ISEN ALL IS DIGITAL! OUEST | yncréa

Les bibliothèque standards en C et C++

La bibliothèque standard en C

Arborescence du système de fichiers

La librairie standard en C et C++

- La **SL (Standard Library)** est une bibliothèque C++ contenant des classes et des fonctions. Elle fournit des outils tels que :
  - un type pour la manipulation les chaînes de caractères
  - des types pour la manipulation de flux (fichiers, entrée et sortie standard…)
  - une collection d'algorithmes de bas niveau comme le tri
  - ...

La librairie standard en C et C++

- La **SL (Standard Library)** est une bibliothèque C++ contenant des classes et des fonctions. Elle fournit des outils tels que :
  - un type pour la manipulation les chaînes de caractères
  - des types pour la manipulation de flux (fichiers, entrée et sortie standard…)
  - une collection d'algorithmes de bas niveau comme le tri
  - ...

- Par exemple → **math.h**, **stdio.h** et **stdlib.h** sont des éléments de la librairie standard et fournissent des fonctions telles que : **sqrt()**, **printf()**, **rand()** etc.

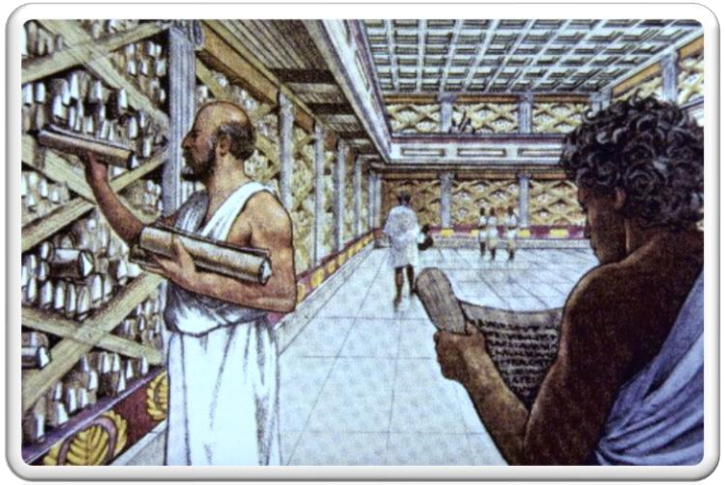
La librairie standard en C et C++

- La **SL (Standard Library)** est une bibliothèque C++ contenant des classes et des fonctions. Elle fournit des outils tels que :
  - un type pour la manipulation les chaînes de caractères
  - des types pour la manipulation de flux (fichiers, entrée et sortie standard…)
  - une collection d'algorithmes de bas niveau comme le tri
  - ...
- Par exemple → **math.h**, **stdio.h** et **stdlib.h** sont des éléments de la librairie standard et fournissent des fonctions telles que : **sqrt()**, **printf()**, **rand()** etc.

La SL du C++ contient aussi la SL du C mais remplace leurs en-têtes « XXX.h » par « cXXX »

De plus, la SL passe tout ses membres dans le **namespace *« std ».***

La Librairie de modèles standard du C++



SL vs STL

- **La Librairie de modèles standard (Standard Template Library : « *STL* »)** est une extension de la librairie standard du C++ (elle même extension de la librairie du C.

SL vs STL

- **La Librairie de modèles standard (Standard Template Library : « *STL* »)** est une extension de la librairie standard du C++ (elle même extension de la librairie du C.
- **Elle est mise en oeuvre à l'aide du système de modèles (Templates).**

SL vs STL

- **La Librairie de modèles standard (Standard Template Library : « *STL* »)** est une extension de la librairie standard du C++ (elle même extension de la librairie du C.
- **Elle est mise en oeuvre à l'aide du système de modèles (Templates).**

Elle fournit notamment :

- les classes conteneurs (vector, map, list)
- les itérateurs
- les algorithmes d'insertion/suppression, recherche et tri
- la classe string pour la manipulation des chaînes de caractères
- **…**

SL vs STL

- **La Librairie de modèles standard (Standard Template Library : « *STL* »)** est une extension de la librairie standard du C++ (elle même extension de la librairie du C.
- **Elle est mise en oeuvre à l'aide du système de modèles (Templates).**

Elle fournit notamment :

- les classes conteneurs (vector, map, list)
- les itérateurs
- les algorithmes d'insertion/suppression, recherche et tri
- la classe string pour la manipulation des chaînes de caractères
- **…**

Hormis la dernière, la classe string, nous n'avons pas encore vu le reste. Sauf les tris, que vous avez vu en algo et qui sont fournis dans la STL !

STL : conteneur

Conteneurs

- La STL fournit un ensemble de classes conteneurs

Conteneur	Entête	Description
std::vector<T>	<vector>	Tableau dynamique de T
std::list<T>	<list>	Liste doublement chaînée de T
std::set<T>	<set>	Ensemble de T
std::map<K,V>	<map>	Tableau associatif de K vers V
std::pair<K,V>	<utility>	Couple de K et V

std::vector<T>

La classe vector est proche par son utilisation du tableau en C.

- **Les avantages de la classe vector sont :**
  - Fonctionne avec n'importe que type de données
  - Capable de réallouer automatiquement l'espace mémoire nécessaire via la méthode push_back
  - Désallocation de mémoire automatique
  - Nombreuses méthodes d'accès au éléments et aux caractéristiques d'ensemble

std::vector<T>

- Création d'un vecteur de 4 entiers contenant 25 / 25 / 25 / 25

```
std::vector<int> my_vector2(4,25);

// Réassignation des valeurs de ce vecteur

my_vector2[0]=5;
my_vector2[1]=3;
my_vector2[2]=7;
my_vector2[3]=4;
my_vector2[4]=8;
```

std::vector<T>

- Création d'un vecteur de 4 entiers contenant 25 / 25 / 25 / 25 :

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

int main(){
std::vector<int> my_vector1;

// Ajout de deux éléments
my_vector1.push_back(4);
my_vector1.push_back(2);

// La méthode size précise le nombre d'entrée courante
for(size_t i=0; i<my_vector1.size();i++)
{
   std::cout<<i<<" "<<my_vector1[i]<<std::endl;
}
```

std::vector<T>

- La classe vector est une classe intelligente :
  - En interne elle utilise un tableau contigu classique alloué avec new et détruit avec delete.
  - On peut accéder au pointeur sur ce tableau si on veut (my_vector.data())
  - Lorsqu'on fait grandir le tableau, il réalloue un tableau de la bonne taille et transfert des éléments
  - Pour éviter de faire ça constamment (très gourmand) souvent, le vector alloue plus de donnée que ce qu'on utilise pour garder un peu de marge, selon des algorithmes complexes qui varient d'une implémentation à l'autre.

- Conclusion → c'est une classe efficace avec pas mal de chose intégrée pour manipuler un tableau de n'importe quel type, mais elle consomme plus de mémoire et peut-être moins efficiente qu'un tableau statique bien pensé.
- **Plus efficace encore -> std::array, le tableau est statique (donc plus de push back etc.) mais remplace avantageusement les tableaux du C.**

std::list<T>

- La classe list est une liste doublement chainée

- **Les avantages de la classe list sont :**
- Fonctionne avec n'importe que type de données
- Capable de rajouter de façon très efficace des éléments n'importe ou dans la liste
- Profite de toutes les fonctions classiques des autres containeurs de la STL (size, find etc.)

std::list<T>

```
#include <algorithm>
#include <iostream>
#include <list>

int main() {
std::list<int> l = { 7, 5, 16, 8 };

l.push_front(25); // On ajoute un élément au début de la liste
l.push_back(13);  // On ajoute un élément à la fin de la liste

// On cherche un élément grâce à std::find entre le début et la fin de la liste
auto it = std::find(l.begin(), l.end(), 16);

if (it != l.end())
{
 l.insert(it, 42); // On insère 42 à cet endroit si on a trouvé 16
}

//Itération et affichage des valeurs de la liste
for (int n : l)
{
  std::cout << n << '\n';
 }
}
```

std::set<T>

La classe set est un ensemble **d'éléments triés** :

- on peut ajouter on supprimer un élément, mais on ne peut pas en modifier.
- À chaque ajout d'élément celui-ci va se trier automatiquement par rapport aux autres.
- Comme on veut pouvoir trier les éléments comme on veut, l'utilisateur définis une fonction « compare » qui est donné au set et qui permet de faire le tri.

- Les std::set sont souvent implémenté comme des arbres binaires similaires à ceux que vous avez vu dans le **tri par tas** en algo ! Mais plus spécifiquement ils sont proches des **arbres rouge-noir**.

  ➔ Pour les plus curieux, voir le wiki sur le sujet.

https://fr.wikipedia.org/wiki/Arbre_bicolore

std::set<T>

```cpp
#include <iostream>
#include <iterator>
#include <set>
int main()
{
    std::set<int, std::greater<int>> s1; //set vide avec fonction greater en comparaison

    s1.insert(40); // insertion dans un ordre aleatoire
    s1.insert(30);
    s1.insert(60);
    s1.insert(20);
    s1.insert(50);
    s1.insert(50); // Un seul 50 sera ajouté car interdit deux éléments identiques !
    s1.insert(10);

    // On peut assigner les éléments de s1 dans s2, mais avec une fonction compare par défaut ici !
    std::set<int> s2(s1.begin(), s1.end());

    s2.erase(s2.begin(), s2.find(30)); // Suppression des éléments inférieur à 30 !

    int num;
    num = s2.erase(50); // supprime les éléments de valeur 50 dans s2
    return 0;
}
```

60,50,40,30,20,10

10,20,30,40,50,60

30,40,50,60

num = 1 s2 = 30,40,60

std::map<K,V>

La classe map est un peu comme un set, un ensemble **d'éléments triés** :

- Mais les éléments sont un couple de valeur, la map est alors un tableau qui associe chaque K à une valeur V.
- Les éléments sont triés en fonctions de K (K pour Key = clé)

```cpp
#include <iostream>
#include <map>
#include <string>
#include <iterator>
int main()
{
  std::map<std::string, int> Tableau_sol;
  Tableau_sol.insert(std::make_pair("Terre", 1)); // Méthode d'insertion no 1
  Tableau_sol.insert(std::make_pair("Lune", 2));
  Tableau_sol["Soleil"] = 3;                      // Méthode d'insertion no 2
  Tableau_sol["Terre"] = 4; // il ne peut y avoir qu'une seul élément avec la clé « Terre » donc
  (Terre,1) se retrouve remplacé par (Terre,4)

  // Insert renvoi une valeur qui nous dit si l'insertion est OK ou non.
  if(Tableau_sol.insert(std::make_pair("Terre", 1)).second == false)
  {
    std::cout<<« Terre,1 n'a pas été inséré car un élément Terre existe"<<std::endl;
  }
  // Recherche d'éléments !
  if(Tableau_sol.find("Soleil") != Tableau_sol.end())
       std::cout<<" Pas de soleil ! "<<std::endl;
  if(Tableau_sol.find("Mars") == Tableau_sol.end())
       std::cout <<" Pas de Mars ! "<< std::endl;

  return 0; }
```

std::pair<A,B>

La classe pair est un type très simple qui permet de regrouper en un seul élément deux éléments de types A et B !

- Pas mal de façon de l'initialiser :

```
std::pair<std::string,double> product1; // par defaut
product1 = std::make_pair(std::string("lightbulbs"),0.99); // Avec la fonction make_pair

std::pair<std::string,double> ("tomatoes",2.30);  // par valeur
std::pair<std::string,double> product3 (product2); // Par copie (constructeur de copie)

// Depuis C++11 on peut aussi faire :
std::pair<int,int> p = {2, 3};
std::pair<int,int> p({2, 3});
std::pair<int,int> p{2, 3};

// Et depuis C++ 17 en utilisant la déduction des types des templates on peut faire :

std::pair p(2, 4.5);
```

Bonne nouvelle !!

- On peut retourner deux valeurs dans une fonction :

```
#include <iostream>
#include <utility>

std::pair<char, double> funcThatReturnsACharAndDouble()
{
     std::pair<char, double> twoValues;
     twoValues.first = 'c';
     twoValues.second = 2.5;
     return twoValues;
}

int main()
{
     std::pair<char, double> twoValuesFromFunc = funcThatReturnsACharAndDouble();
     std::cout << "< " << twoValuesFromFunc.first << " , " << twoValuesFromFunc.second << " > " << std::endl;

     return 0;
}
```

Bonne nouvelle !!

- On peut retourner deux valeurs dans une fonction :

```cpp
#include <iostream>
#include <utility>

std::pair<char, double> funcThatReturnsACharAndDouble()
{
    std::pair<char, double> twoValues;
    twoValues.first = 'c';
    twoValues.second = 2.5;
    return twoValues;
}

int main()
{
    std::pair<char, double> twoValuesFromFunc = funcThatReturnsACharAndDouble();
    std::cout << "< " << twoValuesFromFunc.first << " , " << twoValuesFromFunc.second << " > " << std::endl;

    return 0;
}
```

```
< c , 2.5 >
```

Parcours des conteneurs

- On peut parcourir les éléments d'un conteneur en utilisant :
  - Un itérateur explicite (en le demandant au conteneur)
  - Un itérateur implicite (boucles de type "for")
  - Un parcours par indice

Parcours des conteneurs

- On peut parcourir les éléments d'un conteneur en utilisant :
  - **Un itérateur explicite (en le demandant au conteneur)**
  - Un itérateur implicite (boucles de type "for")
  - Un parcours par indice

Parcours des conteneurs par itérateur explicite

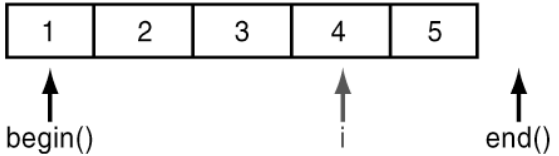
- Parcours en avant :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << endl;
    }
    return 0;
}
```

Début de la collection

Fin de la collection

Comme les pointeurs



Parcours des conteneurs par itérateur explicite

- Modification de valeurs :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        *it = 0;
    }
    return 0;
}
```

Parcours des conteneurs par itérateur explicite

- Modification de valeurs :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    for (vector<int>::const_iterator it = v.begin(); it != v.end(); it++) {
        *it = 0; // erreur de compilation -> le parcours est en lecture seule
    }
    return 0;
}
```

Parcours des conteneurs par itérateur explicite

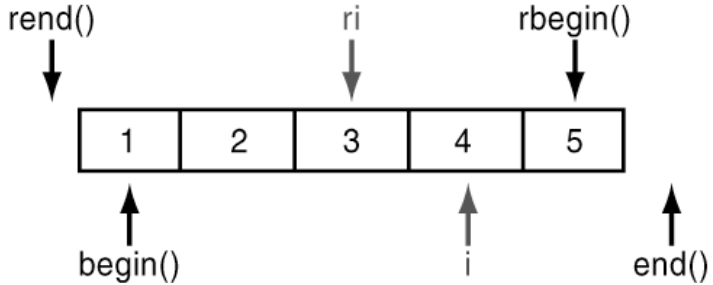
- Filtrage de valeurs :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {-42, 1, -10, -5, 2, 3, -8};
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        if(*it<0){
            v.erase(it);
            it--;
        }
    }
    return 0;
}
```

Parcours des conteneurs par itérateur explicite

- Parcours en arrière :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (vector<int>::reverseiterator it = v.rbegin(); it != v.rend(); it++) {
        cout << *it << endl;
    }
    return 0;
}
```



Parcours des conteneurs

- On peut parcourir les éléments d'un conteneur en utilisant :
  - Un itérateur explicite (en le demandant au conteneur)
  - **Un itérateur implicite (boucles de type "for each")**
  - Un parcours par indice

Parcours des conteneurs par itérateur implicite

- Récupération des éléments par copie :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    for(int i : v){
       cout << i << endl;
    }
    return 0;
}
```

Parcours des conteneurs par itérateur implicite

- Récupération des éléments par copie :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    for(int i : v){
       i = 0; // aucun effet sur "v"
    }
    return 0;
}
```

Parcours des conteneurs par itérateur implicite

- Récupération des éléments par référence :

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1, 2, 3};
    for(int& i : v){
       i = 0;
    }
    // "v" ne contient que des 0
    return 0;
}
```

Parcours des conteneurs par itérateur implicite

- Exemple avec une map :

```
#include <map>
using std::map;

#include <utility>
using std::pair;
using namespace std;

#include <iostream>

int main() {
    map<string, int> userIds = {
            { "John Doe", 404 },
            { "Martin Freeman", 42 }
    };
    for(const pair<string, int>& userId : userIds ){
        cout << userId.first << " -> " << userId.second << endl;
    }
    return 0;
}
```

Parcours des conteneurs

- On peut parcourir les éléments d'un conteneur en utilisant :
  - Un itérateur explicite (en le demandant au conteneur)
  - Un itérateur implicite (boucles de type "for each")
  - **Un parcours par indice**

Parcours des conteneurs par indice

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> v = {1,2,3};
    short nbValues = v.size();
    for(int i = 0; i < nbValues; i++){
        v[i] = 0; // ou v.at(i) = 0
    }
    return 0;
}
```

STL : algorithmes

Algorithmes

- La plupart des algorithmes sont définies dans **<algorithm>**, et couvrent un large panel des opérations courantes sur les conteneurs, par exemple:
  - application d'une fonction sur chaque élément : for_each()
  - recherche : find(), count(), …
  - génération : copy(), fill(), …
  - tri : is_sorted(), sort(), …
  - opérations ensemblistes : set_union(), set_difference()...

- Dans **<numeric>**, il existe des algorithmes spécifiques aux contenus numérique
  - exemple : accumulate() accumule les valeurs du conteneur (par défaut en faisant une somme mais on peut spécifier une fonction personnalisée)

STL : string

String

- Entête à inclure : **<string>**

- Le type string est une classe qui modélise une chaîne de caractères en C++
  - Les allocations et désallocations de mémoire sont automatiques (via les constructeurs et les destructeurs)
  - La concaténation se fait simplement avec l'opérateur **+**
  - La comparaison se fait avec l'opérateur **==**
  - L'opérateur **=** fait une copie profonde
  - La conversion d'une chaîne C (char*) en string est simple et est souvent faite de manière implicite

String

- Exemple :

```
using namespace std;
#include <string>
int main() {
    string s1 = "foo"; // constructeur implicite
    string s2("bar"); // constructeur explicite
    string s3 = s1 + s2; // vaut "foobar"
    string s4 = s3; // vaut "foobar"
    bool equal = (s4 == s3); // vaut true
    return 0;
}
```