Librairies de développement
La Standard Template Library

Yannick Kergosien

Polytech Tours
Université François Rabelais de Tours

Janvier 2016

Sommaire

Développement d'applications complexes

Toute application contient

- Une liste chaînée
- Du traitement de chaîne
- Des algorithmes de tri
- ...

Perte de temps

- Réimplémenter : long, pénible et source d'erreur
- Maintenir : autant de façon d'implémenter une liste chaînée que d'informaticiens

Temps perdu

Cliste

```
1/*****
2 Titre : Clist Classe pour la gestion de liste...
3******
4 Auteur : V. T'kindt
5 Version : 1.0
6 Date : 10/07/2004
7 [...]
```

En chiffres

- ≈ 300 lignes de code (dont commentaires)
- ```
  1 for (iBoucle=0 ;iBoucle<uiLISTaille ; iBoucle++)
  2 pTeLISliste[iBoucle]=objet.pTeLISliste[iBoucle];
  ```
  ```

Temps perdu

Cliste.hpp

```
1// Definition du type d'un element de la liste
2// Il faut modifier le type de base ci-dessous pour
     changer
3// le type des elements de la liste
4typedef int Telement ;
```

- Réutilisabilité ?

Règle générale sur le développement

Sorti de l'exercice pédagogique

- Ne **JAMAIS** réinventer la roue
- Quelqu'un a certainement déjà développé quelque chose qui fait la même chose...
- ...en mieux que ce que vous êtes capables de le faire vous-même (dans le délai qui vous est imparti)
- Attention aux licences !

Programmation générique

- Avantages nombreux
  - ré-utilisabilité,
  - gain de temps,
  - ...
- Passe par l'utilisation de **templates** !
  - notion fortement utilisée dans la STL.

Les templates (patrons)

- L'idée :
  - Écrire une seule fois un code pouvant travailler sur plusieurs types
  - Exemple: un algorithme qui trie un tableau
  - Traitements identiques quelque soit le type des éléments du tableau (int, double, ...)

En pratique

```
1  template <class C> // ou template <typename C>
2  void trie(C* tableau)
3  {
4     ...
5  }
```

Templates : autres exemples

Exemple

```
1    template <typename T>
2    T min (T a, T b) {  return a < b ? a : b; }
```

Instanciation explicite/implicte

```
1 float a, b; ... b = min<float>(a,42);
2 int c, d; ... c = min(c,d);
```

~~min(3,1.2)~~

Templates : autres exemples

Attention

```
1 const char* a = "boo" ;
2 const char* b = min(a, "ba");//Compare les pointeurs !
```

Solution :

```
1  template <> const char* min (const char* a, const
       char* b)
2{ return strcmp (a, b) < 0 ? a : b; }
```

Les templates

Avantages

- Code générique
- Vous gagnez du temps (code à n'écrire qu'une seule fois)
- Maintenance facilitée
- Le type d'un patron peut aussi bien être un type simple qu'une classe, ou encore qu'une fonction

Les patrons de conception

- Design Patterns
- Plusieurs définition :
  - “arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d’un logiciel”
  - “des propositions de conception informatiques largement éprouvées dans la résolution de problèmes génie logiciel types”
  - ...
- De nombreux types : cf. le Web.
- Nous verrons quelques exemples : itérateur, MVC et singleton.

Singleton

- Objectif : Assure qu'une classe n'est instanciée qu'une seule et unique fois.
- Exemple d'utilisation : Connexion de BdD, synchronisation, simulateur-scheduler, etc.

Singleton

Exemple

```
1 class Singleton
2 {
3     private:
4         Singleton();
5         ~Singleton();
6         static Singleton    *instance;
7     public:
8         static Singleton* getInstance() {
9             if(instance == NULL)
10                instance = new Singleton();
11            return instance;
12        }
```

Singleton

Exemple

```
1           static void kill() {
2               if(instance != NULL) {
3                   delete instance;
4                   instance = NULL;
5               }
6           }
7};
8//Dans le cpp: Singleton *Singleton::instance = NULL;
9...
10Singleton    t = Singleton::getInstance(); //
      Utilisation
```

Les espaces de nommage

- Regroupe plusieurs déclarations de variables, fonctions et classes dans un groupe nommé.
- Evite les collisions de noms (un même nom peut être déclaré dans des espaces de noms différents).

Exemple

```
1 namespace exemple
2{
3...//Déclarations des fonctions, variables et classes
4     int suivant(int n) {
5          return n+1;
6     }
7}
```

Utilisation des espaces de nommage

Exemple

```
1int a = exemple::suivant(5);
```

Ou

```
1using exemple::suivant;
2...
3int a = suivant(5);
```

Ou

```
1using namespace exemple;
2...
3int a = suivant(5);
```

Sommaire

La STL

- La STL (Standard Template Library) est une librairie générique qui fournit (presque):
  - toutes les structures de données
  - tous les algorithmes

  de bases en informatique.

Nouveau formalisme

- Définit de nombreux nouveaux concepts en C++
  - Conteneurs (ex : vecteur, liste, pile, etc.).
  - Itérateurs ($\approx$ pointeurs de conteneur pour les parcourir).
  - Algorithmes (ex : find, sort, swap, count, etc.).
  - ...

Les Conteneurs (Containers)

Objectif

Fournir des classes (structures de données) pouvant contenir des objets (int, double, char*, class, etc.).

"En gros", trois groupes :

- Conteneurs type "séquence" :
  - tableau (*vector*),
  - liste (*list*),
  - "Double ended queue" (*deque*).

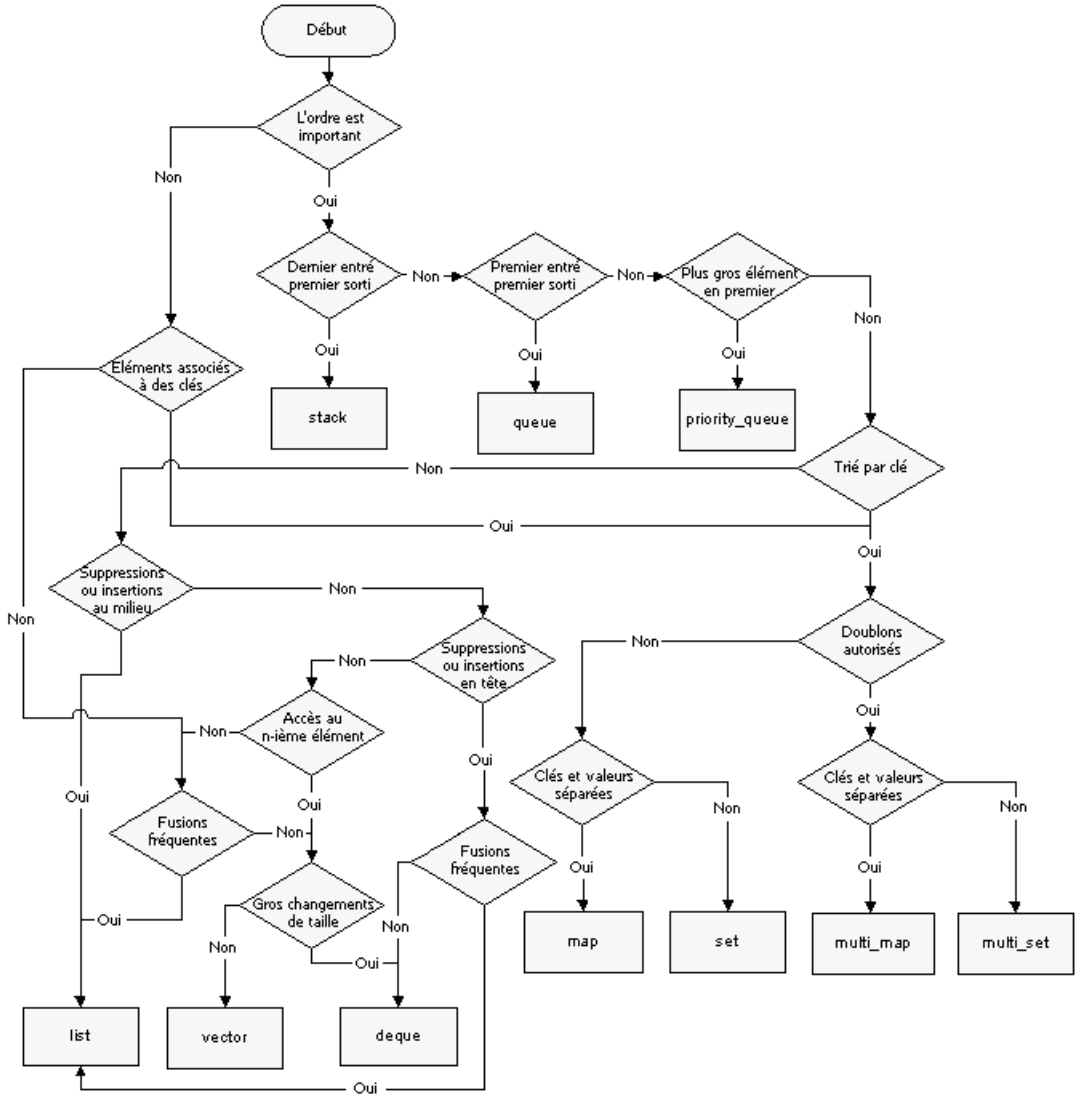
Les Conteneurs (Containers)

- Conteneurs type "dérivés" :
  - pile (*stack*),
  - file d'attente (*queue*),
  - file d'attente avec priorité (*priority_queue*).
- Conteneurs type "associatifs" :
  - ensemble (*set*),
  - ensemble avec doublon possible (*multiset*),
  - ensemble clé/valeur (*map*),
  - ensemble clé/valeur avec doublon possible (*multimap*),
  - ensemble de booléen (*bitset*).

Comment choisir :

- en fonction de vos besoins,
- en fonction des futures opérations (type d'insertion, de recherche, etc.).

Les Conteneurs (Containers)



Conteneurs

Quelques remarques :

- Les conteneurs partagent plusieurs caractéristiques communes et se distinguent par d'autres. Exemple :
  - On peut accéder à un élémet d'un *vector* par [] alors que ce n'est pas le cas pour une *list*.
  - On peut supprimer directement une valeur dans une *list* sans la rechercher alors que ce n'est pas le cas pour un *vector*.

- Gestion de la mémoire transparente,
  - lors d'ajout, de suppression, etc.
  - enfin des "tableaux dynamiques" ! (pas besoin de connaître la taille à l'avance.)

Conteneurs

D'autres remarques :

- Utilisation de fonctions membres pour accéder aux objets.
- Fournissent des itérateurs pertinents pour parcourir les éléments d'un conteneur.
- Interchangeables, généralement, mais attention à la complexité des opérations.
- Modèle objet (Constructeur/destructeur, exceptions, surcharge d'opérateurs, ...)

Les Conteneurs : exemples

Le vector

```
1   using namespace std;
2   ...
3   vector<int> v_i_UnVector;
4
5   v_i_UnVector.push_back(1);
6   v_i_UnVector.push_back(2);
7   v_i_UnVector.push_back(3);
8
9   cout << v_i_UnVector.front() << endl; //affiche 1
10  cout << v_i_UnVector.back() << endl;  //affiche 3
11
12  v_i_UnVector.resize(5); //nouveaux entiers à 0
13  v_i_UnVector[3]=4;
14  v_i_UnVector[4]=v_i_UnVector[3]+1;
```

Les Conteneurs : exemples

Le vector

```
1    for( int i =0; i < v_i_UnVector.size ( ) ; i++)
2     cout << v_i_UnVector[i] << endl;
3    //affiche 1 2 3 4 5
4
5    if( ! v_i_UnVector.empty() )
6        v_i_UnVector.clear();
```

Et encore, c'est sans utiliser les itérateurs...

Les Conteneurs : exemples

Le vector - 2D

```
1   int size_tab_2D=20;
2
3   vector<vector<double>> vec_vec_double(size_tab_2D,
        vector<double>(size_tab_2D,0));
4
5   ...
6   for(int i=0; i<size_tab_2D;i++)
7    for(int j=0; j<size_tab_2D;j++)
8      vec_vec_double[i][j] = vec_vec_double[i][j] +
           vec_vec_double[j][i] ;
```

Les Conteneurs : exemples

La Map

```
1  map<char, float> m_c_f_UneMap;
2
3  m_c_f_UneMap.insert( pair<char,float>('z',4.2) );
4  m_c_f_UneMap.insert( pair<char,float>('g',3.1) );
5  m_c_f_UneMap.insert( pair<char,float>('a',5.1) );
6  cout << "['g'] = " << m_c_f_UneMap['g'] << endl;
7  //affiche ['g'] = 3.1
8
9  cout << m_c_f_UneMap.size() << endl;//affiche 3
10
11 mymap.erase('g');
12
13 //la map contient :
14   'a' => 5.1
15   'z' => 4.2
```

Les Conteneurs : complexité

Conteneur	Insertion			Suppression			Accès
	Début	fin	ième	Début	fin	ième	ième
vector	n/a	O(1)	O(n)	n/a	O(1)	O(n)	O(1)
list	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	n/a
deque	O(1)	O(1)	n/a	O(1)	O(1)	O(n)	O(1)
queue	n/a	O(1)	n/a	O(1)	n/a	n/a	n/a
stack	O(1)	n/a	n/a	O(1)	n/a	n/a	n/a
map	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(log n)
multimap	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(log n)
set	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(log n)
multiset	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(log n)

Les itérateurs

- Généralisation des pointeurs : un itérateur est un objet qui pointe sur un autre objet (un conteneur).
- Utiles pour parcourir/incrémenter des ensembles d'objets : si un itérateur pointe sur un objet (conteneur), il est possible de l'incrémenter pour pointer sur l'objet suivant.
- Masque la complexité de l'organisation des données.
- Aspect central de la programmation générique :
  - Un algorithme a besoin de parcourir un ensemble de données.
  - S'il sait utiliser un itérateur, il est possible d'utiliser le même algorithme sur tous les conteneurs.
- Très grande facilité d'utilisation ! Fonctionne exactement comme on aimerait.

Les itérateurs

- En pratique la STL donne 5 types d'itérateurs :
  - `InputIterator` : lecture des données, déplacement d'un élément à la fois dans une seule direction.
  - `OutputIterator`: écrire des données, déplacement d'un élément à la fois dans une seule direction.
  - `ForwardIterator`: lecture/écrire des données, déplacement d'un élément à la fois dans une seule direction.
  - `BiderectionnalIterator`: lecture/écrire des données, déplacement d'un élément à la fois dans les deux directions (list).
  - `Random Access Iterator`: lecture/écrire des données, déplacement d'un ou plusieurs éléments à la fois dans les deux directions (vector).

Les itérateurs : principe général

Principe général

```
1  //Déclaration
2  TYPE_CONTENEUR::iterator myIterator;
3
4  //Utilisation
5  myIterator=TYPE_CONTENEUR.begin();//Pointe sur le
       premier élément du conteneur
6  ...
7  myIterator=TYPE_CONTENEUR.end();//Pointe sur la "fin"
       du conteneur
8  ...
9  myIntVectorIterator++;//Pointe sur le prochain élément
        du conteneur
10 ...
11 myIntVectorIterator=myIntVectorIterator+5;
12 //Incrémente 5 fois le pointeur (valable que pour les
       Random Access Iterator)
```

Les itérateurs : exemples

Les itérateurs de vector

```
1vector<double> v_d_UnVec;
2vector<double>::iterator it_v_d;
3...
4for(it_v_d = v_d_UnVec.begin(); it_v_d != v_d_UnVec.end
    (); it_v_d++)
5     if (*it_v_d == 3,14159265)
6          cout << "Une valeur trouvée de PI" << endl;
7...
8it_v_d = v_d_UnVec.begin();
9v_d_UnVec.erase(it_v_d,it_v_d+2);
10//Supprime les deux premiers éléments
11...
12vector<int> v_d_tmp(2,42);
13it_v_d = v_d_UnVec.begin();
14v_d_UnVec.insert(it_v_d,v_d_tmp.begin(),v_d_tmp.end());
15   //Insère deux éléments 42 au début du vecteur
```

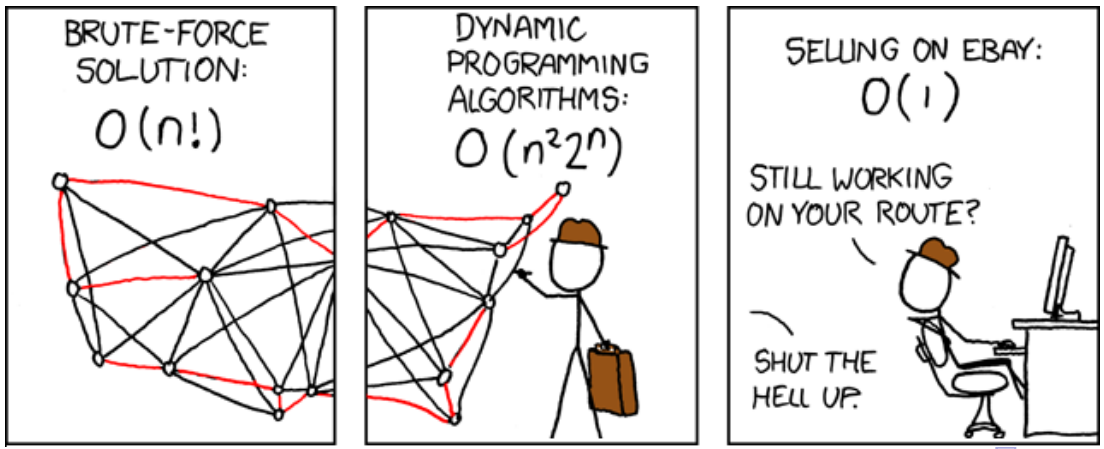
Les itérateurs : exemples

Les itérateurs de vector

```
1  map<char,float> m_c_f_UneMap;
2  map<char,float>::iterator it_m_c_f;
3...
4for (it_m_c_f=m_c_f_UneMap.begin(); it_m_c_f!=
     m_c_f_UneMap.end(); ++it_m_c_f)
5std::cout << it_m_c_f->first << " => " << it_m_c_f->
     second << endl;
6...
7it_m_c_f=mymap.find('b');
8mymap.erase(it_m_c_f);
9...
```

Les algorithmes

- Utilisent les itérateurs
- Dans la STL : algorithmes de base, 2 types :
  - les basiques : recherche d'élément, comptage, copie, remplacement, inversion de tableau, tri, ...
  - les avancés (utilisation de foncteurs) : compter si, appilquer à tous les éléments un traitement, chercher si, ...
- Pas dans la STL : algorithmes complexes
  - Fonctions mathématiques avancées
  - Algorithmes complexes (ex: résolution du PVC)



Les algorithmes : Les basiques

Exemple

```
1 list<int> l_i_UneListe;
2 ...
3 l_i_UneListe.sort();
4 ...
5 l_i_UneListe.reverse();
6 ...
7 l_i_UneListe.unique();
8 ...
9 l_i_UneListe.remove(42);
10 ...
11 l_i_UneListe.merge(l_i_UneAutreListe);
12 //l_i_UneAutreListe est ensuite vide
```

Et encore, c'est sans les foncteurs !

Pour aller plus loin: les foncteurs

- Plusieurs fonctions de la STL permettent d'appliquer des fonctions à des données
- Exemple: generate(it début, it fin, f) applique la fonction f à tous les éléments entre les positions début et fin.

Comment et où définir ces fonctions ?

- On ne peut pas passer des fonctions en paramètre mais des pointeurs de fonctions.
- Définir des fonctions "dans le vide" c'est mal !
- Une fonction est un service, ne prends pas en compte ses appels précédents (sauf variables globales, statiques,... C'est mal aussi (souvent))

Les foncteurs (Functor = Function Object)

- En POO, on aime les objets.
- Un foncteur est un objet que l'on peut utiliser comme une fonction.
- On bénéficie du formalisme des objets, attributs, encapsulation, ...
- Pour la performance, les compilateurs aiment bien (inlining aisé).

En résumé : Un foncteur est une classe qui surcharge l'opérateur d'appel de fonction (*operator()*).

Les algorithmes avancés

Exemple

```
1   int i=1;
2   vector<int> v_i_UnVecteur(5);
3   vector<int>::iterator it_v_i;
4   for(it_v_i=v_i_UnVecteur.begin();it_v_i!=
        v_i_UnVecteur.end();it_v_i++)
5       *it_v_i = i++;
```

- On souhaite affecter la séquence 1..5 à notre tableau.
- `generate` prends en argument une fonction qui ne prends pas d'argument (Generator).
- On a besoin de maintenir un état des appels de la fonction.

=> Foncteur !

Les algorithmes avancés

Un foncteur

```
1class EntierSuiv
2{
3    private:
4        int courrant;
5    public:
6        EntierSuiv(const int init = 0): courrant(init) {}
7        int operator()() {return ++courrant;}
8};
```

- Lorsqu'appel au constructeur, on récupère une référence sur un objet qui possède une opération ()
- Presque comme un pointeur sur une fonction
- En un peu mieux tout de même

Les algorithmes avancés

L'exemple

```
1    int i=1;
2    std::vector<int> v(5);
3    std::vector<int>::iterator it;
4    for(it=v.begin();it!=v.end();it++)
5        *it = i++;
```

Avec foncteur

```
1    std::vector<int> V(5);
2    std::generate(V.begin(),V.end(),EntierSuiv(0));
```

Les algorithmes avancés

Autre exemple avec find_if

```
class Cherche42{
   private:
      int last_int;
   public:
      Cherche42(): last_int(0) {}
      bool operator()(int val){
         if((last_int==4)&&(val==2))  return true;
         last_int=val;
         return false;
   }
};
.....................................
   it=std::find_if(v.begin(),v.end(),Cherche42()) ;
   if(it!=v.end())
      cout << "trouvé !" <<endl;
```

Les algorithmes avancés

Un peu plus loin avec sort

```
1 class A
2 {
3     int i_LaCle;
4     double d_unNombre;
5     A(int i, double d) : i_LaCle(i), d_unNombre(d) {}
6 };
7 ..................................
8   std::vector<A> v;
9   v.push_back(A(2, 4.2));
10  v.push_back(A(1, 5.1));
11  v.push_back(A(3, 3.1));
12  sort(v.begin(), v.end(), MySortByCle());
```

Les algorithmes avancés

Un peu plus loin avec sort

```
1 class MySortByCle
2 {
3     MySortByCle():{}
4     bool operator ()(const A& a1, const A& a2) const
5     {
6         return a1.i_LaCle < a2.i_LaCle;
7     }
8 };
```

for_each

- Appliquer une fonction sur chaque éléments d'un conteneur (entre deux itérateurs)
- Différent de generate:
  - retourne le paramètre d'entrée.
  - ne modifie pas les éléments du conteneur.

for_each

Le foncteur

```
1   template<class T> class Afficher
2   {
3       private:
4           int count;
5           std::ostream &os;
6       public:
7           Afficher(std::ostream &out): os(out),count(0)
                {}
8           void operator()(T x) {os << x << ' '; count
                ++;}
9   };
```

Utilisation

```
1std::for_each(V.begin(),V.end(),Afficher<int>(std::cout
```

for_each

- Un peu plus puissant encore que ça...
- `for_each` renvoie le foncteur après qu'il a été appliqué à tous les éléments

Utilisation

```
1  Afficher<int> afficheur = std::for_each(V.begin(),V.
       end(),Afficher<int>(std::cout));
2  std::cout << std::endl << afficheur.getCount() << "
       valeurs affichées" << std::endl;
```

Afficher un tableau à l'écran

Itérateurs

```
1  for(it=v.begin();it!=v.end();it++) {
2      std::cout << *it << " ";
3  }
```

for_each + foncteur

```
1  std::for_each(v.begin(),v.end(),Afficher<int>(std::
       cout))
```

- Première solution: on gère une boucle, des pointeurs, il va falloir copier/coller pour une autre structure
- Deuxième solution: foncteur et template : on peut faire plein de choses avancées, mais ... ça existe peut-être déjà

Utiliser le bon outil

- On veut recopier les valeurs d'un tableau sur la sortie standard
- La bonne notion ici est celle d'itérateurs
- `ostream_iterator` exactement ce que l'on cherche

ostream_iterator

```
1  std::copy(v.begin(), v.end(), l.begin());
2  std::copy(v.begin(), v.end(), std::ostream_iterator<
       int>(std::cout, " "));
```

transform

- for_each ne permet pas de modifier les éléments du conteneur
- transform permet de le faire
- 4 arguments:
  - intervalle d'itérateurs
  - itérateurs de sortie
  - fonction unaire (binaire)

Passer au carré

```
1    class Carre
2    {
3        public:
4            Carre(){}
5            int operator()(int x) {return x*x;}
6    };
7    ...
8transform(vi.begin(),vi.end(),vo.begin(),Carre());
```

Quelques mots sur bind1st() et bind2nd()

- Transformer un prédicat qui prend 2 arguments en un prédicat qui n'en prend qu'un

bind1st() et bind2nd()

```
1  ...
2  bool IsSup(int a, int b) {return a>b;}
3  ...
4  //it = find_if( v.begin(), v.end(), IsSup(2) );//ko
5  ...
6  it = find_if( v.begin(), v.end(), bind2nd( ptr_fun(
       IsSup ) , 2 ) );//ok, s'arrête dès que v[i]>2
7  // ptr_fun : fonctions de conversions de pointeur de
       fonction vers foncteur
8  ...
9  it = find_if( v.begin(), v.end(), bind1st( ptr_fun(
       IsSup ), 2 ) );//ok, s'arrête dès que 2>v[i]
```

Les algorithmes

- $<$ *algorithm* $>$ contient de nombreux autres algorithmes qui utilisent les concepts précédents
- En règle général, ne pas chercher à faire mieux que la STL.
- Quelques algorithmes utiles :
  - `for_each`, `transform`
  - `copy`
  - `sort`
  - `min_element`, `max_element`
  - `find`, `find_if`
  - `count`, `count_if`
  - `replace_if`, `remove_if`
  - ...

L'objet string

- La librairie standard donne aussi un objet string.
- std::string est un conteneur qui encapsule un buffer de caractères.
- Une API particulièrement riche.
  - `operator[]`
  - `c_str()` renvoie la version C de la chaîne (const char *)
  - itérateurs begin() et end()
  - `operator+`, `operator+=`, `operator=`
  - `insert`, `erase`, ...

L'objet string

Un exemple classique

```
1   std::string chaine = "ls -il | wc -l";
2   std::string membre1(10,' ');
3   std::string membre2;
4
5   std::copy(chaine.begin(),std::find(chaine.begin(),
        chaine.end(),'|'),membre1.begin());
6
7   std::copy(std::find(chaine.begin(),chaine.end(),'|')
        +1,chaine.end(),membre2.begin());
8
9   std::cout << "Chaine: " << membre1 << "|" << membre2
        << std::endl;
```

Conclusion sur la STL

- Trois types fondamentaux:
  - Conteneurs: Organisent les données
  - Itérateurs: généralisent les pointeurs
  - Algorithmes: travaillent sur les conteneurs par le biais des itérateurs
- La STL aide à faire du code réutilisable (ne pas réinventer la roue)
- En général, pas la peine d'essayer de faire mieux que la STL (!)
- Ne pas gérer ses chaînes de caractères comme en C. . .pas de `sprintf` et co
- Gérer les exceptions, éviter de devoir gérer l'allocation mémoire
- Utiliser toute la puissance de algorithm
- Et la puissance des flux C++

Quelques mots sur les flux

Entrées/Sorties gérées par 2 classes ($< iostream >$) :

- ostream : permet d'écrire des données vers la console, un fichier, ... surdéfinition de l'opérateur $<<$,
- istream : permet de lire des données à partir de la console, d'un fichier, ... surdéfinition de l'opérateur $>>$.

Les flux standards :

- cout écrit vers la sortie standard,
- cerr écrit vers la sortie d'erreur,
- cin lit à partir de l'entrée standard.

Les flux

Exemple

```
1 ...
2 const string nom_fichier("plop.txt");
3 ofstream fichier(nom_fichier.c_str());
4 ...
5 if(fichier.fail()) {
6   cerr << "Erreur: " << nom_fichier << endl;
7 }
8 ...
9 fichier << "blablabla" << un_string << endl;
10 ...
11 fichier.close();
```

Les flux

Exemple

```
1...
2const string nom_fichier("plop.txt");
3ifstream fichier(nom_fichier.c_str());
4string nom;
5int id;
6...
7if(fichier.fail()) {
8   cerr << "Erreur: " << nom_fichier << endl;
9}
10do{
11  fichier >> nom >> id;
12}while(!fichier.eof());
13fichier.close();
```

Sommaire

Boost

Boost est un ensemble de bibliothèques C++ gratuites et portables :

- les threads,
- les matrices et les tableaux à dimensions multiples,
- les expressions régulières,
- la méta-programmation,
- la date et l'heure,
- les fichiers et les répertoires,
- gérer la mémoire avec des pointeurs intelligents,
- faire de la sérialisation en binaire / texte / XML,
- manipuler des graphes mathématiques (pour la RO),
- ...

Boost : pointeurs intelligents

- Plusieurs types, les plus courants sont : boost::shared_ptr (pointeur simple) et boost::shared_array (pour les tableaux).
- Objectif : pallier aux problèmes suivants :
  - Oublier de désallouer de la mémoire
  - Libérer plusieurs fois de la mémoire allouée
  - Accéder à la valeur pointée par un pointeur invalide (ex: mémoire désallouer).
- Compteur de référence associé à un objet (si $= 0$ alors désallocation)

Boost : exemple de pointeurs intelligents

Exemple

```
1 ...
2 #include <boost/shared_ptr.hpp>
3
4 class Test
5 {
6   public:
7     Test( const char * sonNom ) nom( sonNom ){ }
8     ~Test() { std::cout<<"Destruction de" << this->nom
          <<'\n';}
9     void printName() {std::cout << this->nom << '\n';}
10
11  private:
12    std::string nom;
13 };
```

Boost : exemple de pointeurs intelligents

```
1typedef boost::shared_ptr<Test> TypePtr; //Déf. type
2...
3     TestPtr ptr; // Initialisé à NULL
4     {
5          TypePtr ptr_tmp( new Test( "objet1" ) );
6          ptr = ptr_tmp;
7     } // ptr_tmp est détruit mais ptr reste valide
8     ptr->printName(); //Affiche "objet1"
9     ptr.reset( new Test( "objet2" ) ); // objet1 est
          détruit, objet2 est créé
10    ptr->printName(); //Affiche "objet2"
11    TypePtr ptr2 = ptr;
12    ptr.reset(); // Mise à NULL de ptr
13    ptr2.reset(); // Mise à NULL de  ptr2, objet2 est
          détruit
14    ptr->printName(); // utilisation du pointeur NULL :
           erreur en mode Debug
15}
```

Boost

Plus d'infos :

- http://www.boost.org/

Cependant, il manque encore quelque chose...

Boost

Plus d'infos :

- http://www.boost.org/

Cependant, il manque encore quelque chose...

...Comment faire des IHMs ?

- MFC
- wxWidgets
- **Qt**
- gtfmm
- ...