La STL (ANSI C++)

J-P Dumas

#
L'OO : La réutilisation !

Les composants de la STL ont été écrits par des spécialistes

Ils sont plus efficaces qu'un composant "Maison"

La librairie STL est standard, son emploi rend les programmes portables

Historique du C++

- liée à l'histoire du C (72), ANSI-C (83) et d'UNIX (69)
- par Dr Bjarne Stroustrup (83) aux Bell Laboratories
- performant mais compliqué, librairies non standards
- C++ a évolué très longtemps
- ANSI/ISO C++ (88) avec introduction de la STL

Ken Thompson (assis) et Dennis Ritchie devant un PDP-11 fonctionnant avec Unix vers 1972

- bool : false ou true
- opérateurs de transtypage : static_cast, ...
- espace de nom : std::cin, math::PI, …
- Run Time Type Information : typeid, dynamic_cast, …
- Opérateurs : and, or, not, not_eq, xor, etc..
- Héritage multiple avec virtual
- La STL.

- ATL (pour DCOM, Microsoft)
- MFC (Microsoft)
- Glib (GNU)
- **STL (standard)**
- Plus des librairies spécifiques (mathématiques, graphiques, XML, multimédia, etc…)

- Développée par Stepanov & Lee (HP)
- Partie intégrante de la norme C++
- Disponible avec tous les compilateurs
- Indépendante des plateformes
- Source libre disponible chez HP
- Vaste, conceptuelle et complexe
- Utilisation à grande échelle.

Les contraintes du C++

- La STL doit proposer des outils génériques.
- **Pas de classe de base, ni d'interface.**
- La généricité sera obtenu par le concept de **Template** (classe et fonctions).
- La réutilisation par dérivation.
- La STL est définie dans l'espace de nom **std**.
- La STL est donc **fortement typée**.
- Beaucoup d'allocations seront implicites (pas de new et delete).

La généricité : classes génériques

Classes paramétrés par le type qu'ils manipulent. C’est un *patron* de module qui sera << instancié >>.

```
#include "Point.h"
int main() {
   Point<int> pointEntier(2, 3);
   Point<double> pointReel(3.14, 2.27);
   pointReel.translation(5.4, 8.65);
   cout << "P(" << pointEntier.x()  << ","
                        << pointEntier.y() << ")" << endl;
}
```

Déclaration d’une classe Template :

```
template <class T>
class Point {
public :
    Point(T x, T y) : _x(x), _y(y)  { }
    const T x() const             { return _x; }
    const T y() const             { return _y; }
    void translation(T x, T y);
protected:
    T _x;        T _y;
};
template <class T>
void Point<T>::translation(T x, T y) {
    _x += x; _y += y;
}
```

Exemple d'utilisation de la STL

```
#include <list>                    // pas de .h !
#include <iostream>
using namespace std;               // ou using std::cout

list <A>  a;      conteneur d'instances de A.
list <int> b;     conteneur d'entiers.
list <A>::iterator it;
it = a.begin();
cout  << (*it).get_nom() << endl;
```

On ne se préoccupe pas de la nature de ce que l'on manipule :

- **Conteneurs (collections)** : chargés de l'*Organisation* des données.
- **Itérateurs** : chargés de l '*Accès* aux données.
- **Fonctions :** chargées du *Traitement* sur ces données (dans **<algorithm>**).

L’idée Générale de la STL

Si le programmeur applique la fonction **sort()** sur un conteneur :

- d’entiers, il récupère le conteneur trié
- de comptes en banque, il obtiendra le conteneur trié. Il devra définir l’opérateur d’infériorité pour dire : “ *Qu’est ce qu’un compte inférieur* ”

La fonction s’applique sur la suite d ’objets et non sur leur nature.

Les Conteneurs

- Stockent des objets de même type ou de ses dérivés.
- Conteneurs par valeur ou par référence (déf par l’association) : classe (list<A>)ou pointeurs de classe (list<A*>).
- Ils organisent l’ensemble des objets en une séquence afin de la parcourir.
- Ils allouent et libèrent la mémoire des éléments stockés seuls (attention pour les collections de pointeurs) !

Gestion des classes et des dérivées

```
class D : public B {};

int main() {
    vector<B*> v;

    v.push_back(new B);
    v.push_back(new D);
    assert(typeid(*v[0]) == typeid(B));
    assert(typeid(*v[1]) == typeid(D));
    return 0;
}
```

Autres éléments de la STL

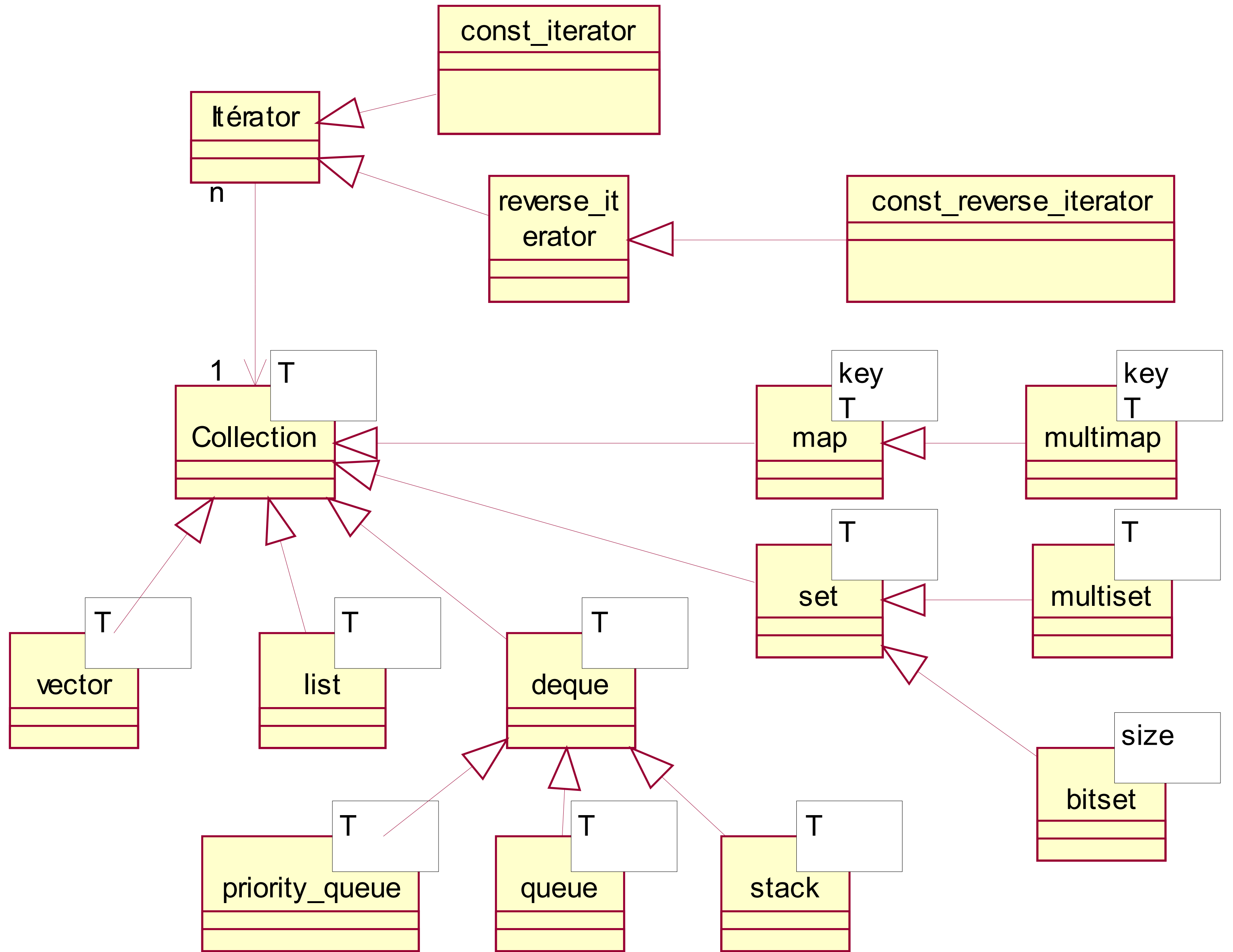
- string
- stream
- valarray
- pair
- etc …

Les Conteneurs (Collections)

2 familles de conteneurs :
- **de séquence** : vector, deque, list (ordonnés),
- **associatifs** : set, multiset, map, multimap, bitset (qualifiés)

Deux implémentations en mémoire
- **Une zone mémoire continue** : vector , deque.
- **La liste** : les objets sont dispersés dans la mémoire et reliés entre eux par un couple de pointeurs 'précédent ' ' suivant'.

Adaptateurs de conteneurs (sans itérateurs)
- stack, queue, priority_queue (piles)
- bitset (champ de bits)


const_iterator
Itérator
n
reverse_it
erator
const_reverse_iterator
1
T
Collection
key
T
map
key
T
multimap
T
set
T
multiset
T
vector
T
list
T
deque
size
bitset
T
priority_queue
T
queue
T
stack

Les Conteneurs

Tous les conteneurs disposent de méthodes communes :

- **begin()** renvoie l'itérateur sur le 1er objet
- **end()** renvoie l'itérateur après le dernier objet
- **empty()** renvoie true si le conteneur est vide
- **size()** renvoie la taille
- **insert()** ajoute un objet
- **erase()** supprimer un objet
- **clear()** vide le conteneur

Voir les détails sur le support de cours.

vector est un tableau dynamique !

- Ce conteneur se comporte comme un tableau (accès direct lecture/écriture aux éléments par indice)
- Mais il a tout les avantages d'un conteneur : insertion, ajout, suppression, etc.
- C'est le plus simple à utiliser !
- Insertion/suppression rapides à la fin, accès direct aux éléments par [].
- Ne pas confondre ajout et modification.
- #include <vector>

- [], push_back(), front(), back (), pop_back()
- insert() est possible mais lent
- empty (), erase ()
- reserve(), resize()
- Réservation à la construction :

```
vector<int> a(100);
a[0] = 12;
```

- Voir exemple.

- insertion/suppression rapides au début et à la fin.
- accès direct aux éléments par [].
- pour le reste comme vector.
- #include < deque >

- pile FiFo
- deque::push() insère à la fin
- deque::pop() retire au début
- Autres méthodes : empty(), back(), front().
- #include <queue>

- pile LiFo
- deque::push() insère à la fin
- deque::pop() retire à la fin
- Autres méthodes : empty(), back().
- #include < stack >

- pile ordonnée
- push() insère à sa place (tri)
- deque::pop() retire à la fin
- #include < priority_queue >

- liste chaînée avant arrière
- donc accès séquentiel avant/arrière
- insertion/suppression rapide partout
- déplacement par itérateur
- #include < list >

set

- collection à clef
- insertion d'un élément par insert()
- ici la clef est l'élément (définir l'opérateur <)
- recherche rapide par find() par l'opérateur ==
  (= end() si pas trouvé)
- unicité de la clef (notion d'ensemble).
- #include <set>
- multiset peut dupliquer les éléments

- collection à clef
- insertion d'une paire (clef, élément) par insert()
- clef = first   élément = second
- recherche rapide par find(clef)
- unicité de la clef (notion d'ensemble).
- #include <set>
- multimap peut dupliquer les clefs.

bitset : gestion de champs de bits

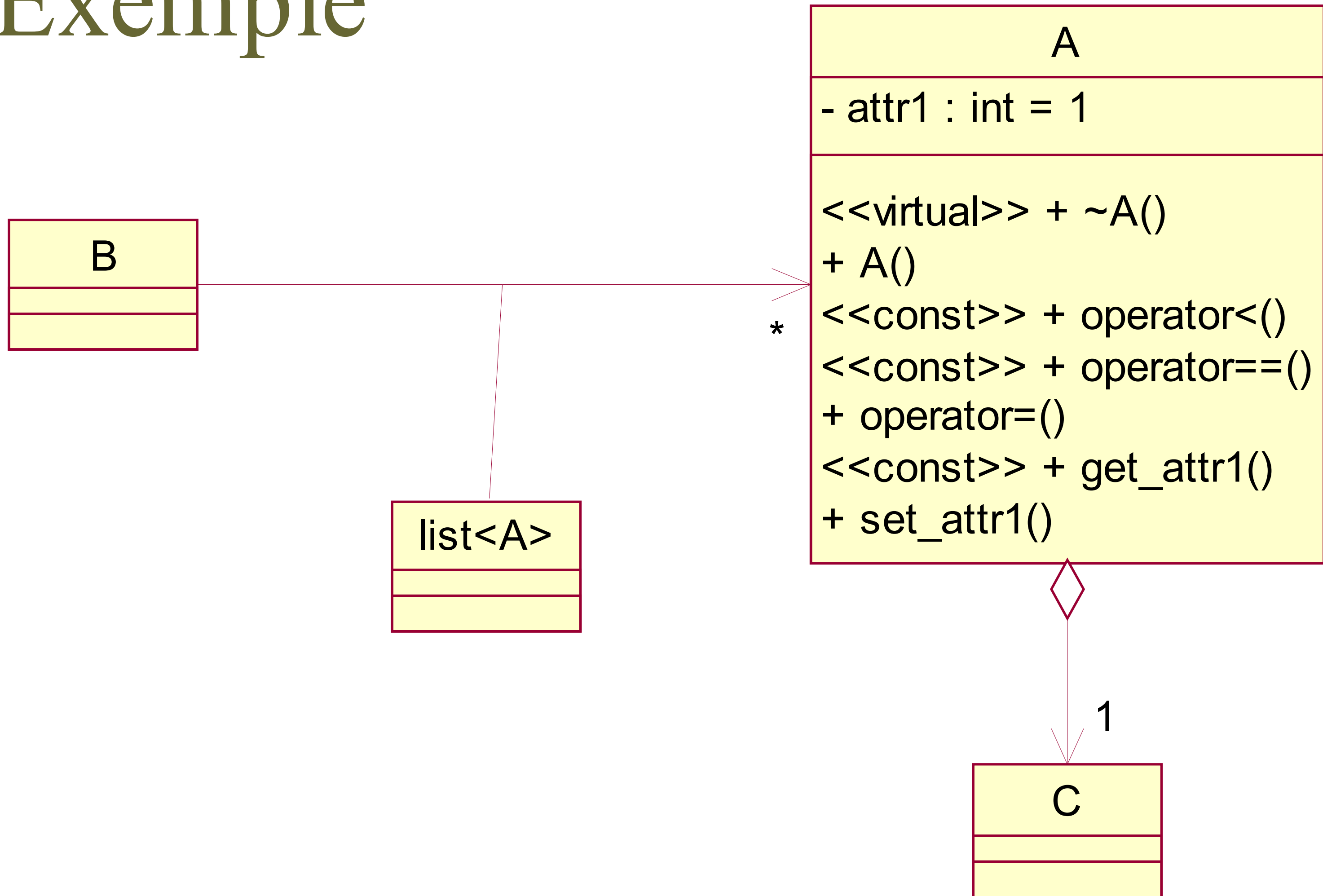
```
bitset<16> bs(21);          // 0000000000010101

bs[3] = 1;
bs.flip(5);
bs.set(15);
bs.reset(0);
cout << bs << endl;         // 1000000000111100
bs &= bitset<16> ( "1111111111100011" );
cout << bs << endl;         // 1000000000100000
bs |= 3328;                 // 0000110100000000
cout << bs << endl;         // 1000110100100000
string s = bs.to_string();
cout << bs.to_ulong() << endl;          // 36128
```

Surcharges d’opérateurs

Un conteneur peut gérer toute donnée si pour cette donnée sont définis (au moins par défaut) :

- X() : un constructeur,
- X(const X&) : un constructeur par copie,
- operator=(const X&) : l'opérateur d'affectation,
- bool operator==(const X&) : l'opérateur d'égalité (utile pour le recherche).
- bool operator<(const X&) : l'opérateur inférieur (utile uniquement pour les tris).

Exemple



Constructeur & destructeurs spécifiques

```
A::A() : attr1(10)
{
    theC = new C;
}
A::~A()
{
    delete theC;
}
```

```
A::A(const A& orig)
{
   attr1 = orig.attr1;
   theC = new C(*(orig.theC));
}
```

Opérateur d'affectation

Code par défaut :

```
A& A::operator=(A& rhs)
{
   return rhs;
}
```

```
bool A::operator==(const A& rhs) const
{
   return (attr1 == rhs.attr1);
}
```

```
bool A::operator<(A& rhs) const
{
   return (attr1 < rhs.attr1);
}
```

Les itérateurs

Les Itérateurs <Iterator>

- Un objet permettant d'accéder aux objets d'un conteneur (*it)
- Pattern différente de Java
- Supporte l’opération d’incrémentation **it++** pour “ passer ” à l’objet suivant (et d’autres fonction du conteneur parcouru : --it; it +=3; it-=2, …)

Mise en oeuvre

```
vector<int>::const_iterator it;

cout << "nombres { " ;
for ( it = nombres.begin();
        it != nombres.end(); it++ )
{
    cout << *it << " " ;
}
cout << " }\n" << endl ;
```

Les Itérateurs

- “ const_iterator ” : pour la lecture
- “ bidirectional_iterator ” : list, set, multiset, map, multimap
- “ random_acces_iterator” : vector, deque

Conteneurs et flots :

- Les itérateurs d'entrée (*input iterators*)
- Les itérateurs de sortie (*output iterators*)

Copie ou ajout d’un élément ?

- La copie utilise l’opérateur = pour remplacer la valeur d’un élément existant.
- L’ajout crée un nouvel élément, y copie la valeur de l’élément source puis l’insère dans le conteneur.
- La copie peut provoquer une exception si l’élément destination n’existe pas !

Notion de constructeur par copie

- L’insertion se fait avec ce constructeur.
- Sachant que l’original sera en général détruit, bien réfléchir aux conséquences d’une insertion par valeur dans un conteneur.
- Une conteneur d’éléments par référence pose moins de soucis dans certains cas !

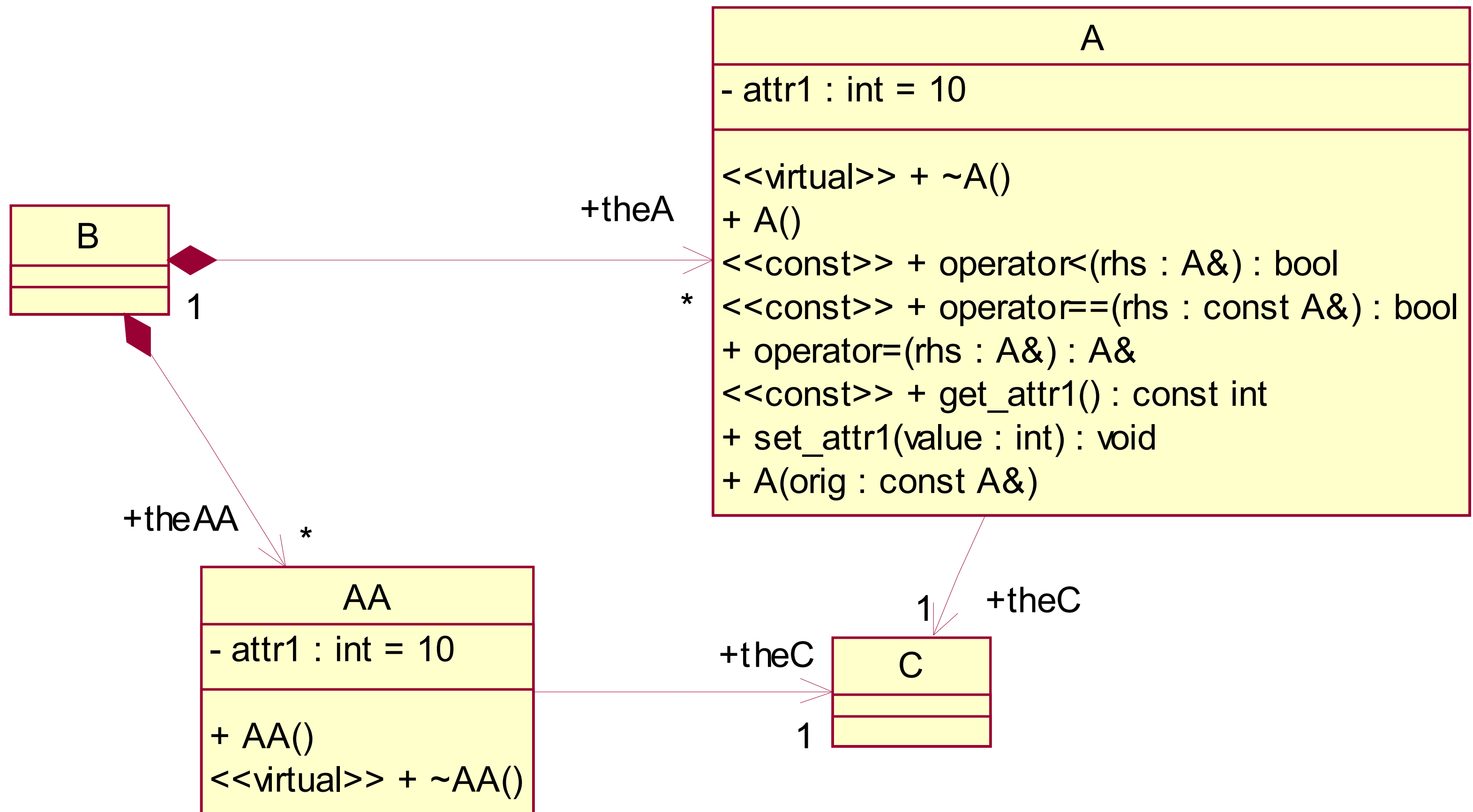
Insertion d’un élément par valeur

```
vector<A> col;

A* pa1 = new A(11);
A a2(12);
col.push_back( A(10) );
col.push_back( *pa1 );
col.push_back( a2 );
delete pa1;
cout << col[0].get_valeur() << endl;
```

Insertion par référence

```
A  a(12);
A* pa = new A(11);

col.push_back( pa );
col.push_back( &a ); // à ne pas faire
cout << col[0]->valeur << endl;
….
delete pa;          // supprimer d'abord l'élément
                    // du conteneur!
```

Exemple de pb !



Mise en oeuvre

```
B b;
list<AA> &col = b.theAA;

AA a;
cout << &a << " et " << a.theC << endl;
col.push_back(a);
AA &a1 = *col.begin();
cout << &a1 << " et " << a1.theC << endl;
/*
0012FF4C et 00322FF0
00322ED8 et 00322FF0
*/
```

C'est la même instance de C, si la première instance de AA est détruite, la deuxième devient fausse !

Suite

```
list<A> &cola = b.theA;
A a;
cout << &a << " et " << a.theC << endl;
cola.push_back(a);
A &a1 = *cola.begin();
cout << &a1 << " et " << a1.theC << endl;
/*
0012FF38 et 00322DE8
00322F20 et 00322F60
*/
```

Exemples (voir polys)

- Gestion d’une list,
- Gestion d’une set,
- Gestion d’une multiset,
- Gestion d’une map.

Conseils

- Utiliser si possible tous les concepts du C++ (plus de printf(), sprintf(), de char*, de itoa(), etc…)
- Travailler avec les stream pour les affichages, pour les fichiers.
- Utiliser systématiquement les collections plutôt que les tableaux statiques.
- Dériver les collections de la STL pour des collections spécifiques.

Exemple

On veut formater des données dans une chaîne de caractères (comme sprintf()) :

```
#include <iostream>
#include <sstream>      // manipule les stringstream
#include <iomanip>      // formatage des stream
#include <string>

using namespace std;
```

Suite

```
int main(void)
{
    stringstream s1;
    string s;
    int sec = 10;
    int mSec = 12;

    s1 << "duree : " << sec << "," << setw(3) << setfill('0') << mSec
                                                                    << " s";
    s = s1.str();
    cout << s << endl;    // affiche : duree : 10,012 s
    return 0;
}
```

Gestion de fichiers

Affichage séquentiel des températures contenues dans un fichier :

```
string nomFichier = "C:\\Temp\\Data\\1202.txt";
ifstream fichier;
unsigned short valeur;

fichier.open( nomFichier.c_str () );
while (fichier >> valeur) {
        cout << valeur << endl;
}
fichier.close();
```

Simulation de l'acquisition d'une température à l'aide d'une collection gérée de manière circulaire.

```
Temperature   CapteurTemperature::lire()
{
    static list<Temperature>::iterator it1 = valeurs.begin();

    Temperature t = *it1;
    if (it1 != valeurs.end())
        it1++;
    else
        it1 = valeurs.begin();
    return t;
}
```

Les Paires <Utility>

Une «paire» est un couple de 2 données. Utilisé comme élément d'une map ou d'une multimap. Elle dispose d'un constructeur et de 2 attributs : first et second

- *pair* :

```
pair<string, int> personne;
personne.first = "Durand";
Personne.second = 37;
```

- *make_pair* : permet de créer une pair.

```
pair<string, int> personne =
                make_pair("Dupont" ,34);
```

Les Itérateurs Adaptateurs

Lier un flux de données à un conteneur,

```
#include <iterator>
ostream_iterator<int>     ecran ( cout, " " );
copy ( vec1.begin(), vec1.end(), ecran  );
```

Réaliser les opérations d’insertion dans un conteneur

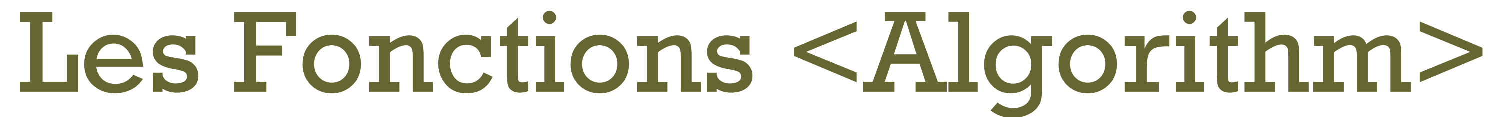
```
copy ( istream_iterator<float> ( fichierCoteR ),
    istream_iterator<float> ( ),
    back_inserter( vecCote )  );
```

Mise en œuvre des algorithmes

- des méthodes des conteneurs :
```
it = nombres.find(a1);
```
- des méthodes des itérateurs :
```
it.swap(col);
```
- des fonctions :
```
replace( it.begin(), it.end(),
                          old1, new1);
   for_each(database.begin(),
    database.end(), printEntry);
```

C’est un jeu de 70 fonctions traitant les algorithmes les plus connus :

- La copie,
- La suppression,
- Le remplacement,
- La transformation,
- La recherche avec un critère,
- Le tri.



S'appliquent à tous les objets ou une partie de 1 ou 2 conteneurs

Reçoivent 2 itérateurs définissant la séquence source

Parcourent la séquence pour traiter avec la fonction <algorithm> choisie

Algorithmes sans prédicat

Attention, les algorithmes sans prédicat s'utilisent en général pour des conteneurs d'objets (pas les conteneurs de références).

Ils nécessitent la surcharge d'un opérateur pour l'objet contenu : ==, <.

Exemple avec l'opérateur == (test sur l'attribut nom) :

```
bool GroupePortes::operator==(const GroupePortes& rhs) const {
    return nom == rhs.get_nom();
}
```

Un “foncteur ” est un objet définissant l'opérateur “( )”.

Les foncteurs prenant :

- 1 paramètre sont dites “unaires ”,
- 2 paramètres sont dits “binaires ”

Une fonction “prédicat” est un foncteur qui renvoie un booléen

Les fonctions <algorithm> utilisent des foncteurs unaires, binaires ou des prédicats

Prédicat : Foncteur booléen

```
class checkPrefix
{
  public:
        checkPrefix (int p) : testPrefix(p) { }
        int testPrefix;
        bool operator () (const entry_type& entry)
        {
              return  prefix(entry) == testPrefix;
         }
};

// Trouver le premier no. téléphone ayant ce code régional
where = find_if(database.begin(), database.end(),
                                              checkPrefix(code));
```

Les Foncteurs <Functional>

Définition d'un foncteur unaire

```
Template<class arg,class result>
    Struct unary_function {
        Typedef arg    argument_type;
        Typedef result result_type;
    }; // Pour une predicat, Result est un bool
```

Le foncteur adaptateur de fonction *ptr_fun(--)* convertit une fonction " maison " en foncteur

Exemple « Trier un Fichier »

```
ifstream fichierCoteR;
ofstream fichierCoteW;
vector<float> vecCote;
ostream_iterator<float> fichierW(fichierCoteW, "\n");
fichierCoteR.open("c:\\temp\\piece.dat");
copy(istream_iterator<float>(fichierCoteR),
        istream_iterator<float>(), back_inserter(vecCote));
sort(vecCote.begin(), vecCote.end());
fichierCoteR.close();
fichierCoteW.open("c:\\temp\\piece_t.dat");
copy(vecCote.begin(), vecCote.end(),  fichierW);
```