# Chapitre 13 : Principe des Templates

Les **templates** (modèles) en C++ permettent d'écrire du **code générique**, c'est-à-dire du code indépendant du type de données utilisé.

## 1. Notion de programmation générique

La **programmation générique** consiste à écrire des fonctions ou classes **indépendantes du type des données**, afin qu'elles puissent fonctionner avec différents types (int, float, double, string, etc.).

Au lieu d'écrire :

int max(int a, int b);

double max(double a, double b);

float max(float a, float b);

On écrit une seule version générique.

### ◆ Pourquoi utiliser la programmation générique ?

✓ Éviter la duplication du code

✓ Améliorer la maintenabilité

✓ Augmenter la réutilisabilité

✓ Réduire les erreurs

### Principe

On utilise le mot-clé :

template <typename T>

ou

template <class T>

Les deux sont équivalents.

## 2. Fonctions de modèles (Function Templates)

Une fonction template est une fonction paramétrée par un type.

◆ **Syntaxe générale**

```
template <typename T>
type_retour nomFonction(parametres);
```

◆ **Exemple : Fonction max générique**

```
#include <iostream>
using namespace std;

template <typename T>
T maximum(T a, T b) {
   return (a > b) ? a : b;
}

int main() {
   cout << maximum(5, 10) << endl;        // int
   cout << maximum(3.4, 2.1) << endl;     // double
   cout << maximum('a', 'z') << endl;     // char
}
```

Le compilateur génère automatiquement la version adaptée selon le type utilisé.

**Déduction automatique du type**

```
maximum(5, 10);
```

Le compilateur déduit automatiquement que T = int.

On peut aussi préciser explicitement :

```
maximum<double>(5, 10);
```

**Templates avec plusieurs types**

```
template <typename T, typename U>
auto addition(T a, U b) {
   return a + b;
}
```

**3. Classe de modèles (Class Templates)**

**Définition**

Une **classe template** permet de créer une classe générique paramétrée par un type.

**Syntaxe générale**

```
template <typename T>
class NomClasse {
   // membres utilisant T
};
```

Exemple : Classe Box

```
#include <iostream>
using namespace std;

template <typename T>
class Box {
private:
   T valeur;

public:
   Box(T v) : valeur(v) {}

   T getValeur() {
      return valeur;
   }
};

int main() {
   Box<int> b1(10);
   Box<double> b2(3.14);

   cout << b1.getValeur() << endl;
   cout << b2.getValeur() << endl;
}
```

**Instanciation**

```
Box<int> objet;
Box<string> texte;
```

Le type doit être précisé lors de la création de l'objet.

**4. Spécialisation de templates**

On peut définir un comportement particulier pour un type spécifique.

Exemple :

```
template <>
class Box<char> {
public:
   void afficher() {

      cout << "Specialisation pour char" << endl;
   }
};
```

**5. Templates et surcharge**

Les templates peuvent coexister avec des fonctions normales :

```
int maximum(int a, int b) {
   cout << "Version normale" << endl;
   return (a > b) ? a : b;
}

template <typename T>
T maximum(T a, T b) {
   cout << "Version template" << endl;
   return (a > b) ? a : b;
}
```

Le compilateur choisit la version la plus adaptée.

**6. Avantages et Inconvénients**

**Avantages**

- Code réutilisable
- Flexibilité
- Génération automatique de code
- Utilisé massivement dans la STL (vector, list, map...)

**Inconvénients**

- Messages d'erreurs parfois complexes
- Augmentation possible du temps de compilation
- Code binaire plus volumineux

**7. Templates dans la STL**

Exemple avec vector :

#include <vector>

vector<int> v1;

vector<double> v2;

vector est une classe template :

template <class T, class Allocator = allocator<T>>

class vector;