

Chapitre 11

Patrons (Templates) de classe

1. Généralités

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. Pour le cas d'une pile, voir l'exemple à la fin de ce chapitre.

2. Exemple de la classe point

```
// fichier : g_classe.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: g_classe.cpp

#include "g_classe.h"
#include "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;

}
```

3. Définition d'une méthode dans une classe patron

3.1. À l'intérieur de la classe

```
// Fichier: g_classe.h

#include <iostream>

#ifdef 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
```

3.2. À l'extérieur de la classe

```
// Fichier: g_classe.h
#include <iostream>

#define 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
```

3.3. Le cas des fonctions amies

```
// Fichier: g_classe.h
#include <iostream>

#define 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& out,point<T>& p);
};

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

#endif
```

```
// Fichier : g_classe.cpp

#include "g_classe.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;
}
```

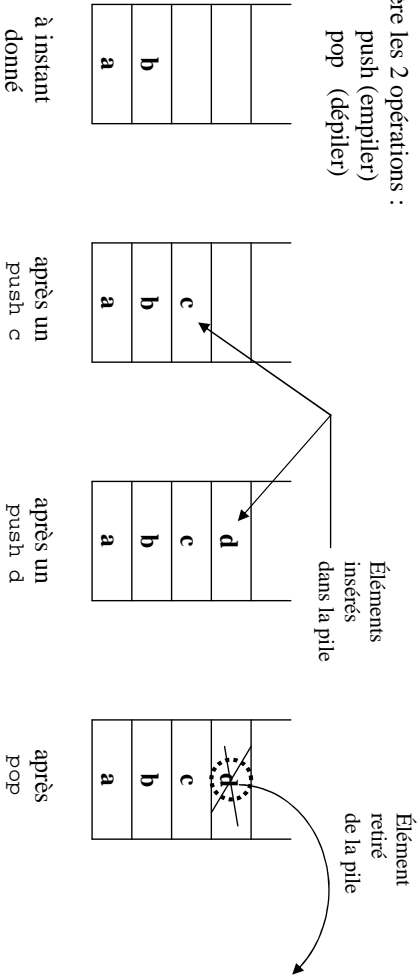
4. Exemple complet: pile

4.1. Description de la pile

La pile décrite dans cet exemple a les caractéristiques suivantes:

- une structure de données contenant des éléments du type T.

- gère les 2 opérations :
push (empiler)
pop (dépiler)



- du type LIFO (Last In First Out)

4.2. Programme

// Fichier **pile.h**

```
#include <iostream>
#include <string>

using namespace std;

#ifdef 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();
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre II : Patron (Templates) de classe

```
// Fonction permettant d'insérer un élément dans la pile.
void push( const T& );

// Fonction permettant de retourner l'élément au sommet 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;
    }
```

© Mohamed N. Lokbani

v3.00

POO avec C++

```
// 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();
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre II : Patron (Templates) de classe

212

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

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

template< class T >
T Pile< T >::pop() {
    if ( !vide() )
        return elements[ sommet-- ];
    else {
        msg( "Pile vide!" );
        T valeur_gcq;
        return valeur_gcq; // on retourne une valeur arbitraire!
    }
}

template< class T >
bool Pile< T >::vide() const {
    return sommet <= VidePile;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

```
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 pile.cpp

#include "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;

}
```

Sortie obtenue après exécution du programme 4.2.

```
*** 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

4.3. Exercice

Tout en justifiant votre réponse, le patron de la classe `pile` défini en 4.2. Traite-t-il correctement les types: `char*` et `compte`?

```
pile<char*> pch;  
pile<compte> ppte;
```

Dans le cas d'une réponse négative, complétez le programme 4.2.