# Définir des méthodes et des classes génériques avec les templates.

Un **template**(**patron**) C++ est une fonctionnalité puissante ajoutée au C++ qui permet de définir des classes et fonctions génériques et fournit ainsi un support pour la programmation générique.

La programmation générique est une technique qui consiste à écrire du code d'une manière indépendante de tout type de données particulier.

Par exemple, cela permet de rendre le code de la classe Liste que nous avons défini pour l’utiliser avec différents type.

Un template est un plan ou une formule pour créer une classe générique ou une fonction. Les conteneurs de bibliothèque comme les itérateurs et les algorithmes (par exemple **STL**) sont des exemples de programmation générique et ont été développés en utilisant le concept de template.

C++ a deux nouveaux mots-clés pour prendre en charge les modèles : "template" et "typename". Le deuxième mot-clé peut toujours être remplacé par le mot-clé "class".

Les templates peuvent être utilisés de deux manières : les **fonctions template** et les **classes template**

1. **Fonction template**

Une fonction template fonctionne de manière similaire à une fonction normale, avec une différence clé. Une seule fonction template peut fonctionner avec différents types de données à la fois, mais une seule fonction normale ne peut fonctionner qu'avec un seul ensemble de types de données.

Normalement, si vous devez effectuer des opérations identiques sur deux ou plusieurs types de données, vous utilisez la surcharge de fonctions, donc créer deux fonctions avec les types de données requises.

Cependant, une meilleure approche serait d'utiliser des modèles de fonction car vous pouvez effectuer la même tâche en écrivant moins de code, efficace et maintenable.

Comment déclarer une fonction template ?

Une fonction template commence par le mot-clé "**template**" suivi du ou des paramètres du template à l'intérieur de **< >** qui est suivi de la déclaration de fonction.

* Syntaxe

|  |  |  |
| --- | --- | --- |
|  |  | template <class T> // ou template <typename T>  type\_retour nomFonction(T p1)  {     ... .. ...  } |

Dans le code ci-dessus, T est un type générique (type du template) qui accepte différents types de données (int, float) et typename est un mot-clé.

Vous pouvez également utiliser le mot-clé " **typename**" au lieu de "**class**" dans l'exemple ci-dessus.

Lorsqu'un argument d'un type de données est passé à **nomFonction()**, le compilateur génère une nouvelle version de **nomFonction()** pour le type de données donné.

Exercices / Exemples :

**Exemple 1 : Fonction template qui calcule la somme de deux nombres du même type.**

**Exemple 2 : Ecrire une foncion max, qui renvoie le plus grand élément entre deux nombres (int, double, …).**

**Exemple 3 : Ecrire une fonction qui prend en arguments deux objets de type différent pour les afficher en console.**

1. **Classe template**

Comme les fonction templates, les classes templates sont utiles lorsqu'une classe définit quelque chose qui est indépendant du type de données. Peut être utile pour des classes comme **LinkedList, BinaryTree, Stack, Queue, Array, etc**.

Comment déclarer une classe template ?

* Syntaxe

template <class T>

class nomClasse

{

    ... .. ...

    public:

        T attr;

        T nomfonction(T arg);

   ... .. ...

};

Dans la déclaration ci-dessus, T est l'argument du template (espace réservé) qui est un espace réservé pour le type de données utilisé. Comme la fonction template, nous pouvons déclarer de nombreux espaces réservés séparés par une virgule.

Comment créer un objet de la classe template ?

Pour créer un objet de la classe template, vous devez définir le type de données à l'intérieur du **< >** lors de la création.

* Exemples

|  |  |
| --- | --- |
|  | nomClasse<int> obj;  nomClasse<float> obj;  nomClasse<string> obj;    // pour une classe template de plusieurs espaces réservés (types de template)  nomClasse<float,int> obj;  nomClasse<string,float> obj; |

Exemples / Exercices

**Exemple 5 : Class Tab qui gère les éléments d'un tableau**

**Exemple 6 : Modifier le type Liste pour utiliser les templates.**

### Pouvons-nous spécifier une valeur par défaut pour les arguments du template?

Oui, comme les paramètres normaux, nous pouvons spécifier le type d'arguments par défaut template. L'exemple suivant spécifie la valeur par défaut "char" pour le deuxième argument du template.

* Exemple

|  |  |
| --- | --- |
|  | #include<iostream>  using namespace std;    template<class T, class U = char>  class A  {      public:          T x;          U y;  };    int main()  {     A<int> a;  // Cela appellera A<int, char>     return 0;  } |

* **Que se passe-t-il lorsqu'il y a un membre statique dans une classe/fonction modèle ?**

Chaque instance du template contient sa propre variable statique.

* **Peut-on passer des paramètres non typés aux templates ?**

Nous pouvons passer des arguments non typés aux templates. Les paramètres non typés sont principalement utilisés pour spécifier des valeurs max ou min ou toute autre valeur constante pour une instance particulière du template. La chose importante à noter à propos des paramètres non typés est qu'ils doivent être const. Le compilateur doit connaître la valeur des paramètres non typés au moment de la compilation. Parce que le compilateur doit créer des fonctions/classes pour une valeur non typée spécifiée au moment de la compilation. Dans le programme ci-dessous, si nous remplaçons 10000 ou 25 par une variable, nous obtenons une erreur de compilation.

* Exemple

|  |  |
| --- | --- |
|  | #include <iostream>  using namespace std;    // calculer minimum d'un tableau  template <class T, int max>  int mini(T tab[], int n)  {     int m = max;     for (int i = 0; i < n; i++)        if (tab[i] < m)           m = tab[i];       return m;  }    int main()  {     int t1[]  = {750, 30, 11, 18};     int n1 = sizeof(t1)/sizeof(t1[0]);// calculer le nombre d'éléments dans le tableau t1       char t2[] = {8, 15, 17,1,2};     int n2 = sizeof(t2)/sizeof(t2[0]); // calculer le nombre d'éléments dans le tableau t2       int maxi=1000;       // le deuxième paramètre est constant, il fonctionne donc parfaitement.     cout << mini<int, 10000>(t1, n1) << endl; |