

---

## TD 2

# Programmation procédurale et introduction à la *Standard Template Library* (STL)

---

### 2.1 Objectif

Ces exercices concernent la programmation procédurale en C++, au sens qu'ils ne requièrent du programmeur aucune *définition* de classe. Cependant on ne se privera pas d'utiliser des classes existantes, notamment certaines des classes prédéfinies de la STL qui représentent les structures de données fondamentales de la programmation.

Parmi les richesses de la STL trois (ensembles de) classes sont plus particulièrement précieuses car elles se substituent avantageusement à des constructions douteuses et pas très sûres de C :

- la classe `string` représente des chaînes de caractères; contrairement aux chaînes de C (de simples pointeurs `char *`, bornées par le caractère nul), la classe `string` gère automatiquement la mémoire et permet des manipulations globales naturelles et sûres (affectation, concaténation...).
- les classes de la bibliothèque `iostream` permettent également de rendre sûres les fonctions d'entrées-sorties de C, en particulier les tristement célèbres `printf` et surtout `scanf` ;
- la classe `vector` fournit un type de tableau mono-dimensionnel ; contrairement aux tableaux de C, les `vectors` ne se confondent pas avec les pointeurs, s'occupent eux-mêmes de leur allocation mémoire, peuvent grossir (ou maigrir) automatiquement et supportent des opérations globales comme l'affectation ou l'initialisation par un autre tableau.

Bien entendu la STL fournit de nombreux autres types représentant des collections homogènes d'objets (on parle de types « conteneurs », *containers*), comme des listes, des piles, des files, des tableaux associatifs (*maps*), des ensembles avec ou sans répétition, etc. Toutes ces collections peuvent être parcourues par un ensemble d'*itérateurs* présentant une interface commune quelle que soit la nature du conteneur. On peut aussi exécuter des opérations globales sur la collection grâce à un ensemble de fonctions génériques appelées *algorithmes*. Nous n'utiliserons pas ces possibilités dans cette série d'exercices.

## 2.2 Description rapide de quelques éléments de la STL

### 2.2.1 Entrées-sorties dirigées par les types (*iostream*)

Les fonctions de la bibliothèque `stdio` de C comme `printf` et `scanf` sont peu sûres : en effet le compilateur n'a, en général, aucun moyen de vérifier la cohérence entre les spécifications données dans le format et les types des paramètres effectifs fournis ; en particulier, dans le cas de `scanf`, il n'a même pas la possibilité de vérifier que ces paramètres sont bien des pointeurs.

C++ introduit une nouvelle bibliothèque, `iostream`, qui se substitue à `stdio`. En particulier deux nouvelles classes, `ostream` et `istream`, viennent remplacer les traditionnels `FILE *` de `stdio`. Les instances de `ostream` (resp. `istream`) correspondent à des flux d'écriture (resp. de lecture) ; comme dans `stdio`, ces flux sont tamponnés (*bufferisés*) dans l'espace utilisateur. Tout programme C++ s'exécute avec trois instances prédéfinies de ces classes : `cin` est une `istream` correspondant à l'entrée standard, `cout` et `cerr` sont des `ostreams` correspondant respectivement à la sortie et à l'erreur standard.

#### Utilisation des *ostreams*

C++ surcharge l'opérateur `<<` (décalage gauche) pour qu'il prenne une `ostream` comme premier opérande et un objet de n'importe quel type de base comme second opérande :

```
int i = 12;
double x = 3.14;
cout << "i = ";      // afficher une chaîne de C
cout << i;           // afficher la valeur de i
cout << '\n';        // afficher une fin de ligne
cout << "x = ";      // afficher une chaîne de C
cout << x;           // afficher la valeur du réel x
cout << endl;        // afficher une fin de ligne et vider le buffer
```

L'exécution des lignes précédentes produit ce qui suit sur la sortie standard (`cout`) :

```
i = 12
x = 3.14
```

Comme on le voit, le format de sortie est déterminé par le type de l'opérande droit. Noter aussi l'utilisation de la fonction spéciale `endl` qui provoque une fin de ligne et force le vidage du buffer interne (ce qui est différent de la simple sortie d'un caractère `'\n'`).

Il est même possible d'utiliser l'opérateur `<<` en cascade, de sorte que les lignes précédentes peuvent s'écrire sous la forme plus compacte

```
cout << "i = " << i << '\n' << "x = " << x << endl;
```

#### Utilisation des *istreams*

De manière similaire à `operator<<`, C++ définit `operator>>` (décalage droit) qui reçoit un premier argument de type `istream` et un second qui peut être une *variable* de n'importe quel type de base:

```
int i;
double x;
cin >> i >> x;
```

Ces instructions lisent (depuis l'entrée standard `cin`) un entier suivi d'un nombre réel. Pour

l'opérateur >>, les espaces<sup>1</sup> servent de séparateurs et sont pour le reste ignorés. Ainsi si on tape sur l'entrée standard

```
..13..5.75
```

(ou les `·` représentent des blancs) alors les instructions précédentes font que la valeur de `i` sera 13 et celle de `x` sera 5.75.

En fin de fichier<sup>2</sup>, `cin.eof()` est vrai ; en cas d'erreur (e.g., mauvais format) `cin.err()` est vrai. Comme toutes les `istreams`, `cin` est convertible implicitement en un booléen qui est vrai si ni `cin.eof()` ni `cin.err()` ne sont vrais. Ainsi on peut écrire

```
if (cin >> i >> j) ...    // si on a pu lire 2 entiers...

// lire un fichiers contenant des entiers
// jusqu'à la fin de fichier ou une erreur
while (cin >> i) ...      // tant que l'on peut lire un entier...
```

### 2.2.2 Chaînes de caractères : la classe `string`

La classe `string` propose une considérable amélioration sur les traditionnelles chaînes terminées par le caractère nul de C :

- la mémoire est gérée automatiquement,
- il n'y a pas besoin de s'occuper de la terminaison de la chaîne (le caractère nul),
- les opérations globales sur les `strings` sont possibles et naturelles (copie, comparaison, concaténation, etc.),
- les opérateurs << et >> sont définies et permettent d'afficher et de lire des `strings`,
- etc.

Bien sûr on peut aussi utiliser l'indexation (`operator[]`) sur les `strings`.

Voici un extrait de programme utilisant des `strings` :

```
string greeting;           // initialisation à la chaîne vide ""
string answer = "mumble, mumble";
cin >> greeting;
if (greeting == "hi,_there!") answer = "hi";
else if (greeting == "how_do_you_do?") answer = greeting;
answer[0] = toupper(answer[0]); // premier caractère en majuscule
cout << answer << endl;
unsigned int n = answer.size(); // longueur (utile) de la chaîne
```

Rappelez vous que quand on utilise l'opérateur >>, les espaces servent de séparateur mais sont autrement ignorés : il n'est donc possible de lire que des `strings` ne contenant pas d'espace (d'où les caractères `'_'` dans l'exemple ci-dessus). En sortie, cette limitation n'existe évidemment pas.

### 2.2.3 Tableaux dynamiques : la classe `vector`

Il s'agit d'une classe générique (*template*), c'est donc `vector<T>` où `T` est le type des éléments : un `vector<T>` est un tableau mono-dimensionnel de `T`. Contrairement aux tableaux de C, un vecteur connaît sa taille (fonction-membre `size()`) et on peut effectuer des opérations globales de copie. Bien sûr, on peut aussi appliquer l'opérateur d'indexation à un vecteur

1. Un *espace* ici doit être pris au sens « unixien » : cela peut être un blanc, un TAB, une fin de ligne, un saut de page, etc.

2. Rappelons que, sous Unix, on produit une fin de fichier au terminal en tapant `^D` seul sur une ligne

(operator[]).

Voici un extrait de programme utilisant des vecteurs :

```
vector<int> vi(10); // un vecteur d'entiers à 10 éléments.
                    // Attention : ce sont des parenthèses et pas
                    // des accolades ! Il s'agit d'un appel de
                    // constructeur ; tous les éléments sont
                    // initialisés à 0

vector<int> v;      // un vecteur vide (taille 0)
v = vi;            // copier vi dans v ; v a donc une taille de 10

// operator<< n'est pas défini pour les vector<T> !
// Donc une boucle est nécessaire
for (int i = 0; i < v.size; ++i) cout << v[i] << ' ';
cout << endl;      // fin de ligne et vidage du buffer
```

De plus les vecteurs peuvent grossir automatiquement sous l'effet de certaines opérations comme `push_back()` :

```
v.push_back(99); // maintenant v.size() == 11 et v[10] == 99
```

## 2.2.4 Une dernière remarque pour utiliser `iostream`, `string` et `vector`

Pour utiliser ces trois classes vous ne devez pas oublier d'inclure le fichier d'entête correspondant et d'utiliser une déclaration d'usage de l'espace de noms `std` :

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

## 2.3 Exercices

### 2.3.1 Palindrome

Un palindrome est, comme chacun sait, une chaîne de caractères qui se lit de la même manière dans les deux sens, comme "otto" ou "madam". Écrivez un programme qui lit des chaînes de caractères sur l'entrée standard (jusqu'à la fin de fichier), vérifie s'il s'agit d'un palindrome et imprime le résultat.

### 2.3.2 Utilisation de `vector`

Un *bookmaker* enregistre dans un fichier chacune de ses transactions sous forme d'une ligne contenant le nom du client et une somme positive si le client lui doit de l'argent ou une somme négative dans le cas contraire (voir le fichier `Strings_and_Vectors/records.txt` pour un exemple).

Écrivez un programme qui effectue les actions suivantes :

- lire chaque ligne d'un fichier comme `records.txt` et la ranger dans un `vector<Bookrecord>`, où `Bookrecord` est la structure suivante :

```
struct Bookecord {
    string _name;
    double _amount;
};
```

- afficher le contenu de ce vecteur sur la sortie standard ;
- entrer dans une boucle où l'on demande de fournir, sur l'entrée standard, un nom de client, où l'on consolide toutes les sommes concernant ce client (c'est-à-dire qu'on fait le total algébrique des sommes dues) et on affiche le résultat sur la sortie standard. ;
- terminer la boucle (et le programme) à la fin de fichier sur l'entrée standard.

**Suggestion** Vous aurez besoin d'ouvrir un fichier (comme `records.txt`) et de lui associer une `istream`. En fait, vous allez créer une `ifstream` (une classe dérivée de `istream`) de la manière suivante :

```
#include <fstream>
...
ifstream is("records.txt"); // constructeur : ouvre le fichier
```

Vous pouvez alors utiliser `is` comme une `istream` (et en particulier profiter de son opérateur `>>`).

### 2.3.3 Indexation et exceptions

En fait, les classes `string` et `vector` ont deux opérations d'indexation : l'opérateur usuel de C avec des crochets carrés (`v[i]`) et la fonction-membre `at()` (`v.at(i)`). Ces opérateurs retournent tous les deux une référence sur l'élément. Cependant `at()` vérifie les bornes de l'indice et lève une exception en cas d'échec alors que `operator[]` n'effectue aucune vérification. L'exception levée par `at()` s'appelle `out_of_range` et pour obtenir sa définition vous devez inclure le fichier d'entête `<stdexcept>`.

Vérifiez ce comportement des opérations d'indexation de `string` et `vector`. Vous pouvez utiliser les deux exercices précédents pour héberger vos expériences.

### 2.3.4 Passage de paramètres par référence ; fonctions génériques

Écrire une fonction `Swap()` (noter la majuscule<sup>1</sup>) qui échange les valeurs de deux variables entières :

```
int a = 1;
int b = 2;
Swap(a, b);
cout << a << ' ' << b << endl; // affiche 2 1
```

Pouvez vous imaginer comment rendre cette fonction générique, c'est-à-dire faire en sorte qu'elle échange les valeurs de ses deux paramètres quel que soit leur type, pourvu que ce soit le même ?

Essayer d'échanger des entiers, des doubles, des strings, des vectors... avec votre fonction générique.

---

1. C'est pour éviter de la confondre avec la fonction `swap()` prédéfinie dans la STL, et qui fait évidemment la même chose.

### 2.3.5 Définition de `operator<<` et `operator>>`

Modifiez le programme de l'exercice 2.3.2 de la manière suivante :

- définissez les opérateurs de décalage pour la structure `Bookrecord` de façon à pouvoir afficher et lire une telle structure d'un seul coup grâce aux `iostreams` ; les prototypes de ces fonctions doivent être les suivants :

```
ostream& operator<<(ostream&, const Bookrecord&);
istream& operator>>(istream&, Bookrecord&);
```

notez le passage par référence non constante pour `>>` (pourquoi ?) ; ces deux opérateurs doivent retourner leur premier paramètre.

- réécrivez le programme de 2.3.2 afin qu'il utilise ces deux opérateurs.

### 2.3.6 Allocation dynamique de mémoire avec l'opérateur `new`

Reprenez la classe `Fifo` de l'exercice 1.2 du 1. La taille maximale de la `Fifo` est un paramètre *template* `N`,

```
template <typename ELEM, int N> class Fifo;
```

Donc sa valeur doit être connue à la compilation.

Nous souhaiterions changer cela et déterminer la taille maximale de la `Fifo` à l'exécution. Nous retirons donc le paramètre *template* `N` et dotons notre classe d'un constructeur prenant un entier en paramètre :

```
template <typename ELEM>
class Fifo
{
public:

    Fifo(int n);
    // ... détails omis
};
```

Ainsi, quand nous créons une `Fifo`, nous devons maintenant donner sa taille maximale :

```
int nElems;
Fifo<double> aFifo(nElems);
```

Bien sûr le constructeur doit non seulement ranger cette taille quelque part, mais encore allouer *dynamiquement* (en utilisant l'opérateur `new`) le tableau interne `_tab`.

Modifiez votre classe `Fifo` pour qu'elle corresponde à ces nouvelles spécifications.

### 2.3.7 Opérateur `new` ou vecteur ?

Modifier la classe générique `Stack` du cours (son code figure dans le répertoire `Fifo` de TD1) en implémentant le tableau interne `_tab` à l'aide d'un vecteur (`vector`) et en utilisant `push_back()` (voir 2.2.3). Profitez en pour réécrire le `main()` de façon qu'il utilise les `iostreams`.