Patrons
Templates

1. Généricité dans le cas de fonctions

- Modèle que le compilateur utilise pour créer des fonctions au besoin.
- Nous désirons écrire une fonction qui va permettre d'interchanger deux valeurs :

```
// Interchanger deux entiers
void echange(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}

// Interchanger deux réels
void echange(double& a, double& b){
    double temp = a;
    a = b;
    b = temp;
}

// Interchanger deux caractères
void echange(char& a, char& b){
    char temp = a;
    a = b;
    b = temp;
}
```

- 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 echange(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.).

- Cette fonction est vue comme la surdéfinition de fonctions, sauf que nous sommes 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 et au besoin ce qui entraîne moins de risques d'erreurs et permet d'éviter tous les types de pléonasmes.

2. Instanciation à partir d'une fonction générique

Type « int »

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

Les variables x et y sont de type « int ».

Le compilateur instancie « T » à « int » et génère la fonction
« void echange(int&,int&) ».

La fonction appelée est « echange(x,y) ».
```

Type « double »

```
double x=10.5, y=20.8;

Les variables x et y sont de type « double ».

Le compilateur instancie « T » à « double » et génère la
fonction « void echange(double&,double&) ».

La fonction appelée est « echange(x,y) ».
```

Type « char »

```
char x='a', y='b';

Les variables x et y sont de type « char ».

Le compilateur instancie « T » à « char » et génère la fonction
« void echange(char&,char&) ».

La fonction appelée est « echange(x,y) ».
```

Type « Compte »

```
Compte x(120.5), y(250.8);

Les variables x et y sont de type « Compte ».

Le compilateur instancie « T » à « Compte » et génère la
fonction « void echange(Compte&,Compte&) ».

La fonction appelée est « echange(x,y) ».
```

- Puisque la fonction « echange » 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.

3. Exemple d'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;
}

// Crée une version du tableau pour chaque type,
// et affiche ses valeurs en sortie
int main() {

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

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

    compte C1(345.68);
    compte C2(999.99);
    compte tab3[2]= {C1,C2};
    affiche(tab3,2);

    return 0;
}
```

4. Répartition des déclarations de patrons de fonctions entre des fichiers « .h » et « .cpp »

- Les modèles de fonctions doivent être déclarés dans le fichier « *.h »

```
// Fichier affiche.h

#include <iostream>

// ifndef/define/endif sont là pour empêcher
// de multiples inclusions du fichier d'en-tête.

#ifndef T_AFF
#define T_AFF

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>

#ifndef H_COMP
#define H_COMP

class compte {
    double solde;
public:
    compte(double m):solde(m) {}
    friend ostream& operator<<(ostream&,const compte&);
};
ostream& operator<<(ostream& out,const compte& 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 tab1[3] = {1.5,2.4,5.8};
    affiche(tab1,3);

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

    const 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 bidon(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 une fonction à 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 d'une recopie de tableau

```
// Fichier copier.h

#include <iostream>

#ifndef 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

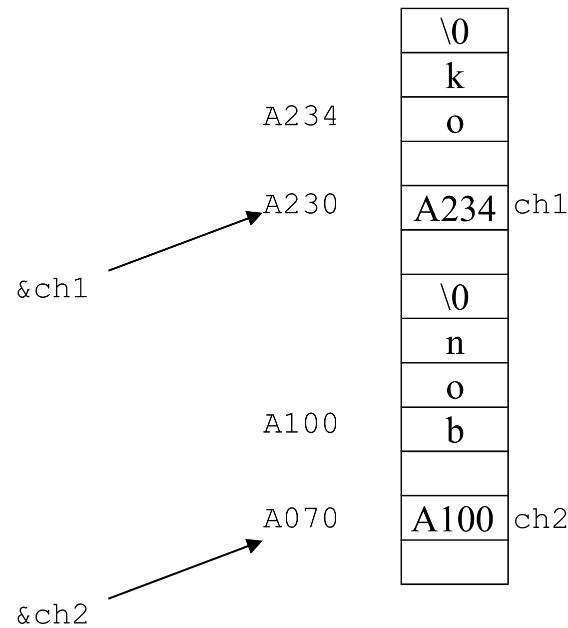
- Si pour un type particulier le code est différent par rapport aux autres types => il faudra surdéfinir le patron.

```
// fichier min.h
#include <iostream>
#ifndef 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 l'on traitait le cas :

```
const char *ch1 = "ok";
const 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 obtenu soit en fonction 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 « `char*` » :

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

8. Générécité pour les classes

- Modèle que le compilateur utilise pour créer des classes au besoin.
- Pour définir des classes génériques dont un ou plusieurs paramètres sont de type quelconque.
- Très utiles pour la réutilisation de classes "conteneurs" dont le rôle est de regrouper des objets suivant une certaine structure. Par exemple: liste, tableau, arbre, pile, etc.

9. Exemple de la classe « Point »

```
// fichier: 04_gclasse.h
#include <iostream>

#ifndef T_CLASSE
#define T_CLASSE

template <class T>
class point {
    T x,y;
public:
    point (T px=0, T py=0): x(px),y(py) {}
};

#endif
```

```
// fichier: 04_gclasse.cpp

#include "04_gclasse.h"
#include "01_compte.h"

int main() {

    point <int> p1(1,2);
    point <double> p2(4.5,7.8);
    point <int> p3(8,9);
    Compte C1(298.78);

    p1=p3; // OK deux int!

    p1 = p2; // erreur, car les deux classes sont différentes,
             // l'une gère des « int » l'autre des « double ».

    p1 = C1; // là aussi erreur, pour les mêmes raisons,
             // deux classes différentes.

    return 0;

}
```

10. Définition d’une méthode dans une classe patron

10.1. À l’intérieur de la classe

```
// Fichier: 05_gclasse.h

#include <iostream>

#ifndef T_CLASSE
#define T_CLASSE

template <class T>
class point {

    T x,y;

public:
    point (T px=0, T py=0): x(px),y(py) {}

    // Affiche est définie à l'intérieur de la classe
    // « template » « point ».

    void affiche() {
         cout << "x: " << x << " y: " << y << endl;
    }
};

#endif
```

10.2. À l’extérieur de la classe

```
// Fichier: 06_gclasse.h

#include <iostream>


#ifndef T_CLASSE
#define T_CLASSE


template <class T>
class point {
    T x,y;

public:
    point (T px=0, T py=0): x(px),y(py) {}

    void affiche();
};

// Doit être définie dans le *.h

template <class T>
void point<T>::affiche() {
    cout << "x: " << x << " y: " << y << endl;
}

#endif
```

10.3. Le cas des fonctions amies

```
// Fichier: 07_gclasse.h
#include <iostream>

#ifndef T_CLASSE
#define T_CLASSE

template <class T>
class point {
    T x,y;
public:
    point (T px=0, T py=0): x(px),y(py) {}

    // Sous g++ ne pas oublier <> sinon le compilateur
    // va considérer la fonction comme étant non-template.
    // Ce signe permet aussi de lever certains conflits,
    // et cela en spécifiant entre <> le nom de la classe
    // sur laquelle la fonction doit être appliquée.

    friend ostream& operator<<<>(ostream&,point<T>&);
};

template <class T>
ostream& operator<<(ostream& out,point<T>& p) {
    out << "x: " << p.x << " y: " << p.y << endl;
    return out;
}
#endif
```

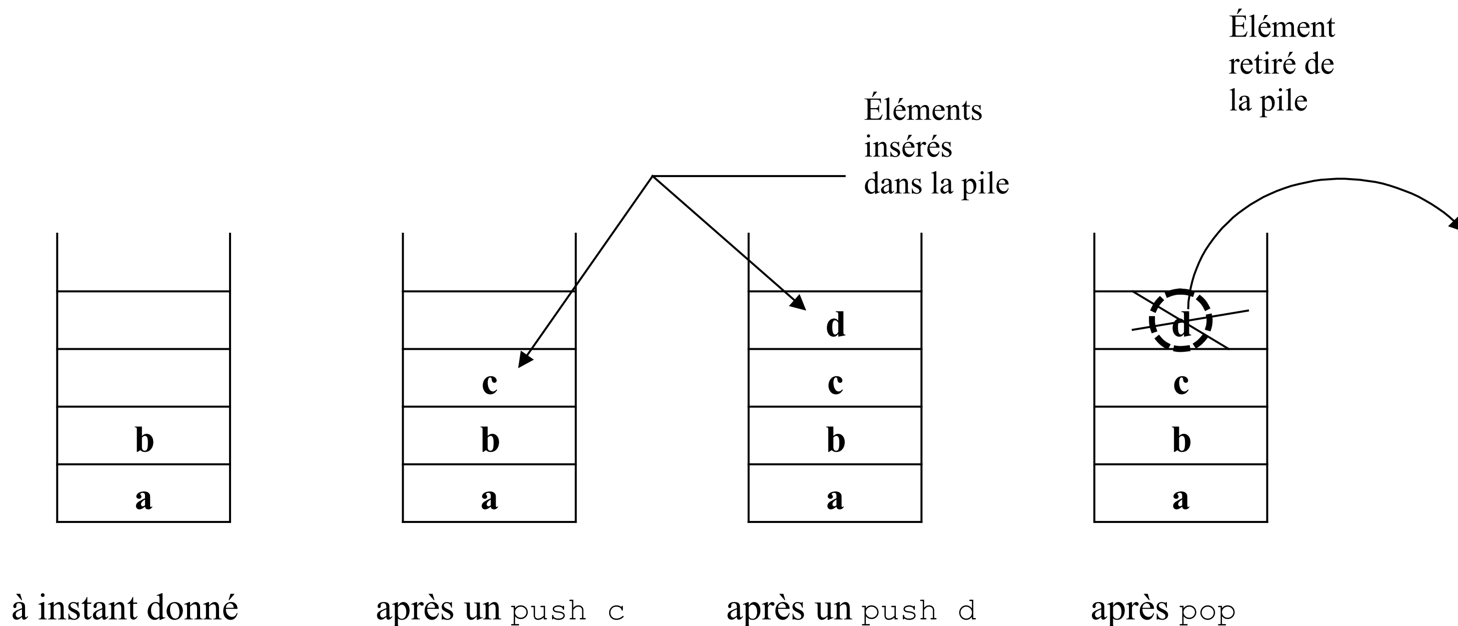
```
// Fichier: 07_gclasse.cpp

#include "07_gclasse.h"

int main() {

    point <int> p1(1,2);
    point <double> p2(4.5,7.8);
    point <int> p3(8,9);

    cout << p2;

    p1=p3;

    cout << p1;

    return 0;

}
```

11. Exemple complet : le cas d'une pile

11.1. Description de la pile

- La pile décrite dans cet exemple a les caractéristiques suivantes :

  - Une structure de données contenant des éléments du type `T`.
  - Elle gère les 2 opérations : empiler (« push ») et dépiler (« pop »)
  - Elle est du type « LIFO » (LastInFirstOut = Le dernier qui entre sera le premier à sortir).



11.2. Programme

```
// Fichier 08_pile.h

#include <iostream>
#include <string>

using namespace std;

#ifndef H_PILE
#define H_PILE

template< class T >
class Pile {

public:

    // Par défaut la pile va contenir 50 éléments.
    enum { DefautPile = 50, VidePile = -1 };

    // Constructeur vide, on va prendre la taille par défaut.
    Pile();

    // Constructeur à un argument, la taille de la pile.
    Pile( int );

    // Destructeur.
    ~Pile();
```

```
        // Fonction permettant d'insérer un élément dans la pile.

        void push( const T& );

        // Fonction permettant de retourner l'élément au somment
        // de la pile.

        T pop();

        // Test si la pile est vide.

        bool vide() const;

        // Test si la pile est pleine.

        bool pleine() const;

    private:

        // Les éléments de la pile du type
        // « T = int, double, char » etc.

        T* elements;

        // La taille réelle
        int sommet;

        // La taille max
        int taille;
```

```
        // Fonction d'allocation des éléments.

        void allocation() {
             elements = new T[ taille ];
             test_alloc();
             sommet = VidePile;
        }

        // Pour tester si l'allocation est OK.

        void test_alloc () {
             if (elements==NULL) {
               cerr << "problemes a l'allocation memoire \
                   de elements\n";
               exit(1);
             }
        }

        // Fonction d'affichage de messages.

        void msg( const char* m ) const {
             cout << "*** " << m << " ***" << endl;
        }

        // Surcharge de l'opérateur de sortie.

        friend ostream& operator<<<>( ostream&, const Pile< T >& );
    };
```

```
template< class T >
Pile< T >::Pile() {
    taille = DefautPile;
    allocation();
}

template< class T >
Pile< T >::Pile( int s ) {
    if ( s < 0 )        // taille négative?
         s *= -1;
    else if ( 0 == s ) // taille nulle?
         s = DefautPile;
    taille = s;
    allocation();
}

template< class T >
Pile< T >::~Pile() {
    delete[ ] elements;
}

template< class T >
void Pile< T >::push( const T& e ) {
    if ( !pleine() )
         elements[ ++sommet ] = e;
    else
         msg( "Pile pleine!" );
}
```

```
template< class T >
T Pile< T >::pop() {
    if ( !vide() )
         return elements[ sommet-- ];
    else {
         msg( "Pile vide!" );
         T valeur_qcq(5);
         // on retourne une valeur arbitraire!
         return valeur_qcq;
    }
}
template< class T >
bool Pile< T >::vide() const {
    return sommet <= VidePile;
}

template< class T >
bool Pile< T >::pleine() const {
    return sommet + 1 >= taille;
}
template< class T >
ostream& operator<<( ostream& os, const Pile< T >& s ) {
    s.msg( "Contenu de la Pile" );
    int t = s.sommet;
    while ( t > s.VidePile )
         cout << s.elements[ t-- ] << endl;
    return os;
}
#endif
```

```
// Fichier 08_pile.cpp

#include "08_pile.h"

int main() {

    Pile<int> pi;
    Pile<double> pd;
    Pile<char> pc;
    Pile<int> px;

    pi.push(1);
    pi.push(2);
    pd.push(120.67);
    pd.push(-23.56);

    cout << pi << endl;
    cout << pd << endl;
    cout << px << endl;
    cout << px.pop() << endl;

    for (int i=0; i< 10; i++) pc.push((char) ('A' + i));

    cout << pc << endl;

    return 0;

}
```

- La sortie obtenue après exécution du programme :

```
*** Contenu de la Pile ***
2
1

*** Contenu de la Pile ***
-23.56
120.67

*** Contenu de la Pile ***

*** Pile vide! ***
1073783752
*** Contenu de la Pile ***
J
I
H
G
F
E
D
C
B
A
```