# Analyse et programmation orientée objet (C++)

Séance 6

STL et autres outils standards

# Objectifs

L’objectif du cours d’aujourd’hui est de vous présenter (sommairement) un certains nombre d’**outils** standards existant en C++

Le but ici n’est pas d’être exhaustif, mais simplement de vous :

- informer de l’existence des **principaux** outils
- faire prendre conscience d’aller **lire/chercher dans la documentation** les éléments qui peuvent vous être utiles

# Bibliothèque standard

La bibliothèque standard (d'outils) C++ **facilite la programmation** et permet de la rendre **plus efficace**, si tant est que l'on connaisse bien les outils qu'elle fournit.

Cette bibliothèque est cependant vaste et complexe, mais elle peut dans la plupart des cas s'utiliser de façon très simple, facilitant ainsi la réutilisation des structures de données abstraites et des algorithmes sophistiqués qu'elle contient.

La bibliothèque standard C++11 est formée de 79 « paquets » :

- 33 « classiques » (C++98)
- 20 nouveaux ( C++11 )
- les 26 bibliothèques C (C99)

# Contenu de la bibliothèque standard

La bibliothèque standard C++ contient 33 « paquets » de C++-98 :

| | |
|---|---|
| **<algorithm>** | plusieurs algorithmes utiles |
| <bitset> | gestions d'ensembles de bits |
| **<complex>** | les nombres complexes |
| <deque> | tableaux dynamiques avec push_front |
| <exception> | diverses fonctions aidant à la gestion des exceptions |
| **<fstream>** | manipulation de fichiers |
| <functional> | objets fonctions |
| **<iomanip>** | manipulation de l'état des flots |
| <ios> | définitions de base des flots |
| <iosfwd> | anticipation de certaines déclarations de flots |
| **<iostream>** | flots standards |
| <istream> | flots d'entrée |
| <iterator> | itérateurs |
| <limits> | diverses bornes concernant les types numériques |
| <list> | listes doublement chaînées |
| <locale> | contrôles liés au choix de la langue |

# Contenu de la bibliothèque standard (2)

| | |
|---|---|
| **<map>** | tables associatives clé–valeur ordonnées |
| <memory> | gestion mémoire pour les containers |
| <new> | gestion mémoire |
| <numeric> | fonctions numériques |
| <ostream> | flots de sortie |
| <queue> | files d'attente |
| **<set>** | ensembles ordonnés |
| **<sstream>** | flots dans des chaînes de caractères |
| **<stack>** | piles |
| <stdexcept> | gestion des exceptions |
| <streambuf> | flots avec tampon (buffer) |
| **<string>** | chaînes de caractères |
| <strstream> | flots dans des chaînes de caractère [en mémoire] |
| <typeinfo> | information sur les types |
| <utility> | divers utilitaires |
| <valarray> | tableaux orientés vers les valeurs |
| **<vector>** | tableaux dynamiques |

# Contenu de la bibliothèque standard (3)

La bibliothèque standard C++ contient 20 nouveaux « paquets » de C++11 :

| | |
|---|---|
| **<array>** | tableaux de taille fixe |
| <atomic> | expression atomique |
| <chrono> | heures et chronomètres |
| <codecvt> | conversions d'encodage de caractères |
| <condition_variable> | concurence (multi-thread) |
| **<forward_list>** | listes simplement chaînées |
| <future> | concurence (multi-thread) |
| **<initializer_list>** | listes d'initialisation |
| <mutex> | concurence (multi-thread) |
| **<random>** | nombres aléatoires |
| <ratio> | constantes rationnelles (Q) |
| <regex> | expressions régulières |
| <scoped_allocator> | allocation mémoire |
| <system_error> | erreurs système |
| <thread> | concurence (multi-thread) |
| <tuple> | *n*-uples |
| <type_traits> | caractéristiques de types |

## Contenu de la bibliothèque standard (4)

| | |
|---|---|
| <typeindex> | utiliser les types comme index de containers |
| **<unordered_map>** | tables associatives non ordonnées |
| **<unordered_set>** | ensembles non ordonnés |

Il existe aussi dans les outils standards les 26 « paquets» venant du langage C (C99) :

| | |
|---|---|
| <cassert> | test d'invariants lors de l'exécution |
| <ccomplex> | (inutile en C++) = <complex> |
| <cctype> | diverses informations sur les caractères |
| <cerrno> | code d'erreurs retournés dans la bibliothèque standard |
| <cfenv> | manipulation des règles de gestion des nombres en virgule flotante |
| <cfloat> | diverses informations sur la représentation des réels |
| <cinttypes> | int de taille fixée (C99) |
| <ciso646> | (inutile en C++) |
| <climits> | diverses informations sur la représentation entiers |
| <clocale> | adaptation à diverses langues |
| **<cmath>** | diverses définitions mathématiques |
| <csetjmp> | branchement non locaux |

# Contenu de la bibliothèque standard (5)

| | |
|---|---|
| <csignal> | contrôle des signaux (processus) |
| <cstdalign> | (inutile en C++) |
| <cstdarg> | nombre variables d'arguments |
| <cstdbool> | (inutile en C++) |
| <cstddef> | diverses définitions utiles (types et macros) |
| <cstdio> | entrées sorties de base |
| <cstdint> | sous-partie de cinttypes |
| <cstdlib> | diverses opérations de base utiles |
| <cstring> | manipulation des chaînes de caractères à la C |
| <ctgmath> | <cmath> + <complex> |
| <ctime> | diverses conversions de date et heures |
| <cuchar> | char de 16 ou 32 bits |
| <cwchar> | utilisation des caractères étendus |
| <cwctype> | classification des codes de caractères étendus |

# Outils standards

On distingue plusieurs types d'outils. Parmi les principaux :

- les containers de base
- les containers avancés (appelés aussi « adaptateurs »)
- les itérateurs
- les algorithmes
- les outils numériques
- Les traitements d'erreurs
- les chaînes de caractères
- les flots

# Outils standards (2)

Les outils les plus utilisés par les débutants sont :

- les chaînes de caractères (string) ✓
- les flots (stream) ✓
- les tableaux dynamiques (vector) [container] ✓
- les listes chaînées (list) [container avancé]
- les piles (stack) [container avancé]
- les algorithmes de tris (sort)
- les algorithmes de recherche (find)
- les itérateurs (iterators)

# String

- Les chaînes de caractères C++ sont définies par le type **string**. (classe)
- Pour utiliser des chaînes de caractères, il faut tout d'abord **importer la bibliothèque:**

```
#include <string>
```

- La déclaration d'une chaîne de caractères se fait alors avec :

```
string identificateur;
```

- Exemple :

```
#include <string>
// declaration
string str1;
// declaration avec initialisation
string str2("Bonjour tout le monde !");
```

# String

- Comme pour n'importe quel autre type, toute variable de type string (qui n'a pas été déclarée comme constante) peut être modifiée par une affectation.

- Exemple :

```
string chaine ; // -> chaine vaut ""
string chaine2("test") ; // -> chaine2 vaut "test"
chaine = "test3" ; // -> chaine vaut "test3"
chaine = chaine2 ; // -> chaine vaut "test"
chaine = 'a' ; // -> chaine vaut "a"
```

- ***Remarque 1 :*** dans le cas de l'affectation d'un caractère, la valeur affectée à la chaîne est la chaîne réduite au caractère affecté.
- ***Remarque 2 :*** Toute variable déclarée mais non initialisée est automatiquement initialisée à la valeur correspondant à la chaîne vide ("").

# String

- La **concaténation** de chaînes est représentée par l'opérateur +.
- chaine1 + chaine2 correspond à une **nouvelle chaîne** associée à la valeur littérale constituée de la concaténation des valeurs littérales de chaine1 et de chaine2.
- Les combinaisons suivantes sont possibles pour la concaténation de deux chaînes :

***string + string, string + "...", "..." + string, string + char, char + string***

où ***string*** correspond à une variable de type string, *"..."* correspond à une valeur littérale et ***char*** à une variable ou une valeur littérale de type char.

# String

- Exemples:

```
string nom;
string prenom;
string famille;
...
nom = famille + ' ' + prenom;
```

- Ajout d'un 's' final au pluriel :

```
string reponse("solution");
//...
if (n > 1) {
      reponse = reponse + 's';
}
```

# String

- Les **opérateurs relationnels** sont **également définis** pour les chaînes de type string (*ordre alphabétique*).

  **==** égalité

  **!=** non-égalité

  **<** strict. inférieur

  **<=** inférieur ou égal

  **>** strict. supérieur

  **>=** supérieur ou égal

# String

- Si `chaine` est une `string`, alors `chaine[i]` est le `(i+1)`ème caractère de chaine (de type `char`).

```
string demo("ABCD");
char prems;
char der;
prems = demo[0]; // recoit 'A'
der = demo[3]; // reçoit 'D'
```

# String

- Certaines fonctions *propres aux* string sont définies. Elle s'utilisent avec la syntaxe suivante :

```
nom_de_chaine.nom_de_fonction(arg1,arg2,...);
```

- **`chaine.size()`** : renvoie la taille (le nombre de caractères) de **`chaine`**.
- **`chaine.insert(position, chaine2)`** : insère, à partir de la position (indice) **`position`** dans la chaîne **`chaine`**, la string **`chaine2`**
- Exemple :

```
string exemple("abcd"); // exemple vaut "abcd"
exemple.insert(1,"xx"); // exemple vaut "axxbcd"
```

# String

- ***chaine.replace(position, n, chaine2)*** : remplace les ***n*** caractères d'indice ***position, position+1, ..., position+n-1*** de ***chaine*** par la string ***chaine2***.
- ***chaine.find(souschaine)*** : renvoie l'indice dans ***chaine*** du 1er caractère de l'occurrence ***la plus à gauche*** de la string ***souschaine***.
- ***chaine.rfind(souschaine)*** : renvoie l'indice dans ***chaine*** du 1er caractère de l'occurrence ***la plus à droite*** de la string ***souschaine***.
- ***chaine.substr(depart, longueur)*** : renvoie la chaîne de ***chaine***, de longueur ***longueur*** et commençant à la position ***depart***.
- Dans les cas où les fonctions ***find()*** et ***rfind()*** ne peuvent s'appliquer, elles renvoient la valeur prédéfinie ***string::npos***

# Containers

Comme le nom l'indique, les containers sont des structures de données abstraites (SDA) servant à **contenir** (« collectionner ») **d'autres objets**.

Vous en connaissez déjà plusieurs : les tableaux, les piles et les listes chaînées.

Il en existe plusieurs autres, parmi lesquels, les files d'attentes (queue), les ensembles (set, unordered_set) et les tables associatives (map, unordered_map).

# Containers (2)

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier... comme dans une file d'attente à un guichet !

(alors que dans une pile « normale », c'est toujours le dernier arrivé qui est dépilé en premier)

Les set permettent de gérer des **ensembles** (finis !) au sens mathématique du terme : collection d'éléments où chaque élément n'est présent qu'une seule fois.

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

# Containers (3)

Tous les containers contiennent les méthodes suivantes :

bool empty() : le containers est-il vide ?

unsigned int size() : nombre d'éléments contenus dans le container

void clear() : vide le container

iterator erase(*it*) : supprime du container l'élément pointé par *it*. *it* est un itérateur (généralisation de la notion de pointeur, voir quelques transparents plus loin)

Ils possèdent également tous les méthodes begin() et end() que nous verrons avec les itérateurs.

Passons maintenant à quelques containers particuliers

# Vector

- Nous avons jusqu'ici vu les tableaux de taille fixe (i.e. connue à l'avance).
- Que faire si la taille n'est pas connue ?
- Il existe en C++ ce que l'on appelle des **tableaux dynamiques**, c'est-à-dire dont la taille peut changer au cours du déroulement du programme, par exemple lorsqu'on ajoute ou retire des éléments dans le tableau.
- Les tableaux dynamiques sont définis en C++ par le biais du type vector.
- Pour les utiliser, il faut tout d'abord importer les définitions associées à l'aide d'un include :

```
#include <vector>
```

# Vector

- Une variable correspondant à un tableau dynamique se déclare de la façon suivante :

```
vector<type> identificateur;
```

où *identificateur* est le nom du tableau et *type* correspond au type des éléments du tableau.

- Il peut être simple ou composé (on peut donc faire des vector de vector !).

- Exemple :

```
#include <vector>
//...
vector<int> tableau;
```

# Vector

- Une taille initiale peut être indiquée si nécessaire.
- La syntaxe de la déclaration/initialisation est alors :

```
vector<type> identificateur(taille);
```

- Exemple : `vector<int> tab(5) ;` correspond à la déclaration d'un tableau initialement constitué de 5 entiers (tous nuls).

  **Attention** ! ce n'est pas la même chose que :

```
int tab[5]; // tableau de taille fixe a la C
```

# Vector

- La déclaration d'un tableau avec taille initiale peut en plus être associée à une initialisation explicite des éléments contenus dans le tableau, mais tous à la *même* valeur. Cela s'écrit :

```
vector<type> identificateur(taille, valeur);
```

  où *valeur* est la *même* valeur initiale affectée à tous les éléments du tableau

- On peut aussi initialiser un tableau dynamique à l'aide d'une copie d'un autre tableau dynamique:

```
vector<type> identificateur(reference);
```

  où ***reference*** est une référence à un tableau de même type de base *type*.

- Exemples :

```
vector<int> tab1(5,1);
vector<int> tab2(tab1);
```

correspondent toutes deux à la déclaration d'un tableau d'entiers dont les 5 éléments de départ sont initialisés à la valeur 1.

# Vector

- L'**indexation** des différents éléments d'un tableau dynamique (c'est-à-dire l'accès direct aux éléments du tableau) se fait de la même façon que pour un tableau de taille fixe :
- Si ***tableau*** est un ***vector***, alors ***tableau[i]*** est une référence au (i+1)ème élément de tableau.

**Attention !** Il est impératif que cet éléments **existe** effectivement ! Sinon risque de *Segmentation Fault !*

- Exemple (à ne **pas** suivre !) d'erreur classique :

```
vector<double> v;
v[0] = 5.4; // Erreur !! : v[0]
                 n'existe pas encore
```

# Vector

- Un certain nombre d'opérations sont **directement attachées** au type vector.
- L'utilisation de ces opérations spécifiques se fait avec la syntaxe suivante :

```
nom_de_tableau.nom_de_fonction(arg1, arg2,...);
```

- Exemple :

```
vector<double> tab;
// ...
nombre = tab.size();
```

# Vector

- Quelques fonctions disponibles pour un tableau dynamique `tableau` de type `vector<type>` :
- `tableau.size()` : renvoie la taille de tableau (type de retour : `size_t`)
- `tableau.front()` : renvoie une référence au 1er élément
- `tableau.front()` est donc équivalent à `tableau[0]`
- `tableau.back()` : renvoie une référence au dernier élément.
- `tableau.back()` est donc équivalent à `tableau[tableau.size()-1]`
- `tableau.empty()` : détermine si tableau est vide ou non (`bool`).
- `tableau.clear()` : supprime tous les éléments de tableau (tableau vide). Pas de retour.
- `tableau.pop_back()` : supprime le dernier élément de tableau. Pas de retour.
- `tableau.push_back(valeur)` : ajoute un nouvel élément de valeur `valeur` à la fin de tableau. Pas de retour.

# Vector

## Exemple 1 : les vector

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

int main() {
  vector<int> v1; // un vecteur d'entier vide

  v1.push_back( 10 ); // ajoute l'entier 10 à la fin
  v1.push_back( 9 ); // ajoute l'entier 9 à la fin
  v1.push_back( 8 ); // ajoute l'entier 8 à la fin
  // enleve le dernier élément
  v1.pop_back(); // supprime l'entier 8

  // utilisation d'un indice pour parcourir le vecteur v1
  cout << "Le vecteur v1 contient " << v1.size() << " entiers : \n";
  for(int i=0;i<v1.size();i++)
      cout << "v1[" << i << "] = " << v1[i] << '\n';
  cout << '\n';

   return 0;
}
```

# Vector

Itérer sur un tableau :

- avec une *itération for* classique :

```
for (size_t i(0); i < v.size(); ++i)
```

- avec les *itérations sur ensemble* de valeurs:
  - Si l'on ne veut pas modifier les éléments du tableau :

```
for (type nom_de_variable : tableau)
```

  - Si l'on veut modifier les éléments du tableau :

```
for (type & nom_de_variable : tableau)
```

où *type* est le type des éléments contenus dans le tableau.

- avec des *"itérateurs"*

# Vector

- Exemples : *itérations sur ensemble*

```
vector<double> salaires(10);
for(double & salaire : salaires) {
    cout << "Salaire de l'employe suivant ? ";
    cin >> salaire ;
}
cout << " Salaires des employes : " << endl;
for(double salaire : salaires ) {
    cout << " " << salaire << endl;
}
```

# Vector : itérateur

- Les **itérateurs** sont une SDA **généralisant** d'une part des accès par ***index*** (SDA séquentielles) et d'autre part les ***pointeurs***, dans le cas de **containers**.

- Ils permettent :
  - de parcourir de façon ***itérative*** les containers
  - d'indiquer (i.e. de pointer sur) un élément d'un container

# Vector : itérateur

- Un itérateur associé à un container ***C<type>*** se déclare simplement comme ***C<type>::iterator nom;***
- Exemples :

```
vector<double>::iterator i;
```

- Il peut s'initialiser grâce aux méthodes ***begin()*** ou ***end()*** du container, voire d'autres méthodes spécifiques, comme par exemple ***find*** pour les containers non-séquentiels.
- Exemples :

```
vector<double>::iterator i(vect.begin());
```

- L'élément indiqué par l'itérateur ***i*** est simplement ****i***.

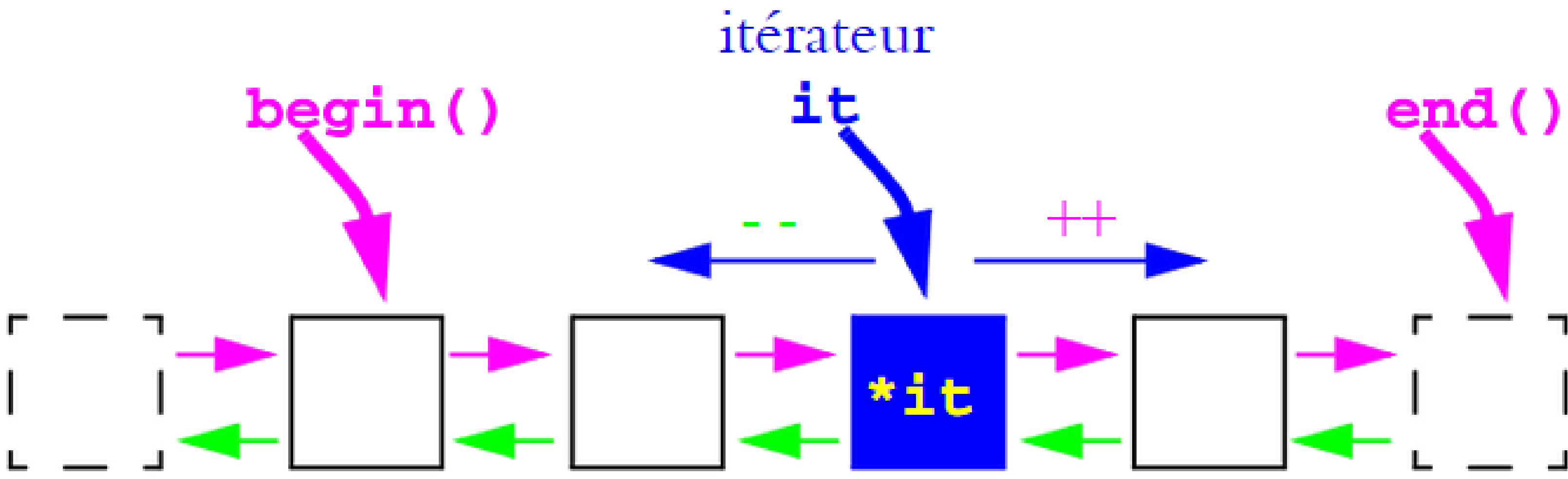
# Vector: itérateur

- Exemples : *itérateurs*

```
vector<double> salaires(10);
    for(vector<double>::iterator i = salaires.begin();
        i != salaires.end(); i++) {

        cout << *i << " " << endl;
    }
```

# Vector: itérateur



# Vector

- Une fonction peut renvoyer un vecteur.
- Voici une fonction qui recueille toutes les valeurs qui se situent dans un certain intervalle.

```
vector<double> between(vector<double> v, double low,
double high)
{
    vector<double> result;
    for (int i = 0; i < v.size(); i++)
        if (low <= v[i] && v[i] <= high)
            result.push_back(v[i]);
    return result;
}
```

# Vector : tableaux multidimensionnels

- Le type de base d'un tableau peut être n'importe quel type, y compris composé. En particulier, le type de base d'un tableau peut être lui même un tableau.
- Les tableaux correspondants sont alors des tableaux de tableaux, c-a-d des **tableaux multidimensionnels**.

- Exemple :

```
vector< vector <int> > tab(5, vector<int>(6));
```

correspond à la déclaration d'un tableau de 5 tableaux de 6 entiers, autrement dit un tableau bidimensionnel à 5 lignes et 6 colonnes.

- Comme pour les tableaux multidimensionnels de taille fixe, ***tab[i][j]*** permet alors de référencer l'élément de la ***(i+1)ème ligne*** et de la ***(j+1)ème colonne***.

# Vector

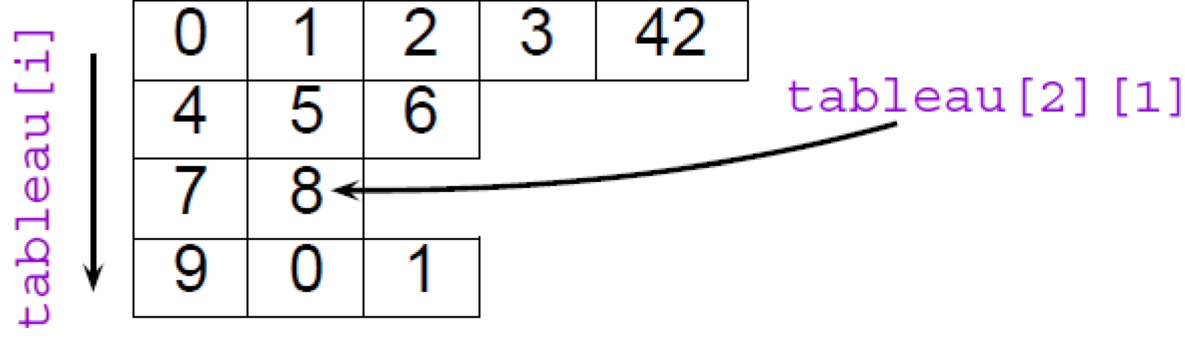
- Notez qu'un ***vector<vector<int>> tab*** n'est pas la même chose qu'un ***int tab[][]*** : dans le premiers cas les lignes de la "***matrice***" n'ont pas forcément toutes la même longueur alors que c'est le cas pour les tableaux de taille fixe.

```
vector<vector<int> >
```

- n'est pas une matrice, mais un vecteur de vecteurs d'entiers (pas nécessairement tous de la même taille !).

```
vector< vector<int> > tableau;
```

```
tableau.size() renvoie 4
tableau[0].size() renvoie 5
tableau[2].size() renvoie 2
```



# Liste (doublement) chaînées

Les listes (doublement) chaînées sont, comme les tableaux dynamiques, des SDA **séquentielles**, c'est-à-dire stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes chaînées sont définies dans la bibliothèque list et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
list<int> maliste;
```

(quelques) méthodes des listes chaînées :

| | |
|---|---|
| *Type*& front() | retourne le premier élément de la liste |
| *Type*& back() | retourne le dernier élément de la liste |
| void push_front(*Type*) | ajoute un élément en tête de liste |
| void push_back(*Type*) | ajoute un élément en queue de liste |
| void pop_front() | supprime le premier élément |
| void pop_back() | supprime le dernier élément |
| void insert(*iterator*, *Type*) | insertion avant un élément de la liste désigné par un itérateur |

# C++11 Liste chaînées

Les listes chaînées sont, comme les tableaux dynamiques, des SDA **séquentielles**, c'est-à-dire stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes simplement chaînées sont définies dans la bibliothèque forward_list et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
forward_list<int> maliste;
```

(quelques) méthodes des listes chaînées :

| | |
|---|---|
| *Type*& front() | retourne le premier élément de la liste |
| void push_front(*Type*) | ajoute un élément en tête de liste |
| void pop_front() | supprime le premier élément |
| void insert(*iterator*, *Type*) | insertion avant un élément de la liste désigné par un itérateur |

# Tableaux dynamiques : petit complément

Pour accéder directement à un élément d'un tableau dynamique (vector) on utilise l'opérateur [] : tab[i].

Il existe une autre méthode pour cet accès : at(*n*) qui, à la différence de [*n*], lance l'exception out_of_range (de la bibliothèque <stdexcept>) si n n'est pas un index correct.

Exemple :

```
#include <vector>
#include <stdexcept>
...
vector<int> v(5,3); // 3, 3, 3, 3, 3
int n(12);
try {
        cout << v.at(n) << endl;
}
catch (out_of_range) {
        cerr << "Erreur : " << n << " n'est pas correct pour v"
            << endl
            << "qui ne contient que " << v.size()
                << " elements." << endl;
}
```

# Ensembles – Exemple

Les ensembles (au sens mathématique) sont implémentés dans la bibliothèque <set>. Ils ne peuvent cependant contenir que des éléments du même type, lesquels sont ordonnés par operator<.

(Pour des éléments de même type mais non ordonnés, i.e. *sans* operator<, on utilisera un unordered_set.)

On déclare un ensemble comme les autres containers, en spécifiant le type de ses éléments, par exemple :

```
set<char> monensemble;
```

Les ensembles n'étant pas des SDA séquentielles, l'accès direct à un élément n'est pas possible.

(quelques) méthodes des ensembles :

| | |
|---|---|
| insert(*Type*) | insère un élément s'il n'y est pas déjà |
| erase(*Type*) | supprime l'élément (s'il y est) |
| find(*Type*) | retourne un itérateur indiquant l'élément a recherché |

À noter que la bibliothèque <algorithm> fournit des fonctions pour faire la réunion, l'intersection et la différence d'ensembles.

# Ensembles – Exemple

```
#include <set>

...

set<char> voyelles;
voyelles.insert('a');
voyelles.insert('b');
voyelles.insert('e');
voyelles.insert('i');
voyelles.erase('b');
voyelles.insert('e'); /* n'insere pas 'e' car *
                       * il y est deja */
```

Comment parcourir cet ensemble ?

```
for (unsigned int i(0); i < voyelles.size(); ++i)
            cout << voyelles[i] << endl;
```

ne fonctionne pas car c'est une SDA non-indexé (et même non-séquentielle).

# Ensembles – parcours

Comment parcourir cet ensemble ?

En C++11 c'est facile :

```
for (auto const v : voyelles)
        cout << v << endl;
```

Il y a aussi un autre moyen, plus avancé :

☞ utilisation d'**itérateurs**

# Itérateurs

Les **itérateurs** sont une SDA **généralisant** d'une part des accès par index (SDA séquentielles) et d'autre part les pointeurs, dans le cas de **containers**.

Ils permettent :

- de parcourir de façon itérative les containers
- d'indiquer (i.e. de pointer sur) un élément d'un container

Il existe en fait 7 sortes d'itérateurs, mais nous ne parlons ici que de la plus générale, qui permet de tout faire : lecture et écriture du containers, aller en avant ou en arrière (accès quelconque en fait).

# Itérateurs (2)

Un itérateur associé à un container *C*<*type*> se déclare simplement comme
*C*<*type*>::iterator *nom*;

Exemples :

```
vector<double>::iterator i;

set<char>::iterator j;
```

Il peut s'initialiser grâce aux méthodes begin() ou end() du container, voire d'autres méthodes spécifiques, comme par exemple find pour les containers non-séquentiels.

Exemples :

```
vector<double>::iterator i(monvect.begin());

set<char>::iterator j(monset.find(monelement));
```

L'élément indiqué par l'itérateur i est simplement *i, comme pour les pointeurs.

# Retour sur l'exemple des ensembles

Pour parcourir notre ensemble précédent, nous devons donc faire :

```
for (set<char>::iterator i(voyelles.begin());
        i != voyelles.end(); ++i)
    cout << *i << endl;
```

Exemple d'utilisation de find :

```
set<char>::iterator i(voyelles.find('c'));

if (i == voyelles.end())
    cout << *i << "n'est pas dans l'ensemble" << endl;
else
    cout << *i << "est dans l'ensemble" << endl;
```

## Code complet de l'exemple

```
#include <set>
#include <iterator>
#include <iostream>
using namespace std;
int main() {
    set<char> voyelles;
    voyelles.insert('a');
    voyelles.insert('b');
    voyelles.insert('e');
    voyelles.insert('i');
    voyelles.insert('a'); // ne fait rien car 'a' y est deja
    voyelles.erase('b'); // supprime 'b'
// parcours l'ensemble
for (set<char>::iterator i(voyelles.begin()); i!=voyelles.end(); ++i)
    cout << *i << endl;
// recherche d'un element
set<char>::iterator element(voyelles.find('c'));
if (element == voyelles.end())
    cout << "l'element n'est pas dans l'ensemble" << endl;
else
    cout << *element << " est dans l'ensemble" << endl;
return 0;
}
```

# Suppression d'un élément d'un container

On a vu que tout container possédait une méthode

```
iterator erase(it)
```

permettant de supprimer un élément, mais...

**Attention !** on **ne** peut **pas** continuer à utiliser l'itérateur *it* sans autre !

(plus exactement : erase rend invalide tout itérateur et référence situé(e) au delà du premier point de suppression)

Exemple d'**erreur** classique :

```
vector<double> v;
...
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) v.erase(i);
```

(avec `bool cond(double);`)

**n'**est **pas** correct («Segmentation fault»)

pas plus que :

```
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) i = v.erase(i);
```

# Suppression d'un élément d'un container (2)

Ce qu'il faut faire c'est :

```
vector<double>::iterator next;
for (vector<double>::iterator i(v.begin()); i != v.end();
                    i = next) {
          if (cond(*i)) { next = v.erase(i); }
          else { next = ++i; }
}
```

ou mieux en utilisant remove_if (ou remove) de <algorithm> :

```
v.erase(remove_if(v.begin(), v.end(), cond), v.end());
```

mais qui sont de toutes façons «**coûteux**» (O(v.size()^2)) (voir transparent suivant)

# Suppression d’un élément d’un container (3)

En effet, un tableau dynamique **n’est pas la bonne SDA** si l’on veut détruire un élément au milieu **et** garder l’ordre (utiliser plutôt des *listes chaînées* pour cela)

Note : si l’on ne tient pas à garder l’ordre, on peut toujours faire :

```
for (unsigned int i(0); i < v.size(); ++i)
        if (cond(v[i])) {
                v[i] = v[v.size()-1];
                v.pop_back();
                --i;
}
```

# Tables associatives

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

On parle d'« associations clé–valeur »

Les tables associatives sont définies dans la bibliothèque <map>.

Elles nécessitent deux types pour leur déclaration : le type des « clés » (les index) et le type des éléments indexé.

Par exemple, pour indexer des nombres réels par des chaînes de caractères on déclarera :

```
map<string,double> une_variable;
```

Si l'ordre (operator<) des clés n'importe pas, on utilisera une unordered_map.

## Tables associatives – exemple

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
map<string,double>
moyenne;
moyenne["Informatique"] =
5.5;
moyenne["Physique"] = 4.5;
moyenne["Histoire des
maths"] = 2.5;
moyenne["Analyse"] = 4.0;
moyenne["Algebre"] = 5.5;
// parcours de tous les elements
for (map<string,double>::iterator
i(moyenne.begin());
i != moyenne.end(); ++i)
cout << "En " << i->first << ", j’ai " << i-
>second
<< " de moyenne." << endl ;
// recherche
cout << "Ma moyenne en Informatique
est de ";
cout << moyenne.find("Informatique")-
>second << endl;
return 0;   }
```

# Piles et files

Pour utiliser les piles de la STL : #include <stack>

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Elles sont définies dans la bibliothèque <queue>.

Une pile de type *type* se déclare par stack<*type*> et une file d'attente par queue<*type*>. par exemple :

stack<double> une_pile;

queue<char> attente;

méthodes :

| | |
|---|---|
| *Type* top() | accède au premier élément (sans l'enlever) |
| void push(*Type*) | empile/ajoute |
| void pop() | dépile/supprime |
| bool empty() | teste si la pile/file est vide |

# Piles - exemple

```
#include <stack>
using namespace std;
...
bool check(string s) {
    stack<char> p;
    for (unsigned int i(0); i < s.size(); ++i) {
        if ((s[i] == '(') || (s[i] == '['))
            p.push(s[i]);
        else if (s[i] == ')') {
            if ((!p.empty()) && (p.top() == '('))
                p.pop();
        else
            return false;
    } else if (s[i] == ']') {
        if ((!p.empty()) && (p.top() == '['))
            p.pop();
        else
            return false;
        }
    }
    return p.empty();               }
```

# Conclusion

Il existe beaucoup d'outils prédéfinis dans la bibliothèque standard de C++

Le but n'est évidemment pas les connaître tous par cœur, mais de **savoir qu'ils existent** pour penser aller chercher dans la documentation les informations complémentaires.