[4] = {'a','b','c','d'};
    affiche(tab2,2);

    char* tab3[4] = {"Fred","Joe","Paul","Marie"};

    // On affiche réellement la chaîne, pas besoin de spécifier le *
    affiche(tab3,4);

    compte C1(345.68);
    compte C2(999.99);
    compte tab4[2] = {C1,C2};
    affiche(tab4,2);

    return 0;
}
// Fin fichier prgtest.cpp
```

```
// Commande de compilation

g++ -Wall -o prgexe prgtest.cpp sinon g++ -pedantic -o prgexe prgtest.cpp

// Affichage en sortie

1.5
2.4
5.8
a
b
Fred
Joe
Paul
Marie
solde: 345.68
solde: 999.99
```

5. compilateur et patron de fonctions

Le compilateur cherche dans ...

- Les fonctions ordinaires en cherchant une signature identique (un match exact) des types d'arguments,
  - Les patrons de fonctions avec une signature identique,
  - Les fonctions ordinaires, en essayant promotions et conversions,
- sinon erreur de compilation.

## 6. Possibilités de paramètres de type

### 6.1. Généralités

Les paramètres de type peuvent être utilisés à tout endroit raisonnable dans la fonction, par exemple:

```
- template <class T>
- double bidoz(int n, T a, double z) {...}
- T x;
- T* ptr;
- ptr = new T;

etc.
```

### 6.2. Utilisation dans un cas simple

```
template <class T>
void echange(T&,T&);
```

(Pas forcément deux arguments, on peut avoir aussi une fonction à un seul argument).

### 6.3. Utilisation avec un type fixe

```
template <class T>
void affiche(T[],int);
```

### 6.4. Utilisation avec plus d'un paramètre générique

```
template <class T>
void copier(T[],int);
```

Ou encore ...

```
template<class T, class U>
void copier(T dest[], U source[],int n)
```

### 6.5. Exemple recopie de tableaux

```
// Fichier copier.h

#include <iostream>

#ifdef T_COPIER
#define T_COPIER

template<class T, class U>
void copier(T dest[], U source[],int n) {
    for (int i=0;i<n;i++) dest[i] = (T) source[i];
}

#endif
```

```
// Fichier copier.cpp

#include "copier.h"
#include "affiche.h"

int main() {

    double dtab[3] = {1.1,2.2,3.3};
    int itab[3] = {4,5,6};

    // On fait l'hypothèse que itab et dtab ont la même taille, sinon il faudra réécrire autrement
    // la fonction template copier.

    copier (itab,dtab,3);
    affiche(itab,3);

    copier (dtab,itab,3);
    affiche(dtab,3);

    return 0;
}
```

## 7. Surdéfinition de patrons de fonctions

Si pour un type particulier le code est différent que pour les autres types => surdéfinir le patron.

```
// fichier min.h

#include <iostream>
#define H_MIN
#define H_MIN

template <class T>
T& min (T& a,T& b) {
    if (a < b) return a;
    return b;
}
#endif

// fichier min.cpp
#include "min.h"

int main() {

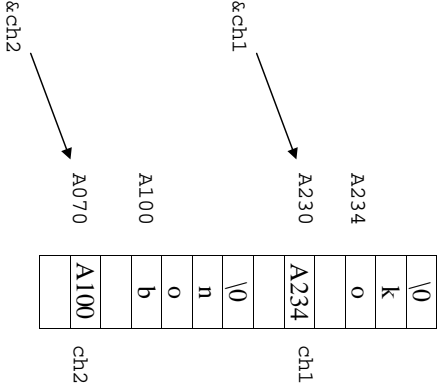
    int x=99, y=20;
    cout << min(x,y) << endl; // 20

    char c1='a',c2='b';
    cout << min(c1,c2) << endl; // a
    return 0;
}
```

Et si on traitait le cas:

```
char *ch1 = "ok";  
char *ch2 = "bon";
```

La fonction template min compare dans ce cas les **adresses** et non pas le contenu des adresses:



Dans cet exemple, nous comparons A100 à A234, nous aurons en sortie "bon" car la chaîne est stockée dans la zone basse de la mémoire.

Pour éviter que le résultat soit dépendant de la position de la chaîne dans la zone mémoire et non pas du contenu des adresses, nous devons définir une fonction particulière pour traiter le cas de `char*`:

```
char* min (char* a, char* b) {  
    if (strcmp(a,b) < 0) return a;  
    return b;  
}
```