

## Gestion mémoire + STL

### Chapitre 5

- Allocation dynamique
- STL

## Références et pointeurs (1)

- Une référence est un deuxième nom pour une même variable  
`int i;`  
`int & ri = i; // déclaration d'une référence sur i`
- Paramètre passé par référence sur des objets pour éviter la copie (avec ou sans const) = sémantique d'adresse
- Paramètre passé par valeur sur des types de base = sémantique de valeur

## Références et pointeurs (2)

- Dès que vous allouez avec new, libérez avec delete
- Si vous allouez avec new[], libérez avec delete[]
  - (Cf Programmation Objet en langage C++, Alexandre Guidet)
- Sémantique d'adresse pour les pointeurs
  - Allocation dans le constructeur
  - Définir un constructeur copie et un opérateur =
  - Libération dans le destructeur

## Composition d'objets

- Quand l'objet est construit/détruit, son attribut l'est aussi

```
class Car {  
    Engine *itsEngine;  
    public :  
    Car () { itsEngine = new Engine; }  
    Car(Car& copy) { itsEngine = new Engine(copy. itsEngine ); }  
    Car& operator=(Car& copy) {  
        delete itsEngine;  
        itsEngine = new Engine(copy. itsEngine );  
        return *this;}  
    virtual ~Car() { delete itsEngine;}  
};
```

## Agrégation d'objets

- Certains attributs existent indépendamment de l'objet courant
- Ils ne sont pas détruits avec l'objet courant

```
class House {  
    Person **persons;  
    int personNb;  
    public :  
        House (int Size = 10) {persons = new Person * [Size]; personNb=0;}  
        void add(Person &P) {persons[personNb ++]=&P;}  
        virtual ~ House () { delete []persons ;};  
};
```

## Matrice de grande taille

```
class Matrice {  
    int width, height;  
  
    char ** tab;  
    // tableau à deux dimensions = pointeur  
    // de pointeurs  
  
    public:  
        Matrice (int largeur = 512, int  
        hauteur = 512 ) ;  
        ~ Matrice () ;  
  
};
```

```
Matrice :: Matrice (int l, int h) :  
    width(l), height(h){  
    // allocation  
    tab = new char* [width];  
    for (int i= 0; i < width;i++)  
        tab[i]= new char [height];  
    // initialisation  
    for (int y= 0; y < height; y++)  
        for (int x= 0; x < width; x++)  
            tab[x][y] = '1';  
}  
Matrice :: ~ Matrice () {  
    for (int i= 0; i < width;i++)  
        delete [] tab[i];  
    delete [] tab;  
};
```

## Les pointeurs intelligents

- Allocation/libération de mémoire à la charge du programmeur
  - Oubli de libération
  - Fuites de mémoires
  - Sémantique d'adresse rendant les copies dangereuses
- Problèmes évités en java ( sémantique d'adresse uniquement avec gestion automatique), mais gaspillage de ressources
- En C++, allocation et libération peuvent être encapsulées dans une classe appelée **pointeur intelligent qui vérifie l'idiome RAI**

## L'idiome RAI

- *Resource Acquisition Is Initialisation*
- Propriété des objets automatiques (ou statiques)
  - Constructeur appelé automatiquement à la déclaration de l'objet
  - Destructeur appelé automatiquement à la sortie du bloc
- Classe vérifiant l'idiome RAI
  - Allouer une ressource dans le constructeur
  - Libérer cette ressource dans le destructeur

```
class OpenDB {  
    Database & db;  
    public :  
        OpenDB (Database & base) : db(base) { db.open();}  
        ~ OpenDB () {db.close();};  
};
```

## Exceptions et objets dynamiques

- Objets statiques correctement détruits
- Désallocation des objets dynamiques non gérée

```
class Article {
    Fournisseur * MonFournisseur;    //désallocation manuelle

    float prix;
public :
    Article(float p) : prix(p) {
        MonFournisseur = new Fournisseur ;
        if (prix <= 0) throw ErreurPrix(prix);
    }
    ~Article() { delete MonFournisseur ;}
};
```

## Encapsulation par une allocation dynamique

```
class Article {
    Allocation <Fournisseur > MonFournisseur; //désallocation autom.
    float prix;
public :
    Article(float p) : prix(p), MonFournisseur(new Fournisseur()){
        if (prix <= 0) throw ErreurPrix(prix);
    }
    ~Article() { delete MonFournisseur ;}
};

template <typename T> class Allocation {
public :
    Allocation (T* ref) : ObjetPointe (ref) {}
    ~Allocation() { delete ObjetPointe;} // destruction automatique
private :
    T * ObjetPointe ;
};
```

## L'objet auto\_ptr (idem Allocation)

- Encapsule un pointeur
- Garantit sa libération
- Est défini dans la STL

```
#include <memory>
int main()
{
    auto_ptr<Article> p (new Article(0.0));
    p->prixTTC();
}
```

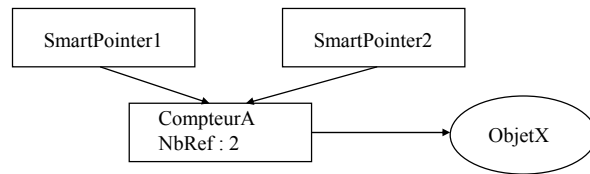
- l'objet pointé est détruit automatiquement à la sortie
- **Limite** : un auto\_pointeur peut pointer sur un seul objet donné !
- solution avec comptage de références (à implanter soi-même)

## Implémentation dans la STL

```
#include <memory> :
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() {delete ptr;}
    T* operator->() {return ptr;}
    T& operator*() {return *ptr;}
    .....
};
```

## Smart pointer

- plusieurs *smart pointers* peuvent pointer sur le même objet
- principe : comptage de références
- gestion automatique de la mémoire (pas de delete)
- contrairement à `auto_ptr` de la STL



## Utilisation des *smart pointer*

```
int main() {
    SmartPointer<Article> p1 = new Article (50); // objet A
    SmartPointer<Article> p2 = new Article (100); // objet B
    SmartPointer<Article> p3; // pointe sur 0
    p1 = 0; // p1 pointe sur 0, A est détruit automatiquement
    p3 = p2; // p2 et p3 pointent sur B
    p2 = 0; // p2 pointe sur 0, B non détruit car pointé par p3
}
// p3 est détruit (var. locale) => B est détruit automatiquement
```

## Opérations sur un *smart pointer*

**p = objet**

incrémente le compteur de références de cet objet

**p = 0**

destruction de p : décrémente le compteur

**p1 = p2**

incrémente le compteur de objet2 et

décrémente le compteur de objet1

l'objet s'auto-détruit quand le compteur arrive à 0

## Une classe Compteur pour le comptage de références

```
class DestructionPointeurInvalide;
class PointeurNull;
template <typename T> class Compteur { // tout est privé
    friend class SmartPointer<T>;
    unsigned int refcount;
    T* Pointeur; // pointeur sur l'objet manipulé

    Compteur (T* pt) : refcount(1), Pointeur(pt) {
        if (!pt) throw PointeurNull();
    }

    void addRef() {refcount++;}

    void removeRef() {
        refcount--; if (refcount == 0) delete this;}
    ~Compteur () {
        if (refcount != 0) throw DestructionPointeurInvalide();
        delete Pointeur;
    }
};
```



## Code d'un smart pointeur

```
template <typename T> class SmartPointer {
    Compteur<T> * MonCompteur;
public:
    SmartPointer(T*);
    ~SmartPointer();
    ....;

    // constructeur copie: SmartPointer <Obj> p = p2;
    template <typename T>
    SmartPointer<T>:: SmartPointer (const SmartPointer & p2) : MonCompteur(p2. MonCompteur) {
        if (MonCompteur) MonCompteur->addRef();
    }

    // destruction.
    template <typename T>
    SmartPointer<T>::~~ SmartPointer() {if (MonCompteur) MonCompteur->removeRef();}

    ///< affectation: p = object;
    template <typename T> SmartPointer<T> &
    SmartPointer<T>:: operator= (const SmartPointer & object)
    {if (MonCompteur) MonCompteur->removeRef(); MonCompteur = object. MonCompteur; if (MonCompteur)
        MonCompteur->addRef(); return *this;}
};
```



## Avantages / Inconvénients

- mémoire libérée automatiquement sans jamais faire delete
- compatible avec toute classe possédant addRef() et removeRef()
- nécessite un compteur de références pour chaque objet
- ne marche pas s'il y a des cycles
- n'est censé pointer qu'un objet créé avec new



## La librairie STL

- STL = Standard Template Library
- Librairie de classes et de fonction génériques

### Doc en ligne

<http://www.sgi.com/tech/stl/>



## Dans la STL

### Conteneurs

Gestion d'une collection d'objets  
Implantés comme une classe générique  
Fournissent des fonctions d'accès aux objets

### Itérateurs

Sorte de pointeur repérant la position d'un élément dans un conteneur

### Algothmes

manipulent les données des conteneurs  
tri, échange de données, recopie de séquences génériques  
interagissent avec les conteneurs via les itérateurs



## Conteneurs

### Séquentiels

Les éléments sont ordonnés  
Leur position est indépendante du contenu d'un élément

### Associatifs (notion de paire : clef, valeur)

Les éléments sont ordonnés selon un critère.  
Leur position dépend de la clé d'un élément

### Adaptateurs (comportement du type abstrait)

Ces classes redéfinissent l'interface d'un conteneur.



## Conteneurs séquentiels

- **vector** : tableau dynamique (insertion à la fin)
- **deque** (double ended queue) : tableau dynamique (insertion au début et à la fin)
- **list** : liste doublement chaînée, performante pour des insertions au milieu de la séquence
- **basic\_string**
- **string**
- **wstring**



## Conteneurs associatifs

- Clé et valeur confondues
  - ensemble : **set** (ne conserve pas les doublons)
  - ensemble multiple : **multiset**  
(conserve les doublons)
- Clé et valeur séparées
  - tables associatives : **map**  
ne conservent pas plusieurs éléments dont les clés sont équivalentes
  - tables associatives multiples : **multimap**  
conservent plusieurs éléments dont les clés sont équivalentes  
Opérateur [] n'est plus disponible



## Créer un conteneur

```
vector<T> ();  
vector<T> (size_type n);  
vector<T> (size_type n, const value_type& value);  
vector<T> (InputIterator deb, InputIterator fin);  
map<Cle,Type> ();  
map<Cle,Type,RelationDOrdre> ();
```

La relation d'ordre utilisée par défaut est le foncteur `less<Cle>` qui utilise l'opérateur `<` par défaut.



## Construire une liste avec un constructeur par défaut

```
#include <list>
using namespace std;

int main() {
    list<int> L; //constructeur par défaut construit une liste vide

    cout << L.size();    // retourne 0

    list<int>::iterator it = L.begin();

    cout << *it; // erreur, liste vide

}
```



## Construire un deque en donnant sa taille


```
#include <deque>
using namespace std;

int main() {
    deque<int> D(10); // deque de 10 int (valeur par défaut 0)

    cout << D.size();    // retourne 10

    cout << D[1];        // affiche 0

}
```



## Construire un vecteur avec une taille et une valeur par défaut

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

int main() {
    vector<int> V(10,2); // vecteur de 10 int (default value 2)

    cout << V.size();    // retourne 10

    cout << V[1];        // affiche 2

}
```



## Construire une liste à partir d'un vecteur

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

int main() {
    vector<int> V;

    for (int i=0;i<10;i++)
        V.push_back(i);

    list<int> L( V.begin(), V.end()); //L contient tous les éléments de V

}
```

## Fonctions partagées par tous les conteneurs

Retourne une référence sur le premier élément

**T& front();**

Retourne une référence sur le dernier élément

**T& back();**

Retire l'élément en fin de collection

**pop\_back();**

Insertion d'un élément en fin

**push\_back(T& );**

Nombre d'éléments

**size\_t size() const;**

Vide?

**bool empty() const;**

Supprime toutes les valeurs

**clear()**

## Fonctions partagées par tous les conteneurs

Retourne un itérateur sur le premier élément

**begin();**

Retourne un itérateur sur le dernier élément

**end() ;**

Efface l'élément pointé par l'itérateur et retourne un itérateur sur l'élément suivant

**erase(iterator);**

Efface les éléments entre les 2 itérateurs (extrémités incluses) et retourne un itérateur sur l'élément suivant

**erase(iterator,iterator);**

Ajoute l'élément à la position pointée par l'itérateur (pas de retour)

**insert(iterator,element);**

## Le conteneur *vector*

- Le vecteur gère un tableau interne qui n'est pas toujours plein;
- Taille :
  - Nombre d'éléments actuellement inclus dans le tableau interne
- Capacité :
  - Taille du tableau interne géré par le vecteur. Cette taille est automatiquement mise à jour quand le tableau interne est plein et quand un élément doit être ajouté.

## Capacité

- Par défaut, la capacité d'un vecteur est 0
- Déclaration d'un vecteur de taille 10 initialisé avec une valeur par défaut  
**vector<int> anArray(10);**
- appel du constructeur par défaut pour les éléments



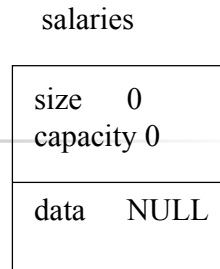


## Exemple

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

```
int main() {
    vector<double> salaries;
    salaries.push_back(3000.0);
    salaries.push_back(2000.0);
}
```

- Le vecteur est créé avec une capacité à 0

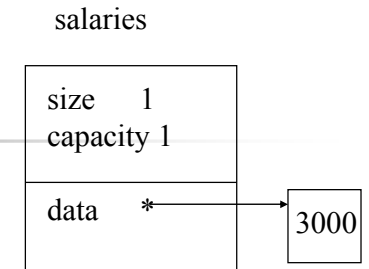


## Exemple

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

```
int main() {
    vector<double> salaries;
    salaries.push_back(3000.0);
    salaries.push_back(2000.0);
}
```

- Un tableau de taille 1 est alloué pour ajouter un élément

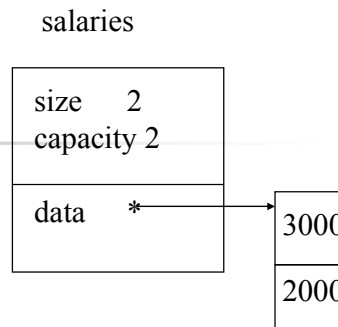


## Exemple

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

```
int main() {
    vector<double> salaries;
    salaries.push_back(3000.0);
    salaries.push_back(2000.0);
}
```

- Comme le tableau est plein, sa taille est doublée pour ajouter un nouvel élément

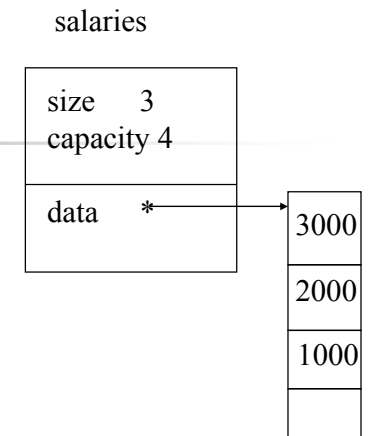


## Exemple

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

```
int main() {
    vector<double> salaries;
    salaries.push_back(3000.0);
    salaries.push_back(2000.0);
    salaries.push_back(1000.0);
}
```

- Comme le tableau est plein, sa taille est doublée pour ajouter un nouvel élément
- Sa capacité est plus grande que sa taille





## Vecteur d'objets

- Avec un vecteur qui gère des objets, l'appel à la fonction `push_back` fera une copie de l'objet.  
`vector<Personne> agenda;`
- Si elle est utilisée fréquemment, cela coûte cher.
- C'est la raison pour laquelle en C++ , on préfère utiliser un vecteur de pointeurs d'objets.



## Vecteur de pointeurs d'objets

- Pour le polymorphisme
- Pour la rapidité

```
vector<Forme*> formes;  
Rectangle R(12,6);  
Forme * ptrCarre = new Carre(2);  
  
formes.push_back(&R);  
formes.push_back(ptrCarre);
```



## Les conteneurs possèdent leurs objets

Les éléments inclus devraient proposer les fonctions suivantes:

- copy constructor
- operator=
- destructor
- default constructor
- operator==
- operator <

Destruction du conteneur

- ⇒ destruction de ses objets ou des pointeurs selon le cas,  
mais pas des objets pointés