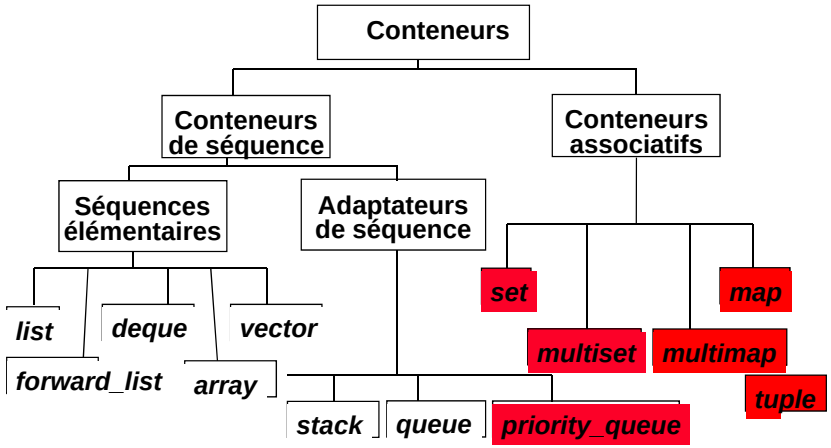
Cours n° 7

Standard Template Library II

Sommaire

Introduction



Définition

Relation binaire associée à un ensemble d'Items

Réflexive (tout Item est en relation avec lui-même)

Antisymétrique (pas de relation mutuelle entre Items distincts)

Transitive (deux Items sont mis en relation s'ils sont en relation avec un troisième)

Exemples et contre-exemples
ordre lexicographe est une relation d'ordre sur les listes de caractères,
« est antérieur ou égale à » est une relation d'ordre sur les données temporelles,
« est plus petit que » n'est pas une relation d'ordre (non réflexive)

Ensemble ordonné

Ensemble d'Items muni d'une relation d'ordre

Ordre total

Relation entre tous les couples d'Items

Foncteurs arithmétiques et logiques

Redéfinition des opérateurs arithmétiques

plus<T>	retourne (arg1 + arg2)
minus<T>	retourne (arg1 - arg2)
multiplies<T>	retourne (arg1 * arg2)
divides<T>	retourne (arg1 / arg2)
modulus<T>	retourne (-arg1)
negate<T>	

Redéfinition des opérateurs logiques

logical_and<T>	retourne (arg1 && arg2)
logical_or <T>	retourne (arg1 || arg2)
logical_not<T>	retourne (!arg1)

Paramètres des algorithmes STL

testFoncteurSTL.cpp

Application du foncteur « plus »

```
template <class T, class F>
T applique(T i, T j, F foncteur) {
    // Applique l'opérateur fonctionnel au foncteur
    // avec comme arguments les deux premiers paramètres :
    return foncteur(i, j);
}

int main(void) {
    Identite Id1("Ducrut","Lucie"), Id2 ("Herbrant","Claude");

    // Utilisation du foncteur "plus" pour faire faire une addition
    // à la fonction "applique" :
    cout << applique(Id1, Id2, plus <Identite>()) << endl;
    return 0;
}
```


Foncteurs de comparaison

Définition de la relation d'ordre à partir d'un prédicat binaire de comparaison

Compare : Objet de type fonction binaire générique (6 possibilité en C++)

equal_to <T>	retourne (arg1 == arg2)
not_equal_to <T>	retourne (arg1 != arg2)
greater <T>	retourne (arg1 > arg2)
less <T>	retourne (arg1 < arg2)
greater_equal <T>	retourne (arg1 >= arg2)
less_equal <T>	retourne (arg1 <= arg2)

Redéfinition de l'opérateur choisi

Patient.h

Redéfinition de l'opérateur « >= » (1/2)

```
#include <iostream>// déclaration des flots standard
#include <string>
using namespace std;
class Patient {
private:
      int age;
      char sexe;
public:
      string nom;
      Patient();
      Patient(string n, int a, char s);
// fonction de comparaison appelée par greater_equal
      bool operator >= (const Patient & p) const;
};
```

Redéfinition de l'opérateur « >= » (2/2)

Patient.cpp

```
#include "Patient.h"
Patient::Patient() {} // Constructeur vide
// Constructeur avec initialisation
Patient::Patient(string n, int a, char s) {
nom = n; age = a; sexe = s;
}
// fonction de comparaison appelée par greater_equal
bool Patient::operator >= (const Patient & p) const {
if (p.age > age) return true;
if (p.age < age) return false;
if (sexe == p.sexe) return true;
if (p.sexe == 'F') return true;
return false;
}
```


Adaptateur priority_queue

priority_queue<T, deque<T> [,Compare<int>]> fp;

Déclaration d'une file à priorité d'éléments de type T

Implémentation d'une structure abstraite de file à priorité

Fonctions membres

bool **empty**();
true si la file est vide

size_type **size**() const;
Taille de la file

T& **top**() :
Lecture de la tête de la file (élément prioritaire)

void **push**(const T&);
Ajout de l'élément en queue

T& **pop**()(const T&);
Suppression d'un élément en tête

priority_queue avec le foncteur « greater_equal »

testPriorityQueue.cpp

```
priority_queue<Patient, vector<Patient>, greater_equal <Patient> >
fpp;
fpp.push(*new Patient("Line", 25, 'F'));
fpp.push(*new Patient("Lucie", 10, 'F'));
fpp.push(*new Patient("Simon", 9, 'H'));
fpp.push(*new Patient("Eric", 25, 'H'));
fpp.push(*new Patient("Jean", 26, 'H'));

cout << fpp.size() << endl;
Patient p;
while (fpp.empty() == false) { p = fpp.top(); fpp.pop();
cout << p.nom << " ";
}
```

```
5
Jean Line Eric Lucie Simon
```


Introduction

Ensemble ordonné (fonction de comparaison)

set

Eléments tous distincts, pas d'insertion d'une nouvelle occurrence

multiset

Plusieurs occurrences possibles du même élément

Conteneurs indexés par une clé de type quelconque et triés suivant cette clé (généralisation des séquences)

Concept de paire (clé, valeur) pour représenter un élément

map

Clé et valeur sont distinctes, clé unique par élément

multimap

Une même clé peut correspondre à plusieurs éléments

Insertion et suppression

bool **empty**(); true si le conteneur est vide

size_type **size**() const; Nombre d'éléments du conteneur

iterator **insert**(const key_T& k) Insertion de l'élément k

iterator **insert**(iterator pos, const key_T& k)

Insertion de l'élément k à partir l'emplacement pos

void **insert**(iterator debut, iterator fin)

Insertion de la séquence d'éléments

void **erase**(iterator i);

Suppression de l'élément référencé par l'itérateur i

void **erase**(iterator debut, iterator fin);

Suppression des éléments de la séquence [debut fin[

void **erase**(const key_T& k);

Suppression des éléments de clé k

void **clear**() Suppression de tous les éléments

Recherche

size_type **count**(const key_T& k) const; Nombre d'éléments de clé k

iterator **find**(const key_T& k); Recherche d'un élément de clé k

iterator **lower_bound**(const key_T& k) const;

Itérateur sur le 1er élément dont la clé est supérieure ou égale à k

iterator **upper_bound**(const key_T& k) const;

Itérateur sur le 1er élément dont la clé est inférieure ou égale à k

pair<iterator, iterator> **equal_range**(const key_T& k) const

Paire d'itérateurs <lower_bound, upper_bound>

key_compare **key_comp**() const;

Objet de type fonction binaire de comparaison de deux clés

value_compare **value_comp**() const;

Objet de type fonction binaire de comparaison de deux valeurs

testSet.cpp

Instanciation et propriétés

set<T [,Compare<T>]> s(n);

Déclaration d'un ensemble ordonné s de n Items uniques de type T

```
typedef set <Patient, greater_equal <Patient> > sPatient;
typedef sPatient::iterator itsPatient;
sPatient sp;
sp.insert(*new Patient("Line", 25, 'F'));
sp.insert(*new Patient("Lucie", 10, 'F'));
sp.insert(*new Patient("Simon", 9, 'H'));
sp.insert(*new Patient("Eric", 25, 'H'));
sp.insert(*new Patient("Jean", 26, 'H'));
for (itsPatient it = sp.begin();it != sp.end(); it++)
cout << it->nom << " ";
```

```
Simon Lucie Eric Line Jean
```


testMultiset.cpp

Instanciation et propriétés

multiset<T [,Compare<T>]> ms(n);

Déclaration d'un ensemble ordonné ms de n Items de type T

```
typedef multiset <string> msString;
typedef msString::iterator iMsString;
int main() {
msString amb;
amb.insert("canapé"); amb.insert("tapis"); amb.insert("table");
amb.insert("chaise"); amb.insert("chaise");amb.insert("bureau");
amb.insert("lampe"); amb.insert("fauteuil");

for (iMsString it = amb.begin();it != amb.end(); it++)
cout << *it << " ";
cout << endl << amb.count("chaise");

}
```

```
bureau canapé chaise chaise fauteuil lampe table tapis
2
```

Instanciation et propriétés

map<T1, T2 [,Compare<T1>]> mc(n);

Déclaration d'un ensemble mc de n couples (clés de type T1, valeurs associées de type T2)

pair **make_pair**(const key_T& k, const value_T& v)

Regroupement de la clé et de la valeur dans un seul élément

Insertion réussie si absence d'élément de même clé

pair<iterator, bool> **insert**(const T& e);

Tentative d'insertion d'un élément e

pair<iterator, bool> **insert**(iterator i, const T& e);

Tentative d'insertion d'un élément e à l'emplacement spécifié par l'itérateur i (amélioration du temps de recherche de l'emplacement)

void **insert**(iterator debut, iterator, fin)

Tentative d'insertion des éléments de [debut fin[d'une autre séquence

testMap.cpp

Exemple

```
typedef map<string, float, less<string> > mStringFloat;
void prix(mStringFloat prixFruit, string fruit) {
mStringFloat::iterator it;
if ((it = prixFruit.find(fruit)) == prixFruit.end())
cout << "fruit non référencé"; else cout << it->second;
cout << endl; }

int main() {
mStringFloat prixFruit;
prixFruit.insert(make_pair("poire", 1.5));
prixFruit.insert(make_pair("pêche", 2.7));
prixFruit.insert(make_pair("orange", 1.2));
prix(prixFruit, "poire"); prix(prixFruit, "pomme"); }
```

```
1.5
fruit non référencé
```


Instanciation et propriétés

multimap<T1, int [,Compare<T2>]> mm(n);

Déclaration d'un ensemble mm de n couples (clés de type T1, valeurs associées de type T2)

pair **make_pair**(const key_T& k, const value_T& v)

Regroupement de la clé et de la valeur dans un seul élément

Pas d'échec d'insertion

iterator **lower_bound**(const T& e);

Recherche du premier élément de clé k

iterator **upper_bound**(const T& e);

Recherche du dernier élément de clé k

testMultimap.cpp

Exemple

```
multimap<int, string, less<int>> conjug;
conjug.insert(make_pair(1, "parler"));
conjug.insert(make_pair(2, "choisir"));
conjug.insert(make_pair(3, "prendre"));
conjug.insert(make_pair(2, "finir"));
conjug.insert(make_pair(1, "manger"));

auto range = conjug.equal_range(2);
      for (auto it = range.first; it != range.second; ++it) {
            std::cout << it->second << " ";
         }
```

```
choisir finir
```

Instanciation et propriétés

tuple<T1, T2, T3, […]>;

Extension du conteneur Pair (plusieurs éléments de différents types)

tuple **make_tuple**(T1& v1, T2& v2, T3& v3)

création d'une association de variables de type tuple

tie (T1& v1, T2& v2, T3& v3) = t

dissociation des variables d'un tuple t

tuple **tuple_cat** (tuple& t1, tuple& t1, T3& t3)

création d'une association de tuple de type tuple

T& **get<n>** (tuple& t)

référence sur le nième élément d'un tuple t

testTuple.cpp

Exemple

```
typedef tuple<Chien, int, string> chientuple;
vector <chientuple> vct;

vct.push_back(chientuple(new Chien(true, "Milou"), 10, "Paris"))
vct.push_back(chientuple(new Chien(true, "Fidel"), 12, "Lille"))
vct.push_back(chientuple(new Chien(true, "Babette"), 8, "Rouen"))

for(chientuple t: vct) {
      cout << std::get<0>(t) << endl;
      cout << std::get<1>(t) << endl;
      cout << std::get<2>(t) << endl;
  }
```