

## 5. Programmation générique : les « templates »

### 5.1. Introduction

Jusqu'à présent, on a vu que les méthodes pouvaient être surchargées pour s'adapter à différents types de données, que les classes abstraites pouvaient servir de modèle pour leur classe dérivée. C'est ces différents concepts que nous allons approfondir maintenant avec l'idée de chercher à réduire l'écriture du code. Prenons l'exemple de la fonction suivante :

#### Fonction RechercherPlusPetit

*pluspetit.cpp*

```
int RechercherPlusPetit(int a, int b)
{
    int retour;
    if (a < b)
        retour = a;
    else
        retour = b;
    return retour;
}
```

L'objectif est de retrouver le plus petit des deux nombres entiers, on peut bien sûr la surcharger pour obtenir la même chose avec des réels, des caractères...

L'idée est de développer une fonction générique qui s'adapte au type qu'on lui applique sans réécrire de code.

### 5.2. Fonctions modèles ou « template »

Dans un premier temps, il est nécessaire de définir un type de **variable générique**. Il peut représenter n'importe quel autre type. Puis dans un deuxième temps de réaliser la fonction en utilisant ce type générique. La syntaxe pour déclarer le type générique utilise les signes inférieur et supérieur et n'est pas terminée par un point virgule.

#### Fonction RechercherPlusPetit

*fonctiontemplate.cpp*

```
template <typename T> // déclaration du type générique T
T RechercherPlusPetit(const T& a, const T& b)
{
    T retour;
    if (a < b)
        retour = a;
    else
        retour = b;
    return retour;
}
```

Le compilateur va générer automatiquement toutes les fonctions dont vous avez besoin à partir de leur utilisation. Lors de l'appel de la fonction, il faut juste préciser le type réel avec lequel elle doit travailler.

#### Programme principal

*fonctiontemplate.cpp*

```
int main()
{
    double densiteAl(2.7);
    double densiteCu(8.9);

    cout << RechercherPlusPetit<double>(densiteCu, densiteAl) << endl;

    int parking(-1);
    int terrasse(5);

    cout << RechercherPlusPetit<int>(parking, terrasse) << endl;

    return 0;
}
```

Le type à traiter est également mis entre les signes inférieur et supérieur après le nom de la fonction.

On peut également imaginer une fonction qui calcule la moyenne des éléments d'un tableau. Cette fonction possède deux paramètres, elle peut s'écrire de la manière suivante :

## Fonction CalculerMoyenne

fonctiontemplate.cpp

```
template <typename T>
T CalculerMoyenne(T tab[], int nbElements)
{
    T somme = 0;
    for (int indice = 0; indice < nbElements; indice++)
    {
        somme += tab[indice];
    }
    return somme / nbElements;
}
```

La fonction peut être appelée de la manière suivante :

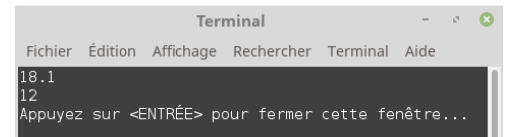
## programme principal

fonctiontemplate.cpp

```
int main()
{
    double temperature[5] = {12.5, 25.2, 14.4, 18.3, 20.1};
    int note[5] = {11, 10, 13, 15, 14};

    cout << CalculerMoyenne<double>(temperature, 5) << endl;
    cout << CalculerMoyenne<int>(note, 5) << endl;

    return 0;
}
```



On remarque que tout semble normal dans le premier cas, mais pour le second une question se pose : le résultat attendu doit-il être sous la forme d'un entier ou d'un réel ?

Cette première version a retourné un nombre du même type que celui fourni à la fonction.

Avec les fonctions modèles, il est tout à fait envisageable de préciser plusieurs types génériques et ainsi avoir un paramètre de retour différent par exemple :

## Fonction CalculerMoyenne

fonctiontemplate.cpp

```
template <typename T, typename R>
R CalculerMoyenneV2(T tab[], int nbElements)
{
    R somme=0;
    for (int indice = 0; indice<nbElements; indice++)
    {
        somme += tab[indice];
    }
    return somme/static_cast<int>(nbElements);
}
```

La ligne : `cout << CalculerMoyenneV2<int,double>(note,5) << endl;` dans le programme principal donne comme résultat cette fois-ci 12,6 au lieu de 12.

## À retenir

Les types génériques peuvent s'appliquer à n'importe quel type ou objet dès l'instant où l'opérateur utilisé dans la fonction a été surchargé. Par exemple l'opérateur `<` pour la fonction **RecherPlusPetit()** ou l'opérateur `+` dans la fonction **CalculerMoyenne()**.

## 5.3. Les classes templates

Les classes templates, dans le même ordre d'idée, sont des classes dont le type des arguments peut varier. Dans le module **Structure et gestion de données**, la structure de type Pile a pu être étudiée. Cette pile était typée à un seul objet, des entiers par exemple. Ce paragraphe va étendre la notion de Pile à toutes sortes d'objets sans pour autant multiplier le code.

Comme pour les fonctions, il faut définir le type générique, puis définir la classe classiquement en utilisant le type générique là où cela est nécessaire. L'exemple ci-dessous définit une pile de manière minimaliste.

### Classe Pile

*pile.h*

```
#ifndef PILE_H
#define PILE_H

template <typename T>
class Pile
{
private:
    int taille ;
    int sommet ;
    T *laPile;
public:
    Pile(const int _taille=10);
    void Empiler(const T element);
    T Depiler();
    bool PileVide();
};
```

Jusqu'ici rien de particulier, pour la suite, les méthodes doivent également être impérativement définies dans le fichier d'en-tête **pile.h** sinon le compilateur refuse de faire son travail.

### Implémentation des méthodes

*pile.h*

```
template<typename T>
Pile<T>::~Pile(const int _taille):
    taille(_taille),
    sommet(0)
{
    laPile = new T[taille];
}

template<typename T>
T Pile<T>::Depiler()
{
    T retour;
    if(!PileVide())
        retour = laPile[--sommet];
    return retour;
}

template<typename T>
Pile<T>::~~Pile()
{
    delete[] laPile;
}

template<typename T>
bool Pile<T>::PileVide()
{
    bool retour = false;
    if(sommet == 0)
        retour = true;
    return retour;
}

template<typename T>
void Pile<T>::Empiler(const T element)
{
    if(sommet < taille)
        laPile[sommet++] = element;
}

#endif // PILE_H
```

Le type générique doit être repris devant chaque méthode. Il faut également indiquer l'utilisation du type générique dans le nom de la classe puisqu'elle sera instanciée de manière différente pour chaque type.

### À retenir

Seule restriction, la déclaration de la classe et la définition des méthodes doivent se trouver dans le fichier d'en-tête, sinon il n'y a pas de compilation.

L'utilisation reste ensuite très classique, pour exemple, on fabrique une pile d'entiers et une pile de caractères. On empile 5 valeurs de chaque et on les dépile.

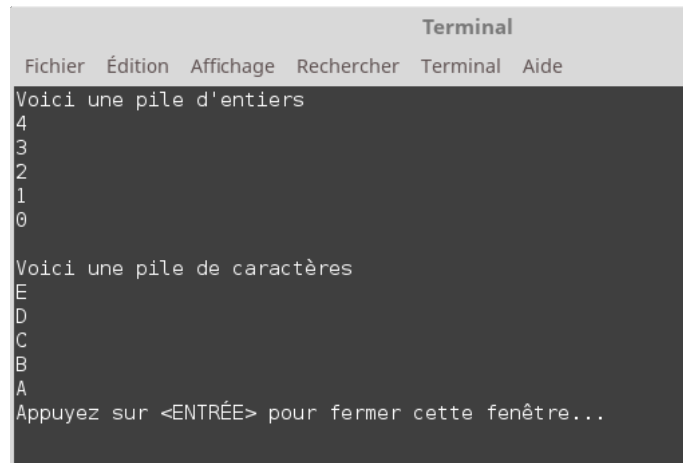
**Classe Pile***pile.cpp*

```
#include <iostream>
#include "pile.h"

using namespace std;

int main()
{
    cout << "Voici une pile d'entiers" << endl;
    Pile<int> pile1(5);          // on précise que c'est une pile d'entiers
    for (int indice =0;indice < 5; indice++)
    {
        pile1.Emplier(indice);
    }
    for (int indice=0; indice < 5;indice++)
    {
        cout << pile1.Depiler() << endl;
    }
    cout << endl << "Voici une pile de caractères" << endl;
    Pile<char> pile2(5);          // on précise que c'est une pile de caractères
    for (int indice =0;indice < 5; indice++)
    {
        pile2.Emplier('A'+ static_cast<char>(indice));
    }
    for (int indice=0; indice < 5;indice++)
    {
        cout << pile2.Depiler() << endl;
    }
    return 0;
}
```

Les éléments sont bien affichés dans l'ordre inverse dans lequel ils ont été introduit dans la pile.



```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Voici une pile d'entiers
4
3
2
1
0
Voici une pile de caractères
E
D
C
B
A
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

## 6. Programmation générique : Utilisation de la STL

### 6.1. Introduction

La librairie STL pour Standard Template Library, soit la librairie standard de modèle est une librairie normalisée pour le C++. Tous les identificateurs de la STL sont regroupés dans l'espace de nommage **std**. Cette librairie utilise massivement l'utilisation de types génériques développés lors du chapitre précédent.

Elle fournit un grand nombre d'éléments permettant une gestion efficace des structures de données complexes implémentées en C++, par exemple des tableaux, des piles, des files, des listes chaînées et des algorithmes optimisés pour les manipuler, recherche, tri, parcours, ajout ou encore suppression d'objets.

Plus particulièrement on y trouve trois familles d'éléments :

- **Les conteneurs** sont utilisés pour stocker des objets de même type. On parle donc de collection d'objets.
- **Les itérateurs** sont des objets qui permettent de naviguer parmi les éléments d'un conteneur. Un itérateur est un pointeur « intelligent ». L'itérateur fait le lien entre les conteneurs et les algorithmes
- **Les algorithmes** de la STL offrent des services fondamentaux sur les conteneurs. Ce sont des fonctions globales qui opèrent avec les itérateurs.

### 6.2. Les conteneurs

Les conteneurs sont des structures de données abstraites permettant de stocker des données de manière organisée. On en distingue deux catégories, les conteneurs séquentiels et les conteneurs associatifs. Ils diffèrent par la méthode d'accès aux données et à la façon dont la mémoire utilisée est organisée. Les premiers stockent les données de manière séquentielle, soit les une à la suite des autres. C'est le cas des vecteurs, des tableaux dynamiques, des piles, des files et des listes chaînées. Les seconds regroupent des ensembles ordonnés au sens mathématique ou des tables associatives ordonnées qui pourraient s'apparenter à des tableaux dont l'indexe ne serait pas forcément un entier, mais une chaîne de caractères par exemple, chaque donnée est associée à une clé permettant par la suite d'y accéder rapidement.

Conteneurs séquentiels		Conteneurs associatifs	
Les tableaux	Les listes	Les ensembles ordonnés	Ensemble indexé par une clé
<b>array</b> tableau statique contigu	<b>forward_list</b> liste simplement chaînée	N'accepte pas les doublons, chaque donnée est unique dans la collection	
<b>vector</b> tableau dynamique contigu	<b>list</b> liste doublement chaînée	<b>set</b> collection triée	<b>map</b> collection triée par une clé
Dérivés de vector		<b>unordered_set</b> collection non triée	<b>unordered_map</b> non triée avec clé
<b>deque</b> file d'attente à 2 bouts		Accepte les doublons	
<b>stack</b> pile (LIFO)		<b>multiset</b> collection triée	<b>mutimap</b> collection triée par une clé
<b>queue</b> file d'attente (FIFO)		<b>unordered_multiset</b> collection non triée	<b>unordered_multimap</b> non triée avec clé
<b>priority_queue</b> file d'attente avec priorité			

Le détail des différents conteneurs ne peut pas être passé en revue dans ce document. En cas de besoin, il est nécessaire de ce reporté à la documentation de référence : <https://en.cppreference.com/w/cpp/container> ou à sa version française, dans une traduction approximative : <https://fr.cppreference.com/w/cpp/container> réaliser avec Google traduction. D'autres exemple sont présentés dans le cours sur les strutures et la gestion de données.

Le choix du type de conteneur dépend fondamentalement de l'utilisation et des performances que l'on souhaite en avoir par exemple, les conteneurs de type **array**, **vector**, **deque**, **map** et **unordered\_map** permettent d'accéder rapidement à un élément avec l'opérateur **[ ]** contrairement aux autres. Un élément peut être inséré n'importe où pour une **list** sans que cela soit pénalisant au niveau temps, à la fin ou au début pour un **deque**, et uniquement à la fin pour un **vector**.

Utiliser un **vector**, est donc un bon choix lorsque l'on a besoin d'insérer/supprimer des éléments seulement à la fin, lorsque le conteneur doit être compatible au tableau C standard.

Utiliser une **list** est un bon choix lorsque l'on a besoin d'insérer/supprimer des éléments au milieu du conteneur et que le conteneur n'a pas besoin d'être compatible avec un tableau C standard ou que la taille maximum requise du conteneur n'est pas connue.

Le temps d'exécution pour les différentes opérations insertion, suppression, accès, recherche est fonction du type de conteneur et de sa structure. En fonction du lieu, la suppression ou l'insertion d'un élément dans un conteneur de type tableau peut demander la recopie complète de la collection alors que pour une liste c'est instantané. Par contre, l'accès à un élément sera instantané pour un tableau alors que pour une liste, elle doit être parcourue jusqu'à trouver l'élément.

#### Remarque

Les objets qui sont déposés dans un conteneur doivent répondre au modèle canonique dit **Coplien** et éventuellement avoir surchargés les opérateurs de tri comme inférieur < ou supérieur > et l'opérateur d'égalité ==.

Pour de nombreux conteneurs, un ensemble de méthode est défini :

<b>empty()</b>	Pour savoir si le conteneur est vide
<b>size()</b>	Pour déterminer le nombre d'éléments dans le conteneur
<b>erase()</b>	Pour supprimer un élément ou un ensemble d'éléments du conteneur
<b>clear()</b>	Pour supprimer tous les éléments

Pour déclarer un conteneur, comme pour chaque classe générique, il est nécessaire de préciser le type de données que contient le conteneur.

#### Exemple :

*conteneur.cpp*

```
#include <array>
#include <list>

class Rouleau;

using namespace std;

int main()
{
    // déclaration d'un tableau d'entiers avec 10 cases
    array<int, 10> tableau1;
    // déclaration d'un tableau de 3 caractères et son initialisation
    array<char, 3> tableau2 = {'a', 'b', 'c'};
    // déclaration d'une liste de Rouleaux, classe déclarée par ailleurs
    list<Rouleau> listeDeRouleaux;

    return 0;
}
```

## 6.3. Les itérateurs

Un itérateur est un objet permettant manipuler plus facilement les éléments d'un conteneur. Il permet de le parcourir, d'accéder aux données et éventuellement de les modifier. Il existe deux types d'itérateurs, ceux qui permettent d'effectuer un parcours du début à la fin et ceux qui permettent l'inverse.

<i>iterator</i>	Parcours du début à la fin	<i>const_iterator</i>	L'élément désigné ne peut pas être modifié
<i>reverse_iterator</i>	Parcours inverse	<i>Const_reverse_iterator</i>	

Pour être utilisé, un itérateur doit être initialisé. Les méthodes **begin()** et **end()** présentes dans la plupart des conteneurs permettent une initialisation au début ou à la fin de la collection. D'autres méthodes retournant un conteneur comme pour une recherche **find()** réalise également son initialisation.

Pour sa déclaration, l'itérateur fait référence au conteneur auquel il va être associé et au type de données qu'il reçoit. Voici un exemple d'utilisation d'un conteneur de type **vector** avec un itérateur.

**Exemple :***conteneur\_iterateur.cpp*

```
#include <vector>
#include <iostream>

using namespace std;

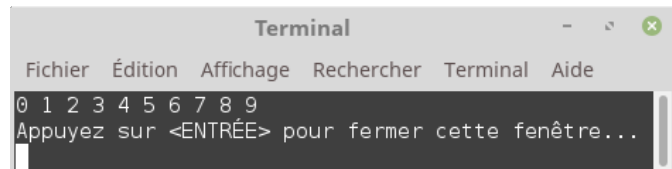
int main()
{
    //déclaration d'un tableau d'entiers dynamique avec 10 cases de réservées
    vector<int> tableau(10);

    for (int indice=0 ; indice < 10 ; indice++)
    {
        tableau[static_cast<size_t>(indice)] = indice;
    }

    //déclaration d'un itérateur sur le tableau
    vector<int>::iterator it;

    for (it=tableau.begin(); it != tableau.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```



L'itérateur s'utilise comme un pointeur. Il a été initialisé avec la méthode **begin()** de **vector**, le parcours se fait jusqu'à la fin du vecteur, sans se préoccuper des indices du tableau.

Remarque, l'avantage d'un tableau dynamique comme **vector** est qu'il peut s'agrandir ou se réduire en fonction des besoins.

Autre exemple avec un conteneur de type map :

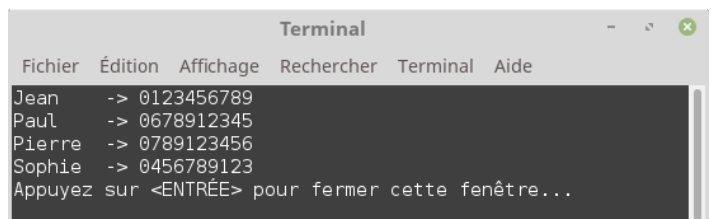
**Exemple :***conteneur\_iterateur.cpp*

```
#include <iostream>
#include <iomanip>
#include <map>

using namespace std ;

int
main ( void )
{
    map < string , string > telephones ;
    telephones ["Jean"] = "0123456789";
    telephones ["Paul"] = "0678912345";
    telephones ["Pierre"] = "0789123456";
    telephones ["Sophie"] = "0456789123";

    map < string , string >:: iterator it ;
    for ( it = telephones.begin() ; it != telephones.end() ; it ++ )
    {
        cout << setw(7) << left << it->first << " -> " << it->second << endl ;
    }
    return 0;
}
```



**it->first** représente la clé,  
**it->second** représente la valeur.

## 6.4. Les algorithmes de la librairie STL

La STL offre un ensemble d'algorithmes applicable aux différents conteneurs au travers des itérateurs. Parmi ces algorithmes, il existe plusieurs catégories :

<b>Algorithmes de base</b> <code>#include &lt;algorithm&gt;</code>	Recherche d'éléments : <code>min</code> , <code>max</code> , <code>find</code> , <code>search</code> ... Déplacement d'éléments : <code>swap</code> , <code>move</code> , <code>reverse</code> , <code>rotate</code> ... Tri : <code>sort</code>
<b>Algorithmes numériques</b> <code>#include &lt;numeric&gt;</code>	<code>accumulate</code> : effectue la somme des éléments
<b>Autres algorithmes de tri</b> <code>#include &lt;cstdlib&gt;</code>	<code>qsort</code> : Tri rapide, du plus petit au plus grand <code>qsort_r</code> : Tri rapide inverse

Les différents algorithmes sont présentés ici <https://fr.cppreference.com/w/cpp/algorithm> en français traduit approximativement par Google traduction, voici la version original : <https://en.cppreference.com/w/cpp/algorithm>.

### Exemple :

*conteneur\_tri.cpp*

```
#include <iostream>
#include <iomanip>
#include <array>
#include <algorithm>

using namespace std;

void AfficherValeur(int val);

int main()
{
    array<int, 10> tableau1 = {15,8,25,2,9,0,12,38,10,3};

    cout << "Tableau d'origine : ";
    for_each (tableau1.begin(), tableau1.end(),AfficherValeur);
    cout << endl;

    cout << "Tableau trié : ";
    sort(tableau1.begin(),tableau1.end());
    for_each (tableau1.begin(), tableau1.end(),AfficherValeur);
    cout << endl;

    return 0;
}

void AfficherValeur(int val)
{
    cout << setw(5) << val;
}
```

```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
Tableau d'origine : 15  8 25  2  9  0 12 38 10  3
Tableau trié      :  0  2  3  8  9 10 12 15 25 38
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Ce programme met en œuvre deux algorithmes de la librairie STL, **for\_each** qui permet d'appeler une fonction pour chaque élément du conteneur et **sort** qui permet de le trier dans l'ordre croissant.



## 7. Programmation générique : patrons de développement

La programmation orientée-objet en C++ a permis d'introduire les concepts de surcharge, d'héritage, et de polymorphisme. Ces principes permettent d'accroître l'adaptabilité du code :

- en redéfinissant une méthode avec des paramètres différents tout en gardant un fonctionnement similaire,
- en spécialisant le code pour chaque membre d'une famille d'objets et ainsi obtenir une programmation générique.

Le chapitre sur les templates a permis de montrer encore une extension de ces concepts en les généralisant à différents types de données.

Les patrons de développement ou **design patterns** étendent cette idée de ré-utilisabilité à la conception du logiciel. Ce mode de conception tend à rechercher des solutions standards qui pourraient s'appliquer à différents problèmes.

Lors de la phase de développement, les patrons servent de guide pour réaliser le code source du module en question.

Un patron regroupe un ensemble de composants apportant une solution à un problème. Ils sont regroupés par famille :

- Les créateurs : leur rôle est de définir les instanciations et la configuration des classes et des objets,
- Les structuraux : leur rôle est d'organiser les classes,
- Les comportementaux : ils décrivent la collaboration entre les objets, les algorithmes qui les composent et déterminent les responsabilités de chacun.

## 8. Conclusion

Le langage C++ est un langage polyvalent, réputé pour ses hautes performances, sa fiabilité, le faible encombrement du code produit et donc pour sa faible consommation d'énergie. Il permet de développer en bas niveau, au plus proche des composants ou en haut niveau, au niveau applicatif. C'est pourquoi on le retrouve dans le domaine de l'embarqué, du traitement d'image, des télécommunications, des jeux vidéo, des environnements graphiques, des applications scientifiques, dans l'écriture des systèmes d'exploitation... C'est dans les applications Web qu'il est peut-être le moins présent, même si côté serveur on peut le retrouver avec l'utilisation des applications utilisant les websocket par exemple.

Le langage C++ est l'un des langages les plus utilisés au monde. Il a la réputation d'être d'un abord plus complexe, mais ces performances et ces mécanismes de sûreté le rendent incontournable pour celui qui prend le temps de les appréhender. On retrouve sa syntaxe dans de nombreux autres langages. Certains d'entre eux sont même écrits en C++. Cela explique ces meilleures performances par rapport à ces autres langages. Connaître le C++ permet de s'adapter facilement à la programmation dans d'autres langages qui peuvent être dédiés à certains domaines.

C'est un langage en constante évolution, tout en gardant une compatibilité avec les versions antérieures, il est un gage de stabilité.

Le C++ est riche de nombreuses bibliothèques dans tous les domaines qui facilitent le développement d'application. Il peut-être complété par des frameworks tels que Qt qui le rend plus abordable et moderne.

Le C++ est le seul langage orienté objet qui est utilisé lors de l'épreuve écrite du BTS SN option Informatique et Réseaux.