

Classes et fonctions patron

Les templates

Les templates

- Les patrons (templates) permettent de créer des fonctions et des classes génériques et indépendantes d'un type particulier.
- Ce concept permet de développer un code hautement réutilisable tout en réduisant la quantité du code écrit.
- Les patrons permettent donc de créer des fonctions, des collections et des classes génériques en se focalisant sur les algorithmes plutôt que sur les types des données manipulées.
- Exemples:
 - ✓ Fonctions min, max, tri, recherche d'un élément, etc...
 - ✓ Les conteneurs liste, vecteur, pile, file, arbre, etc...

Les fonctions patrons

Exemple

- Imaginons le cas de la fonction *min* qui prend en entrée deux éléments pour retourner le minimum entre eux.
- Le programmeur devra surcharger cette fonction autant de fois que les types de données qui devront être comparés pour choisir le minimum.
- Tâche fastidieuse

```
if(entité1<entité2){return entité1}else{return entité2}
```

Solution


- Les fonctions patrons permettent d'écrire un sous programme (fonction) qui effectue une tâche bien définie indépendamment du type de données manipulé.
- Pas besoin de réécrire la fonction pour chaque type.
- Le compilateur se chargera de la génération du code nécessaire pour prendre en considération les types qui vont réellement être utilisés.

Les fonctions patrons

- La déclaration et la définition des fonctions *template* se fait comme une fonction classique sauf qu'elle doit être précédée de la déclaration des paramètres *template*.
- La syntaxe d'une déclaration de fonction *template* est donc la suivante :

```
template <class T1 [, class T2]>
type fonction(paramètres_fonction);
```

Si plusieurs types génériques
sont requis dans la définition



- Les fonctions *template* peuvent être surchargées, aussi bien par des fonctions classiques que par d'autres fonctions *template*.
- Une fonction *template* peut être déclarée amie d'une classe (*template* ou non).
- Toutes les instances générées à partir d'une fonction amie *template* sont amies de la classe donnant l'amitié.

Fonctions patrons

Syntaxe :

```
template <class T>  
T minimum(T t1, T t2) {  
    if (t1 < t2) {  
        return t1;  
    } else {  
        return t2;  
    }  
}
```

Utilisation:

- Indication implicite du type :

```
int n = minimum(1, 2);
```

- Indication explicite du type

```
int n = minimum<int>(1, 2);  
double n = minimum<double>(1, 2.5);
```

- T va être considéré comme un type générique
- T va être remplacé par le type adéquat lors de l'appel à la fonction.
- La fonction va comparer les deux paramètres en se basant sur l'opérateur < correspondant au type T.

Fonctions patrons

- La fonction minimum ainsi déclarée peut être utilisée sur n'importe quel type, pourvu que l'opérateur < soit bien défini pour celui-ci.
- On pourrait bien évidemment utiliser la fonction patron minimum pour déterminer le minimum entre deux objets d'une classe définie par l'utilisateur.

Syntaxe :

```
class Etudiant{
public:
    string nom;
    double moyenne;
    Etudiant(string n,double m){
        nom = n;
        moyenne = m;
    }
};

bool operator<(Etudiant e1,Etudiant e2){
    return e1.moyenne < e2.moyenne;
}
```

Utilisation:

```
Etudiant et1("mohamed",10.5);
Etudiant et2("salah",13.6);
Etudiant e = minimum(et1,et3);
```

Fonctions patrons

- Fonction template pour le tri d'un tableau?

Fonctions patrons

Fonction template pour le tri d'un tableau?

```
template <class T>
void tri_insertion(T* t, int n){
    int i, j;
    for (i = 1; i < n; i++) {
        /* Stockage de la valeur en i */
        T elementInsere = t[i];
        /* Décale les éléments situés avant t[i] vers la droite
           jusqu'à trouver la position d'insertion */
        for (j = i; j > 0 && t[j - 1] < elementInsere; j--) {
            t[j] = t[j - 1];
        }
        t[j] = elementInsere;
    }
}
```


Classes patrons

- Il y a un besoin récurrent pour les structures de données comme les tableaux, les listes ou les piles.
- Ces structures ainsi que les algorithmes qui travaillent dessus ont toujours le même code à l'exception des types de leurs contenus qui change.
- Les classes patrons/génériques/conteneurs permettent d'écrire un code générique indépendant des types de données manipulés.
- Les structures de données ainsi que les algorithmes sont écrits en utilisant des types formels/fictifs.
- Les types fictifs seront remplacés par ceux qui seront employés lors de l'utilisation de la classe.
- Le compilateur se charge de l'emploi des types adéquats.

Classes patrons

- La déclaration et la définition d'une classe *template* se font comme celles d'une fonction patron et doivent donc être précédées de la déclaration des types génériques.
- La déclaration suit donc la syntaxe suivante :

```
template <paramètres_template> class|struct|union nom;
```

- si les méthodes de la classe *template* ne sont pas implémentées dans la déclaration de la classe, elles devront être déclarées *template* :

```
template <paramètres_template>
```

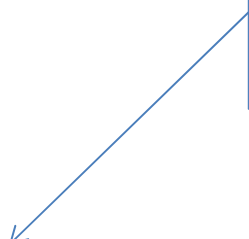
```
Type classe<paramètres>::nom(paramètres_méthode) { ... }
```

- Hormis les destructeurs, les méthodes d'une classe peuvent être *template*, que la classe soit patron ou non.
- Les fonctions membres *template* peuvent appartenir à une classe *template* ou à une classe normale.
- Lorsque la classe à laquelle elles appartiennent n'est pas *template*, la syntaxe des fonctions membres templates est exactement la même que pour les fonctions *template* non membre.

Classes patron

```
template <class TRUC>
class tableau{
    int tailleMax, nbElements;
    TRUC * tab ;
public :
    Tableau (int n = 100){
        tab= new TRUC [n] ;
        tailleMax = n ;
        nbElements = 0 ;
    }
    ~Tableau(){ delete [ ] tab; }
    bool inserer (TRUC &obj){
        if (nbElements == tailleMax)
            return false ;
        tab[nbElements] = obj;
        nbElements ++;
        return true;
    }
};
```

Déclaration des méthodes
membres à l'intérieur de la
classe



Classes patron

```
template <class TRUC>
class Tableau{
    int tailleMax, nbElements;
    TRUC * tab ;
public :
    Tableau (int n = 100){
        tab= new TRUC [n] ;
        tailleMax = n ;
        nbElements = 0 ;
    }
    bool inserer (TRUC &obj);
    ~Tableau(){ delete [ ] tab; }
};
```

```
template <class TRUC>
bool Tableau <TRUC>::inserer (TRUC &obj){
    if (nbElements == tailleMax)
        return false ;
    tab[nbElements] = obj;
    nbElements ++;
    return true;
}
```

Déclaration des méthodes
membres à l'extérieur de la
classe

