C++ avancé

Design patterns et STL

Aurélien Géron, vendredi 2 décembre 2005

Sommaire

- Quelques rappels techniques
- Les design patterns
- La STL

Sommaire

- **Quelques rappels techniques**
  - Templates
  - Namespaces
- Les design patterns
- La STL

Sommaire

- **Quelques rappels techniques**
  - **Templates**
  - Namespaces
- Les design patterns
- La STL

Une classe Liste

- Faut-il écrire une classe pour la gestion de listes d'entiers, une pour les listes de chaînes de caractères, une pour…?
- Ce serait pénible : il faudrait écrire *n* fois les mêmes algorithmes (tri, ajout…)
- La solution habituelle en programmation objet serait d'avoir une classe de base `Objet` et une classe `Liste` qui gèrerait un tableau de pointeurs d'`Objet`

Classe Liste « classique »

```
class Objet { … }
class Entier : public Objet { … };
class Chaine : public Objet { … };
class Liste {
    private: Objet * tab;
    public: void ajout(Objet * obj);
    …
};

int main () {
  Liste maListe;
  maListe.ajout(new Entier(5));
  maListe.ajout(new Chaine("Bonjour"));
}
```

Implémenter le tri

- Il faut déclarer la fonction abstraite suivante dans la classe `Objet` :
  ```
  virtual bool operator < (const Objet & obj) = 0
  ```
- Les classes dérivées doivent l'implémenter
  - Elles doivent pour cela convertir le paramètre `obj` vers un type correct, exemple :
    ```
    const Chaine & e =
        dynamic_cast<const Chaine &> (obj);
    ```
- La classe Liste peut maintenant comparer entre eux les éléments de la liste, et du coup les trier correctement
- Bien entendu, cela ne fonctionnera que si les éléments sont comparables

Code générique dynamique

- L'exemple précédent montre qu'il est possible d'écrire du code générique (c'est-à-dire évolutif) en programmation objet
  - Il suffit d'utiliser l'héritage et le polymorphisme
- Il s'agit d'une solution gérée à l'exécution : la performance est moins élevée qu'en écrivant *n* classes de gestion de listes
- Rien n'empêche de mettre dans la même liste des objets non comparables

Code générique statique

- Les templates permettent d'écrire du code générique en reposant sur un mécanisme statique plutôt que dynamique
- La compilation est plus lente, mais l'exécution est plus rapide
- En outre, les templates peuvent manipuler indifféremment les types primitifs et les types objet

Avec le précompilateur ?

- Avant l'invention des templates :

```
#define DefinirListe (typeDesElements)  \
class Liste_##typeDesElements {         \
   typeDesElements * tab;
     \
   …                                    \
}
…
DefinirListe(int);
DefinirListe(Chaine);
Liste_int maListeEntiers;
Liste_Chaine maListeDeChaines;
```

Avec les templates

```
template <class T> class Liste {
  private:
    T * tab;
  public:
    void ajout(const T & elem);
    void tri();
    T & operator[] (int index);
    …
};
```

Avec les templates (suite)

```
template <class T>
void Liste<T>::tri(const T & elem) {
    …
    if ((*this)[i] < elem) { … }
    …
}
…
Liste<int> maListeEntiers;
Liste<Chaine> maListeDeChaines;
```

Spécialisation de templates

- Si l’on souhaite gérer différemment une Liste de Chaines (par exemple) :

```
template <class T> class Liste { … };

template class Liste<Chaine> { … };
```

Fonction template

- On peut définir des fonctions template :

```
template <class X>
X & max (X & a, X & b) {
  if (a < b) return b; else return a;
}
```

- Cette fonction template peut être utilisée ainsi :

```
int a = max(2,3);
Chaine s = max(chaine1, chaine2);
```

Templates « avancés »

```
class Vecteur {
  int * tab;
 public:
  template <class F> transformer(const F & f) {
   for (int i=0; i < len; i++) f.modif(tab[i]);
  }
};

struct Incrementeur {
  template <class T> void modif(T & t) { t++; }
};

int main() {
  Vecteur v(3);  v[0]=1;  v[1]=4;  v[2]=9;
  Incrementeur incr;
  v.transformer(incr); // =>  v={2,5,10}
}
```

Avantages des templates

- Programmation générique
- Possibilité de gérer les types primitifs
- Performance élevée
- Possibilité de débogage
  - Contrairement à la solution du précompilateur
- Liste<int> ne peut pas contenir de Chaines
  - C’est un avantage ou un inconvénient

Inconvénients des templates

- Assez difficiles à programmer
  - Mais généralement assez simples à utiliser
- Compilation longue
- Ce que le type passé en paramètre doit implémenter est parfois difficile à cerner :
  - Leur faut-il implémenter l'opérateur <, l'opérateur [], la fonction-membre afficher()…?
  - Il faut toujours qu'une documentation claire accompagne un template
- Le code source doit être disponible

Inconvénients (suite)

- Les classes générées à partir d'un même template n'ont aucun lien entre elles :

```
void uneFonction(Liste<const int> & liste) { … }
…
Liste<int> uneListe;
uneFonction(uneListe); // ERREUR DE COMPILATION !

Chaine * ptrChn = new Chaine("ABCD");
Objet * ptrObj = ptrChn; // OK

Liste<Chaine> * ptrChn = …;
Liste<Objet> * ptrObj = ptrChn; // ERREUR COMPIL. !
```

Sommaire

- **Quelques rappels techniques**
  - Templates
  - **Namespaces**
- Les design patterns
- La STL

Les conflits de nom

- Avec des noms comme « Date », « Object » ou encore « String », il y a un risque important de conflit
- Une solution, préfixer toutes les classes par un code (exemple : CString, CObject dans les MFC)
- Soit le préfixe est trop court et le risque de conflit existe encore, soit il est trop long et il devient pénible à écrire

Les namespaces

```
namespace SocieteX {
   namespace Interface_Graphique {
      class Objet { ... };
      class ZoneEdition: public Objet { ... };
   }
}

namespace SocieteY {
   class Objet { ... };
}
```

Utiliser un namespace

- En écrivant le nom complet :

```
SocieteX::Interface_Graphique::ZoneEdition za;
```

- Pour simplifier, on peut utiliser un alias local :

```
namespace SocieteX::Interface_Graphique = IG;
IG::ZoneEdition za;
```

Utiliser un namespace (suite)

- Avec la directive using :

```
using namespace SocieteX::Interface_Graphique;
ZoneEdition za;
```

- Avec la déclaration using :

```
using SocieteX::Interface_Graphique::ZoneEdition;
ZoneEdition za;
```

Templates et namespaces

- Pour utiliser un template dans un namespace :

```
using outils::Liste;
Liste<float> maListe;

using namespace outils;
Liste<float> maListe;

outils::Liste<float> maListe;

typedef outils::Liste<float> ListeFloat;
ListeFloat maListe;

 using outils::Liste<float>;  // ERREUR DE COMPILATION
```

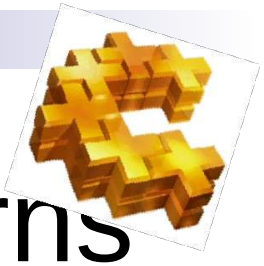
Sommaire

- Quelques rappels techniques
- **Les design patterns**
  - Introduction aux design patterns
  - Les principaux design patterns
- La STL

Sommaire

- Quelques rappels techniques
- **Les design patterns**
  - **Introduction aux design patterns**
  - Les principaux design patterns
- La STL

Introduction aux design patterns



- Dans le monde procédural, il existe depuis longtemps des algorithmes de référence
- Dans le monde objet, des modèles d'architectures et de conception ont commencé à être formalisés à la fin du XXème siècle
- Le livre « Design Patterns » du « Gang of Four » (GoF) est une référence

Design patterns et généricité

- Les design patterns sont conçus pour être utilisés dans n'importe quel langage objet
- Leur description fait généralement usage de diagrammes de classes UML
- L'héritage et le polymorphisme sont largement utilisés
- Cependant, bien que cela demande un peu plus de réflexion, rien n'empêche de mettre en œuvre les design patterns en utilisant les templates

Sommaire

- Quelques rappels techniques
- **Les design patterns**
  - Introduction aux design patterns
  - **Les principaux design patterns**
- La STL

Conteneur / Itérateur

- Une multitude d'objets peuvent être rangés dans divers conteneurs :
  - Listes, Matrices, Dictionnaires…
- On souhaite écrire des algorithmes qui fonctionneront avec tous les conteneurs
- Il nous faut un mécanisme unique pour parcourir les éléments d'un conteneur, quel qu'il soit : c'est le rôle des objets itérateurs

Stratégie

- Un objet stratégie représente une fonction
- Un algorithme générique peut être paramétré par des stratégies, exemple :

```
struct OrdreAscendant {
  template <class T> int comparer
    (const T & a, const T & b) { return a-b; }
};
struct OrdreDescendant {
  template <class T> int comparer
    (const T & a, const T & b) { return b-a; }
};
…
OrdreDescendant desc;
maListe.trier(desc);
```

Adaptateur

- Un adaptateur est un objet qui « entoure » un autre objet pour en modifier l'interface
- Par exemple :

```
class Vecteur { int * tab; … };
class Matrice {
  Vecteur v;
 public:
  void int & element(int x, int y) {
    return v[y*largeur+x];
  }
  …
};
```

Composite

- Un composite est un objet qui peut contenir d'autres objets, dont éventuellement d'autres composites
- Ceci permet de représenter des structures arborescentes
- L'implémentation classique ressemble à :

```
class Element { … };
class Composite : public Element {
  Element* tableauElements; …
};
```

Fabrique

- Un objet fabrique a pour but de créer des instances d'une autre classe
- Exemple : un programme doit régulièrement créer de nouvelles connexions à une base de données
- On lui fournit un objet FabriqueCnx dont le rôle est de fabriquer des objets CnxBase
- Si l'on change de base de données, il suffira de fournir une nouvelle fabrique au programme

Singleton

- Il arrive fréquemment que certains objets soient uniques au sein d'une application
  - Connexion base de données, objet de configuration…
- Pour éviter de définir des variables globales et pour offrir un peu plus de flexibilité au programme, on peut définir une classe qui ne peut avoir qu'une seule instance : il s'agit alors d'un Singleton
- Exemple :

```
class Config {
  protected: Config() { … }  // constructeur protégé
  public:
    static Config & instance() { return instance; }
    static Config instance;
    …
};
…
Chaine s = Config.instance().parametre("host");
```

Observateur

- Il est fréquent de devoir gérer des évènements dans une application
- Un observateur est un objet qui doit être « prévenu » lorsqu'un évènement a lieu
- Il faut auparavant l'enregistrer auprès de l'objet qui est susceptible de déclencher l'évènement

Observateur : exemple

```
class ObservateurDeBouton {
  public:
    virtual void clic(bool estDouble) = 0;
};

class Bouton { …
    ObservateurDeBouton * tableauObservateurs;
    void declencherEvenement();
  public:
    void enregistrer (const Observateur & obs) {…}
};
```

Observateur : exemple (suite)

```
class Afficheur : public ObservateurDeBouton {
  public:
    virtual void clic(bool estDouble) {
      if (estDouble) afficherMessage("Clic");
    }
};

int main() {
  Afficheur monAfficheur;
  Bouton monBouton;
  monBouton.enregistrer(monAfficheur);
}
```

Proxy

- Lorsque l’on veut communiquer avec un objet distant, une technique consiste à communiquer avec un objet « proxy » local, qui s’occupe de relayer les requêtes
- L’objet proxy possède la même interface que l’objet distant, ce qui facilite la programmation du logiciel client

Visiteur

- Lorsque l’on souhaite appliquer un algorithme à un graphe d’objets (généralement une arborescence), le plus simple est souvent de fournir un objet « visiteur » à l’un des nœuds du graphe
- Les nœuds doivent être programmés pour propager les visiteurs dans tout le graphe
- L’objet visiteur applique l’algorithme souhaité au fur et à mesure qu’on lui fait parcourir le graphe

Visiteur : exemple

```
class Fichier;
class VisiteurDeFichier { public:
  virtual void visiter(Fichier & fichier) = 0;
};

class Fichier { … public:
  virtual void propager(VisiteurDeFichier & v) {
      v.visiter(*this);
  }
};

class Répertoire : public Fichier { … };
// propager est redéfinie pour appeler propager sur
// chacun des fichiers du répertoire
```

Visiteur : exemple (suite)

```
class TrouveFichiers : public VisiteurDeFichier {
  Fichier * tableauFichiersTrouves;
  Chaine critereRecherche;
 public:
  virtual void visiter(Fichier & fichier) {
    if (fichier.filtre(critereRecherche))
                    ajouteFichierTrouve(fichier);
  }
};
…
TrouveFichiers trouveFichiersTexte("*.txt");
Repertoire rep ("C:\programmation\");
rep.propager(trouveFichiersTexte);
```

Façade

- Lorsqu'un module d'une application a de multiples fonctionnalités, mises en œuvre par de nombreux objets, il vaut mieux éviter que les autres modules puissent accéder à tous ces objets
  - surtout s'il s'agit d'une application distribuée !
- Il est préférable de définir un petit nombre d'objets (la façade) pour l'interface avec le reste de l'application
  - Un peu comme un serveur de restaurant qui prend les commandes et les transmet ensuite en cuisine

Sommaire

- Quelques rappels techniques
- Les design patterns
- **La STL**
  - Objectifs de la STL
  - Les conteneurs, itérateurs et adaptateurs
  - Algorithmes et stratégies génériques
  - Gestion de la mémoire

Sommaire

- Quelques rappels techniques
- Les design patterns
- **La STL**
  - **Objectifs de la STL**
  - Les conteneurs, itérateurs et adaptateurs
  - Algorithmes et stratégies génériques
  - Gestion de la mémoire

Bref historique

- Le père de la STL est Alexandre Stepanov, associé à Ming Lee
- Il a commencé en 1979 en créant une bibliothèque semblable en bien des points à la STL, mais destinée au langage Ada
- Ada propose une notion équivalente aux templates de C++ : les *generics*
- Le langage Ada ne s'est pas développé autant que C++, en restant cantonné à des domaines tels que la défense ou les logiciels de sécurité
- Stepanov s'est donc tourné vers C++ pour mettre en œuvre ses idées

La standardisation

- C++ était avant tout un langage, à l'origine avec très peu de librairies
- La STL fut intégrée au draft standard en juillet 1994 et est aujourd'hui une partie essentielle du standard ANSI-ISO de C++
- Il est assez étonnant de constater qu'alors que le comité de standardisation était en général plutôt précautionneux, il aura accepté la STL un an seulement après sa proposition

Les objectifs

- Des conteneurs et itérateurs performants qui gèrent leur propre mémoire
- Des algorithmes à appliquer aux conteneurs
- Quelques outils variés, notamment pour la gestion de la mémoire automatique

Une classe sans STL

```
class Individu {
   char * nom;
   void nouveauNom (const char * p_nom) {
      nom = new char[strlen(p_nom) + 1];
      strcpy (nom, p_nom);
   }
 public:
   Individu (const char * p_nom) {
      nouveauNom (p_nom);
   }
   Individu (const Individu & ind) {
      nouveauNom (ind.p_nom);
   }
   Individu & operator = (const Individu & ind) {
      if (this != &ind) {
         delete [] nom;
         nouveauNom (p_nom);
      }
      return this;
   }
   ~Individu() { delete [] nom; }
};
```

Avec la STL…

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

class Individu {
   string nom;
 public:
   Individu (const char * p_nom) : nom (p_nom) {}
};
```

- La classe string est définie ainsi :

```
typedef basic_string<char> string;
```

- Elle est très simple à utiliser :

```
string s1 ("cogito");
s1 += " ergo sum";
cout << "Dans ‘" << s1 << "’, le 3ème caractère est : "
     << s1[2] << endl;
```

Le template basic_string

- Voici un extrait du template basic_string dans le fichier <string> :

```
template<class E, class T = char_traits<E>,
                  class A = allocator<T> >
class basic_string {
public:
   typedef T traits_type;
   typedef T::char_type char_type;
   typedef A::size_type size_type;
   typedef A::pointer pointer;
   typedef A::reference reference;
   typedef A::pointer iterator;
   …
```

Fonctions de parcours

```
…
iterator begin();
iterator end();
reference at(size_type pos);
reference operator[](size_type pos);
const E *c_str() const;
const E *data() const;
…
```

Fonctions de capacité

```
…
size_type length() const;
size_type size() const;
size_type max_size() const;
void resize(size_type n, E c = E());
size_type capacity() const;
void reserve(size_type n = 0);
bool empty() const;
…
```

Fonctions de manipulation

```
 …
 basic_string& insert(size_type p, const basic_string& s);
 basic_string& erase(size_type p=0, size_type n = npos);
 size_type find(const E *s, size_type pos, size_type n)…;
 basic_string substr(size_type pos=0, size_type n=npos)…;
 int compare(const basic_string& str) const;
 …

};
```

Sommaire

- Quelques rappels techniques
- Les design patterns
- **La STL**
  - Objectifs de la STL
  - **Les conteneurs, itérateurs et adaptateurs**
  - Algorithmes et stratégies génériques
  - Gestion de la mémoire

Conteneurs de la STL

- Les conteneurs de la STL sont :
  - vector : des tableaux redimensionnables
  - list : des listes chaînées bidirectionnelles
  - deque : des listes chaînées à accès rapide
  - queue et priority_queue : des files d’attente
  - stack : des piles
  - map et multimap : des dictionnaires associatifs
  - set et multiset : des ensembles

Vector

- Simples tableaux redimensionnables :

```
class Voiture { ... };
…
vector <Voiture> garage;
garage.push_back (Voiture ("Lada"));
garage.push_back (Voiture ("Jaguar"));
garage.push_back (Voiture ("Renault"));
for (int i = 0; i < garage.size(); i++)
  { garage[i].demarrer();
}
```

Définition de vector

```
template<class T, class A = allocator<T> > class vector {
public:
   typedef A::reference reference;
   typedef A::pointer iterator; // un simple pointeur

   explicit vector(const A& al = A());
   explicit vector(size_type n, const T& v = T(), const A& al = A());
   vector(const vector& x);

   void reserve(size_type n);
   size_type capacity() const;
   iterator begin();
   iterator end();
   void resize(size_type n, T x = T());
   size_type size() const;
   size_type max_size() const;
   bool empty() const;
   …
```

Définition de vector (suite)

```
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
reference back();
void push_back(const T& x);
void pop_back();
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
iterator erase(iterator it);
void clear();
void swap(vector x);
```

Parcours sans itérateur

- L’algorithme suivant ne fonctionne qu’avec les conteneurs qui possèdent l’opérateur [] :

```
vector <Voiture> garage;
garage.push_back (Voiture ("Lada"));
garage.push_back (Voiture ("Jaguar"));
garage.push_back (Voiture ("Renault"));
for (int i = 0; i < garage.size(); i++)
  garage[i].demarrer();
```

Parcours avec itérateur

- L’algorithme suivant fonctionnera avec tous les conteneurs : il est « générique »

```
vector <Voiture> garage;
// ... on remplit le vecteur
vector <Voiture>::iterator it
       = garage.begin();
for ( ; it != garage.end(); it++)
       it->demarrer();
```

Validité des itérateurs

- Comme pour les pointeurs classiques, il faut faire attention à ne pas utiliser un itérateur qui « pointe » vers un objet qui n'existe plus
- Dans le cas d'un vector, tant qu'on ne dépasse pas la capacité, il n'y a pas de réallocation, et les itérateurs restent valides

Le template list

- Les list sont des listes chaînées bidirectionnelles :
  - Chaque élément pointe vers le suivant et le précédent
- Les insertions et suppressions prennent un temps constant
- En revanche, trouver le $n^{ème}$ élément d'une liste prend un temps proportionnel à n
  - Il n'y a pas de fonction d'accès direct à l'élément n (pas d'opérateur [] ni de fonction `at`)
- Un itérateur de list reste toujours valide, tant que l'élément qu'il pointe n'est pas effacé

Exemple d’utilisation de list

```
class Produit {  ...  };
...
typedef list<Produit> ListeProduits;
ListeProduits mesCourses;
mesCourses.push_back (Produit("Vin",45));
mesCourses.push_back (Produit("Chocolat",19.90));

ListeProduits::iterator it = mesCourses.begin();
for ( ; it != mesCourses.end(); it++) { it->acheter(); }

cout << "J'enlève le produit suivant : ";
mesCourses.front().afficher();
mesCourses.pop_front();

cout << "Je range tout...  ";
mesCourses.sort();

cout << "...et j’inverse le sens de la liste !" << endl;
mesCourses.reverse();
```

Quelques fonctions de list

- **splice** : transférer le contenu un conteneur dans la liste
- **sort** : trier la liste en se basant sur l'operator < des éléments
  - L'algorithme est stable car les éléments ne sont pas déplacés en mémoire ; environ N*log(N) comparaisons sont effectuées
- **unique** : remplace toute séquence d'éléments identiques par un seul élément ; en général, on aura trié la liste auparavant
- **merge** : permet de transférer le contenu d'une autre liste dans la liste ; contrairement à splice, cet algorithme est stable
- **reverse** : inverse l'ordre de la liste selon un temps proportionnel à N

Le conteneur deque

- Un conteneur deque est un intermédiaire entre un vector et une list : il gère des tableaux d'éléments chaînés de façon bidirectionnelle
- Ceci permet à la fois des insertions et suppressions rapides, et un accès rapide aux éléments
- L'opérateur [] et la fonction `at` sont définis

Tableau des performances

Fonction	tableau	*vector*	*deque*	*list*
Insertion/destruction au début	Non défini	*Linéaire*	Constant	Constant
Insertion/destruction à la fin	Non défini	Constant	Constant	Constant
Insertion/destruction au milieu	Non défini	*Linéaire*	*Linéaire*	Constant
Accès au premier élément	Constant	Constant	Constant	Constant
Accès au dernier élément	Constant	Constant	Constant	Constant
Accès au éléments du milieu	Constant	Constant	Constant	*Linéaire*

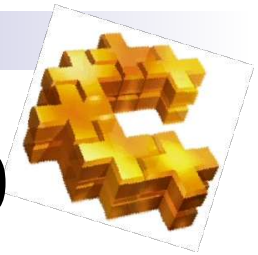
Les map et multimap

- Le conteneur map est un conteneur associatif : il permet de récupérer une *valeur* lorsqu'on lui donne une *clé*
  - Exemple : un dictionnaire
- Les éléments sont rangés en arbre dichotomique, ce qui permet une recherche très rapide, de l'ordre de log(N)
- Les clés donc doivent pouvoir être comparées avec l'opérateur <
- Un multimap est identique à un map, à ceci près qu'il autorise le stockage de plusieurs valeurs ayant des clés identiques

Exemple d’utilisation de map

```
typedef map<double, string> DoubleEnString;

DoubleEnString leMap;

leMap.insert(DoubleEnString::value_type(3.14, "Pi"));
leMap.insert(DoubleEnString::value_type(0.00, "Zéro"));
leMap.insert(DoubleEnString::value_type(0.50, "Un demi"));

double aChercher;
cout << "Tapez un réel connu : ";
cin >> aChercher;

DoubleEnString::iterator iter = leMap.find(aChercher);
if (iter == leMap.end())
   cout << "Inconnu au bataillon" << endl;
else
   cout << (*iter).second << endl;
```

Aperçu de la définition de map



```
template<class Key, class T, class Pred = less<Key>, class
   A = allocator<T> >  class map { public:
    iterator begin();
    iterator end();
    size_type size    () const;
    bool empty() const;
    A::reference operator[](const Key& key);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(map& x);
    key_compare key_comp() const;
    iterator find(const Key& key);
    size_type count(const Key& key) const;
    iterator lower_bound(const Key& key);
    iterator upper_bound(const Key& key);
    pair<iterator, iterator> equal_range(const Key& key);
    pair<iterator, bool> insert(const value_type& x);
    // ...
};
```

Les set et multiset

- Un set est également un conteneur associatif utilisant un algorithme d'arbre dichotomique pour indexer les clés
- Toutefois, il n'y a pas de valeur associée à une clé
- On utilise donc un set comme un ensemble dans lequel on stocke des éléments
- Vérifier la présence d'un élément dans un set prend un temps de l'ordre de log(N)
- Un multiset peut contenir plusieurs fois le même élément

Exemple d’utilisation de set

```
multiset<string> s1;

s1.insert("Jean");
s1.insert("Catherine");
s1.insert("Pierre");
s1.insert("Jean"); // Insertion multiple possible

set<string>::iterator it;
it=s1.find("Jacques");
if (it != s1.end()) cout << "Jacques est là" << endl;

cout << "Nombre de ‘Jean’ : " << s1.count("Jean") << endl;
s1.clear();
if (s1.empty()) cout << "L’ensemble est vidé" << endl;
```

Adaptateurs de conteneurs

- Les trois derniers conteneurs de la STL sont en réalité des adaptateurs des conteneurs précédents
- Par défaut, ils reposent sur un deque, mais il est possible de créer par exemple un stack reposant sur un vector

Définition de stack

```
template <class T, class C = deque<T> >
class stack {
 protected:
   C c;
 public:
   bool empty() const { return (c.empty()); }
   value_type& top() { return (c.back()); }
   void push (const value& v) { c.push_back(v); }
   void pop() { c.pop_back(); }
   …
};
```

Exemple d’utilisation de stack

```
stack<int, list<int> > unePile;
unePile.push (2);
unePile.push (5);

while (!unePile.empty()) {
   cout << unePile.top() << endl;
   unePile.pop();
}
```

Définition de queue

```
template <class T, class C = deque<T> >
class queue {
 protected:
   C c;
 public:
   bool empty() const { return (c.empty()); }
   value_type& top() { return (c.back()); }
   void push (const value& v) { c.push_front(v); }
   void pop() { c.pop_back(); }
   …
};
```

Exemple d’utilisation de queue

```
queue<int> uneFile;
uneFile.push (2);
uneFile.push (5);

while (!uneFile.empty()) {
   cout << uneFile.top() << endl;
   uneFile.pop();
}
```

Le conteneur priority_queue

- Contrairement à stack et queue, le conteneur priority_queue “ajoute un peu de valeur” au conteneur sous-jacent : il ajoute les éléments dans la liste en fonction de leur priorité
- Celle-ci est géré grâce à un objet stratégie qui par défaut est le template `Less`

Définition de priority_queue

```
template <class T, class C = vector<T>,
          class P = less<C::value_type> >
class priority_queue {
 protected:
   C c;
   P p;
 public:
   explicit priority_queue(const P & _p = P(),…);
   bool empty() const;
   value_type& top();
   void push (const value& X);
   void pop();
   …
};
```

Des requêtes priorisées

```
class Requete {
private:
   int prio;
   string mes;
public:
   Requete (const char * m = "", int p = 0):
                                mes(m), prio(p) {}
   bool operator < (const Requete & req) const {
      return prio < req.prio;
   }
   bool operator == (const Requete & req) const {
      return prio == req.prio;
   }
   void traiter() { cout << mes << endl; }
};
```

Des requêtes priorisées

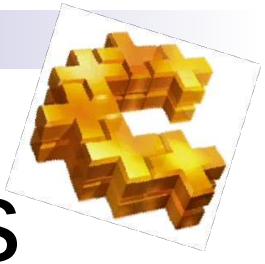
```
void main () {
   priority_queue<Requete> file;
   file.push(Requete("Dormir", 2));
   file.push(Requete("Travailler", 5));
   file.push(Requete("Lire", 4));

   while (!file.empty()) {
      file.top().traiter();
      file.pop();
   }
}
```

Les adaptateurs d’itérateurs

- Les adaptateurs d’itérateurs permettent d’obtenir des itérateurs à sens de parcours inversé, des itérateurs vers objets constant, etc.
- Exemple :

```
vector<int> v;
// ... on remplit le vecteur
vector<int>::reverse_iterator rt = v.rbegin();
while (rt != v.rend())
  cout << *rt << endl; rit++;
```

Autres adaptateurs d’itérateurs



- back_insert_iterator, front_insert_iterator et insert_iterator sont d’autres adaptateurs d’itérateurs définis dans la STL
- Comme leur nom l’indique, ils permettent de parcourir un conteneur en insérant au passage des éléments
- On peut les obtenir en appelant l’une des fonctions template suivantes :
  - back_inserter(cont), front_inserter(cont) ou encore inserter(cont, iter)

L’itérateur istream

- L’itérateur istream permet de parcourir un flux (exemple un fichier) comme un conteneur séquentiel. Par exemple :

```
ifstream fichier ("C:\\Data\\donnees.dat", ios::in);
if (fichier.good()) {
   list<int> donnees;
   istream_iterator<int> it (fichier);
   copy (it, istream_iterator<int>(),
         back_inserter(donnees));
}
```

Sommaire

- Quelques rappels techniques
- Les design patterns
- **La STL**
  - Objectifs de la STL
  - Les conteneurs, itérateurs et adaptateurs
  - **Algorithmes et stratégies génériques**
  - Gestion de la mémoire

Algorithmes génériques

- Comme nous venons de le voir avec la fonction template copy, la STL définit des fonctions génériques applicables à divers conteneurs
- Un exemple de fonction générique que l'on peut définir soi-même :

```
template <class Conteneur>
void afficher (const Conteneur & cont) {
   Conteneur::const_iterator it = cont.begin();
   for ( ; it != cont.end(); it++)
        cout << *it << endl;
}
```

Interface d’un conteneur

- Un conteneur définit toujours : typedef value_type, reference, const_reference, iterator, const_iterator, difference_type, size_type, reverse_iterator et const_reverse_iterator
- Il définit également toujours un constructeur par défaut, un constructeur par copie (ceci implique que les éléments peuvent également être copiés) et un destructeur
- Il définit toujours les fonctions begin, end, rbegin, rend, size, max_size, empty, swap
- Si c1 et c2 sont deux conteneurs (de la même classe), on peut toujours écrire c1==c2, c1!=c2, c1<c2, c1>c2, c1<=c2, c1>=c2, c1=c2

Interface d’un conteneur

- Un conteneur définit toujours : typedef value_type, reference, const_reference, iterator, const_iterator, difference_type, size_type, reverse_iterator et const_reverse_iterator
- Il définit également toujours un constructeur par défaut, un constructeur par copie (ceci implique que les éléments peuvent également être copiés) et un destructeur
- Il définit toujours les fonctions begin, end, rbegin, rend, size, max_size, empty, swap
- Si c1 et c2 sont deux conteneurs (de la même classe), on peut toujours écrire c1==c2, c1!=c2, c1<c2, c1>c2, c1<=c2, c1>=c2, c1=c2

Interface d’un conteneur

- Les conteneurs séquentiels disposent en outre des constructeurs suivants :
  - Cont(n, v) : constructeur d’un conteneur avec n copies de v
  - Cont(it1,it2) : constructeur à partir de l’intervalle [i1, i2[
- De plus, ils ont tous au moins les fonctions d’insertion suivantes :
  - insert(it, v) : insertion d’une valeur v à la position de l’itérateur it
  - insert(it, n, v) : insertion de n copies de v à la position de l’itérateur it
  - insert(it, it1, it2) : insertion à la position it des valeurs entre it1 et it2
- Enfin, ils définissent les fonctions d’effacement suivantes :
  - erase(it) : destruction de la valeur « pointée » par l’itérateur it
  - erase(it1, it2) : destruction de toutes les valeurs entre it1 et it2

Conteneurs associatifs

- Les conteneurs associatifs définissent également des types supplémentaires :
  - key_type
  - key_compare
  - value_compare

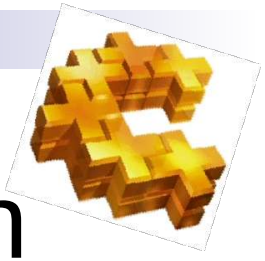
Interface des itérateurs

- input_iterator : par exemple, l'itérateur istream_iterator
  - ils permettent la consultation des éléments d'un conteneur
  - Il est uniquement possible d'incrémenter l'itérateur d'une unité comme dans it++ et de lire la valeur « pointée » avec *it et it->..
- output_iterator : par exemple, l'itérateur ostream_iterator
  - Idem mais ils ne permettent que la modification des éléments
- forward_iterator : par exemple raw_storage_iterator
  - Mélange de input_iterator et output_iterator
- bidirectional_iterator : itérateurs des list, set, multiset, map et multimap
  - Idem avec en plus la possibilité de décrémenter l'itérateur avec it--
- random_access_iterator : itérateurs des tableaux C++, vector et deque
  - Les plus complets : ils permettent la même chose que les bidirectional_iterator, avec en plus la possibilité de déplacements directs de plusieurs unités : it -= 5 ou encore it[3]

Algorithmes non-mutants

- Dans la STL, un algorithme non-mutant est un algorithme qui ne modifie pas les données du conteneur qu'il manipule
- Quelques algorithmes non-mutants:
  - **for_each** : permet d'appliquer un traitement (non mutant) à tous les éléments
  - **find** : permet de rechercher une valeur
  - **adjacent_find** : recherche deux valeurs consécutives égales
  - **count** : compte les valeurs égales à une valeur donnée
  - **mismatch** : détecte des valeurs différentes
  - **equal** : détermine si deux conteneurs sont égaux en comparant leur contenu
  - **search** : recherche une séquence d'éléments

Exemple d’utilisation de for_each



- La fonction template for_each est définie de la façon suivante :

```
template<class InIt, class Fct>
inline Fct for_each(InIt first, InIt last, Fct apply);
```

- Le paramètre apply est le nom de la fonction à appliquer (en fait, c’est un pointeur sur une fonction) :

```
void afficher (int a) { cout << a << endl; }
…
vector<int> v;
// ... on remplit le vecteur
for_each (v.begin(), v.end(), afficher);
```

Algorithmes mutants

- **copy**, recopie le contenu d'un intervalle dans un conteneur
- **copy_backward**, fait la même chose mais en parcourant l'intervalle à l'envers
- **swap**, échange le contenu de deux conteneurs
- **transform**, modifie les valeurs d'un conteneur
- **replace**, remplace les valeurs d'un conteneur
- **fill**, remplit le conteneur avec une valeur donnée
- **generate**, produit une suite de valeurs dans un conteneur résultant de l'application d'une fonction
- **remove**, suppression des valeurs qui correspondent à un critère
- **unique**, élimination des éléments uniques d'un conteneur trié
- **reverse**, inversion des valeurs d'un conteneur par rapport à un pivot
- **rotate**, rotation des valeurs d'un conteneur
- **random_shuffle**, distribue uniformément les valeurs d'un conteneur
- **partition**, partition des valeurs d'un conteneur vis-à-vis d'un pivot

Tri et fusion

- **sort**, tri les valeurs d'un conteneur
- **nth_element**, place un élément à la position qu'il aurait si la séquence était triée
- **binary_search**, recherche binaire dans un conteneur trié
- **lower_bound**, recherche d'une borne inférieure pour les valeurs d'un conteneur répondant à un critère donné
- **upper_bound**, idem, mais avec une borne supérieure
- **equal_range**, recherche les zones d'égalité
- **merge**, fusionne des séquences triées
- **set_xxx**, algorithmes ensemblistes tels que set_union,...
- **xxx_heap**, algorithmes sur des tas tels que push_heap, make_heap,...
- **min**, **max**, **minimum** et **maximum** de 2 ou 3 valeurs
- **median**, **mediane** de 2 ou 3 valeurs
- **lexicographical_compare**, comparaison lexicographique
- **next_permutation** et **prev_permutation**, permettent de permuter des séquences

Calculs sur le contenu

- Dans <numeric>, on trouve des algorithme permettant de réaliser des calculs sur les éléments d'un conteneur :
  - **accumulate**, accumulation de données dans une variable
  - **inner_product**, produit intérieur de conteneurs
  - **partial_sum**, somme partielle des valeurs d'un conteneur
  - **adjacent_difference**, différence entre deux éléments adjacents

Utilisation d'accumulate

- Définition de la fonction `accumulate` :

```
template<class InIt, class T, class Pred>
T accumulate(InIt first, InIt last, T val, Pred pr);
```

- On peut l'utiliser de la façon suivante :

```
vector<float> v;
v.push_back (3.14);
v.push_back (5);
// ...
float resultat = accumulate
    (v.begin(), v.end(), 1.0, times<float>);

cout << "Le résultat du produit est : "
     << resultat << endl;
```

Sommaire

- Quelques rappels techniques
- Les design patterns
- **La STL**
  - Objectifs de la STL
  - Les conteneurs, itérateurs et adaptateurs
  - Algorithmes et stratégies génériques
  - **Gestion de la mémoire**

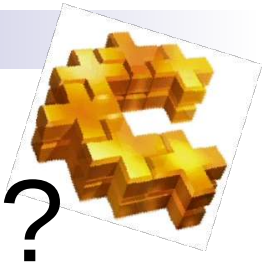
L’héritage et les conteneurs

- Les conteneurs ne gèrent pas l’héritage :

```
class Animal { public:
   virtual void afficher() const {
     cout << "Animal" << endl; }
}
class Chien { public:
   virtual void afficher() const {
     cout << "Chien" << endl; }
}
vector<Animal> zoo(10);
zoo[0] = Chien("Droopy");
zoo[0].afficher();
```

- Ce code affichera malheureusement « Animal » et non « Chien » !

Des conteneurs de pointeurs ?



- Une solution possible :

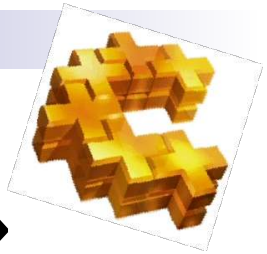
```
vector<Animal *> v(2);
v[0] = new Chien("Snoopy");
v[0]->afficher();
```

- Affiche bien « Chien ». Malheureusement, pour copier un objet, rien n'est prévu : il faut implémenter une fonction de clonage :

```
v[1] = v[0].clone();
v.push_back(v[1]);
…
delete v[0];
delete v[1];
```

- Il faut gérer soi-même la mémoire !

Un conteneur « polymorphe »



- On peut définir soi-même un conteneur polymorphe :

```
template <class Elem>
class VecteurPoly {
   vector<Elem *> v;
   … // gérer la copie, la libération, etc.
};
```

- Ce template pourra ensuite être utilisé simplement :

```
VecteurPoly<Animal> v(2);
v[0] = Chien("Snoopy"); // appellera la fonction clone()
v[0].afficher(); // affiche « Chien » !
v[1] = v[0]; // l'objet v[0] est bien copié avec clone()
v.push_back(v[1]); // le conteneur s'agrandit tout seul,
                   // et v[1] est copié avec clone()
```

Les « smart pointers »

- Un « pointeur intelligent » est un objet qui référence un autre objet (un peu comme un itérateur) et exécute des traitements particuliers
- Exemple, on peut définir un smart pointer de clonage pour résoudre le problème précédent :

```
vector<SmartPointerClonage<Animal> > v(2);
v[0] = new Chien("Snoopy");
v[0]->afficher(); // affiche « Chien »
v[1] = v[0]; // l'objet v[0] est cloné
v.push_back(v[1]); // v[1] est cloné
```

Comptage de références

```
vector<SmartPointerReference<Animal> > v(2);
v[0] = new Chien("Snoopy");
v[0]->afficher(); // affiche « Chien »
v[1] = v[0];
   // l'objet v[0] n'est pas copié
   // v[1] et v[0] pointe vers le même objet
v.push_back(v[1]);
   // le conteneur s'agrandit tout seul,
   // et v[1] n'est pas copié

// A la destruction du conteneur, les objets
// sont correctement détruits
```

Le type `auto_ptr`

- Dans le fichier <memory>
- Un smart pointer pas très « smart » :
  - un seul auto_ptr est « propriétaire » de l'objet pointé
  - lors de sa destruction, un auto_ptr détruit également l'objet pointé s'il en est le propriétaire
  - lors d'une affectation entre deux auto_ptr, comme ptrA = ptrB, le pointeur affecté (ici, ptrA) devient le propriétaire si l'autre pointeur (ici, ptrB) l'est

```
auto_ptr<Animal> p1 ( new Chien ("Lassie") );
auto_ptr<Animal> p2 = p1;

p1->afficher();
// à la fin, p2 (le propriétaire) détruit l'objet
```

Les limites des `auto_ptr`

- On ne doit pas passer un `auto_ptr` en paramètre d'une fonction par copie
- L'opérateur d'affectation modifie l'objet à copier, car ce dernier n'est plus propriétaire de l'objet pointé
- Or les conteneurs de la STL refusent d'avoir des éléments qui ne peuvent pas être copié d'une façon « normale » : on ne peut donc pas créer de conteneurs d'`auto_ptr` !
- Pour les `auto_ptr`, ainsi que pour la plupart des s.p. avec compteurs de référence, les références circulaires sont interdites

Les allocateurs

- Pour gérer la mémoire dans certains cas (rares)
- Exemple de définition :

```
template <class T>
class allocMemPartagee : public allocator<T> {
public:
   pointer allocate(size_type n, const void *hint) {
      // appel à la fonction d'allocation de mémoire
      // partagée et renvoi d'un pointeur sur la zone
      // allouée
   }
   void deallocate(void * p, size_type n) {
      // appel à la fonction de désallocation de
      // mémoire partagée
   }
};
```

Les allocateurs (suite)

- Le template allocateur défini à la page précédente peut s'utiliser comme ceci :

```
vector <Animal, allocMemPartagee <Animal> > zoo(3);
zoo[0] = Chien("Pif");
zoo[1] = Chat ("Hercule");
```

- Les copies des objets sont allouées dans la mémoire partagée

C++ avancé

Fin de la présentation