Contents

1 Vector

#include <vector>

C'est une classe en C++ qui représente un tableau de taille dynamique, c'est à dire ici que la taille n'est pas fixe tout au long du déroulement du programme.

1.1 Syntaxe

A la déclaration on a:

- vector<TYPE> nom; où TYPE représente le type de donnée que contiendra le tableau (int, double, string ou une classe).
- vector<TYPE> nom(taille); Ici c'est le constructeur qui prend un paramètre qui est la taille initiale du vecteur (elle peut varier plus tard).
- vector<TYPE> nom(taille, valeurInitiale); Ici on crée un tableau d'une taille donnée et on initialise les cases par valeurInitiale.
- vector<TYPE> nom{valeurs1,valeurs2}; Initialise directement le tableau avec des valeurs.
- vector<TYPE> v1{v2}; Copie v2 dans v1 (les deux doivent contenir le même type de données).

1.2 Méthodes usuelles

- front() retourne le premier élément du tableau.
- size() retourne le nombre d'élément du tableau.
- back() retourne le dernier élément du tableau.
- push_back(valeur) ajoute valeur à la fin du tableau (même si la taille de départ est déjà atteinte).
- pop_back() retire l'élément à la fin du tableau.
- resize(nouvelleTaille) redimensionne le tableau.
- clear() nettoie le tableau.

Il faut noter que Vector s'utilise comme les tableaux en ce qui concerne l'indexage:

Pour vector<int> v{1,2,3}; v[0] donne 1.

2 List

#include <list>

Ce sont des listes elles fonctionnent exactement comme des tableaux mais, avec les Vector les emplacements des cases se suivent en mémoires, mais ce n'est pas le cas pour les listes. Ici, chaque case à l'addresse de la case suivante. Mais toutes ces opérations sont internes.

2.1 Syntaxe

A la déclaration on a:

- list<TYPE> nom; où TYPE représente le type de donnée que contiendra la liste (int, double, string ou une classe).
- list<TYPE> nom(taille); Ici c'est le constructeur qui prend un paramètre qui est la taille initiale de la liste (elle peut varier plus tard).
- list<TYPE> nom(taille, valeurInitiale); Ici on crée une liste d'une taille donnée et on initialise les cases par valeurInitiale.
- list<TYPE> nom{valeurs1,valeurs2}; Initialise directement le tableau avec des valeurs.
- list<TYPE> v1{v2}; Copie v2 dans v1 (les deux doivent contenir le même type de données).

l'exception de resize(), les listes utilisent les mêmes méthodes que les vector.

3 Deque

#include <deque>

Avec ce conteneur, on peut ajouter une valeur à une extrémité, le retirer à l'autre, supprimer une valeur au début ou à la fin. La taille est variable également.

3.1 Syntaxe

A la déclaration on a:

- deque<TYPE> nom; où TYPE représente le type de donnée que contiendra le deque (int, double, string ou une classe).
- deque<TYPE> nom(taille); Ici c'est le constructeur qui prend un paramètre qui est la taille initiale de la liste (elle peut varier plus tard).
- deque<TYPE> nom(taille, valeurInitiale); Ici on crée un deque d'une taille donnée et on initialise les cases par valeurInitiale.
- deque<TYPE> nom{valeurs1,valeurs2}; Initialise directement le deque avec des valeurs.

3.2 Méthodes usuelles

- front() retourne le premier élément du tableau.
- size() retourne le nombre d'élément du tableau.
- back() retourne le dernier élément du tableau.
- push_back(valeur) ajoute valeur à la fin du tableau (même si la taille de départ est déjà atteinte).
- pop_back() retire l'élément à la fin du tableau.
- push_front(valeur) ajoute valeur au début du tableau (même si la taille de départ est déjà atteinte).
- pop_front() retire l'élément à la fin du tableau.

L'accès direct des valeurs est possible grâce à l'indexage.

4 Unordered_map

#include <unordered_map>
unordered_map<type_clef, type_valeur> dictionnaire
{
cle1, valeur,
clef2, valeur2

};

Chaque élement de la liste est une paire, ainsi donc on a
paire.first pour la clef et .second pour la valeur

pair<type_clef, type_valeur> nom = valeur;

valeur.first
valeur.second

dic.insert(); ajout d'une nouvelle clé

Information sur insert(), insert renvoie une pair ,dont l'element second est un booléen qui est vrai si l'element vient d'être inséré et faux sinon.

insert_or_assign(key,val) ajoute si inexistant et met à jour si existe.

Mis à part celà l'utilisation est la même qu'en Python.

dic["clef"] = valeur;

Rechercher une valeur:
dic.find(clef); // renvoie un itérateur sur la position si trouvé et un itérateur sur end
sinon

Et pour obtenir la valeur il suffit de faire it->second

dic.erase(key) supprime la pair de clé key.

dic.size() renvoie la taille.

5 unordered_set

#include<unordered_set>

Ils sont semblable aux dico sauf qu'ici la clef est unique:

std:: unordered_set<string> pseudos { "mehdidou99","informaticienzero" };

// Ajouts pseudos.insert("Dwayn");

// Parcours for (auto const & cle : pseudos)
{
std:: cout « cle « std:: endl;
}

find(valeur)
erase("valeur");

6 map et set

Pour map et set l'utilisation est pareil pour respectivement unordered_map et unordered_set juste que les clé sont rangés par ordre croissant.