Programmation avancée en C++: STL 1

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

STL: principe

- La notion de classe couplée avec l'héritage et les templates permettent de construire et gérer des structures de données génériques.
- Un certain nombre de structures de données/algorithmes abstraites et efficaces existent: pile, liste, "map" etc.
- Ces algorithmes/structures de données standard sont déjà implémentés dans la STL.
- Elle contient:
  - **Des conteneurs**: structures de données telles que: vecteurs, listes, tableaux associatifs etc,
  - **Des itérateurs**: outil généralisant les pointeurs permettant de manipuler un conteneur. Il "pointe" sur un élément courant du conteneur, "++" permet de passer à l'élément suivant.
  - **Des algorithmes**: sur ces conteneurs. Exemple : tri, copie, maximum etc
  - **Quelques outils numériques**: complexe, vecteurs, etc.

1. Outils numériques

Outils numériques I

- Les complexe et une classe pour manipuler les vecteurs numériques sont implémentées dans la STL.
- Il s'agit de template de classe pour les rendre génériques.
- Ces classes fournis utilise les templates et l'héritage et polymorphisme.

```
1 #include <complex>
2 #include <valarray>
```

- La classe "complexe" est proche de la nôtre. Mais les fonctions exp, log, sinus etc sont implémentées pour les complexes.
- La classe "valarray" permet du calcul numérique sur des vecteurs, on a accès aux opérateurs de calcul mais aussi la capacité d'extraire des sections etc.

Outils numériques II

- Exemple: les complexes

```
template <class T> void exemple_1(complex<T> x){
  complex<T> y,z;
  double r;
  y = conj(x);
  r = abs(y);
  z = y *x;
  z = z/r;
  cout<< " z egal: "<<z<<endl;
}
template <class T> void exemple_2(complex<T> x){
  complex<T> i(0,1);
  complex<T> y,z,k;

  z = 1.0+i; k=polar<T>(2.0,pi/4.0);
  y=exp(z);  x=log(x)*k;
  cout<< " x egal: "<<x<<endl;
}
```

- On voit que les opérateurs de type produit externe, sont aussi définis.
- Le template de fonction ”polar” permet de créer un complexe par la forme exponentielle (notre ”complexe” de module 1).

Outils numériques II

- Exemple: constructeurs de ”valarray”

```
valarray <int> v1(10); // un vecteur de taille 10
valarray <double> v2(0.5,10); // un vecteur de taille 10 init à 0.5
complex<int> t[] = {-4,1,2};
valarray < complex<int> > v3(t,3);
// un vecteur de taille 3 init avec le tableau t
```

- On peut construire un vecteur avec l’aide d’un certain nombre de constructeurs comme les ”tableaux”) avec le type de notre choix.
- Les opérateurs: ”+”, ”-”, ”[]”, ”==”, ”¡” etc sont surchargés. On peut appliquer des fonctions à l’ensemble du vecteur.

```
double pol(double x){
  double res;
  res = x*x - x + 4.0;
  return 0;  }
double t2[] = {-4.0,1.0,2.5,1.4};
valarray <double> v4(t2,4),v5,v6;
v5 = exp(v4);
v6 = v4.apply(pol);
```

Outils numériques III

- Les valarray ressemble beaucoup au ”array” de numpi et sont très utiles pour le calcul.
- Exemple: opérations sur les ”valarray”

```
1 valarray <double> v2(0.5,10);
2 valarray <double> v4(t2,4);
3 v2=v4;
4 v2.resize(1);
```

- v2 est transformé en un vecteur de taille 4. Les autres valeurs sont effacées.
- On ne peut pas faire si le type des vecteurs est différent.
- ”resize()” permet de modifier la taille du vecteur. La fin est effacée dans ce cas.
- On peut prendre des ”sections” de vecteurs:

```
1 valarray <double> v2(0.5,10);
2 valarray <double> v4(t2,4);
3 v4 = v2[slice(1,3,2)]
```

- La coupe part de 1, prend trois nombres, un nombre sur deux.

2. Conteneurs sequentiels

Conteneurs :généralité

- Les conteneurs sont des structures de données (en pratique template de classe) fournies par la STL.
- **Conteneurs**: **vector** (tableau dynamique) , **list** (liste doublement chaînée), **deque** (queue a double entrée).
- Pour chaque conteneur, on a:
  - la construction (par constructeur),
  - l'affectation,
  - la comparaison (surcharge d'opérateur),
  - l'insertion et la suppression d'éléments (par les "iterator").
- **1er conteneur**: vector.
  - taille variable pendant l'exécution,
  - accès à un élément en $O(1)$,
  - applicable à toutes les classes.

```
1 #include <complex>
2 #include <valarray>
```

Vector I: Construction

- On détaille la forme des constructeurs pour le conteneur ”vecteur”.

```
1  #include "complexe.hpp"
2  #include "polynome.hpp"
3  #include "rationnel.hpp"
4  #include <vector>
5  using namespace std;
6  int main (){
7    vector<double> c1;
8    vector<rationnel> c2;
9    vector< polynome<double,double> > c3(5);
10   complexe<double> a(0.5,0.1);
11   vector < complexe<double> > c4(5,a);
12   vector <double> c5(c1);
13   int  t[] = {2,-1,3};
14   vector <int > c6(t,t+3);
15   return 0; }
```

- Un constructeur vide, un constructeur prenant un nombre élément donné, le même avec en plus une valeur par défaut, un constructeur par copie et un avec un tableau de valeur.
- **Utilisation**: utilisable avec n’importe quelle classe.

Vector II: Affectation et comparaison

- Il existe plusieurs voies pour affecter des ”vector”.
- Le ”=” est surchargé. Valide quelque soit la taille des ”vector”. Impossible entre deux ”vector” de type différents (”vector< *int* >”, ”vector< *double* >”).
- Autre possibilité: méthode ”assign”.

```
1 vector< polynome<double,double> > c3(5);
2 vector < complexe<double> > c4(5,a);
3 int t[] = {2,-1,3};
4 c6.assign(t,t+3);
5 c4.assign(5,a);
6 c4.clear();
7 c1.swap(c5);
```

- Même signature pour ”assign” et pour les constructeurs. ”clear” pour vider un conteneur et ”swap” pour les échanger.
- **Opérateurs de comparaison**: les opérateurs de comparaison ”==”, ”<”,”>” sont surchargés.
  - Pour des tableaux de même taille, ”==” est vrai si il est vrai pour tout élément.
  - Pour < il test < pour chaque élément et s’arrête à la fin d’un des deux tableaux (vrai) ou lorsqu’on une comparaison est fausse (faux).

Vector IV : itérateur

- On peut accéder à un élément d'un conteneur par les crochets: ”[]”.

```
1 cout<<" Ecrire c1: " <<c6[0]<<endl;
```

```
user $ Ecrire c1: 2
```

- Cependant afin d'assurer les actions sur un conteneur il est proposé la notion d'itérateur (iterator)
- Il s'agit d'une généralisation de pointeur pour manipuler un conteneur:
  - il pointe vers un élément du conteneur, peut être incrémenté, décrémenté, déréférencé avec * (si ”it” est un itérateur sur un element ”*it” désigne l'élément en question).
  - Les itérateurs peuvent être comparés.

```
1 vector <int > c6(t,t+3);
2 vector <int> :: iterator i;
3 for (i =c6.begin(); i < c6.end(); i++) {
4   cout << "c6: "<<*i<<endl;}
```

Vector V : itérateur

- Les méthodes ”begin” et ”end” permettent d’obtenir la position de début et de fin dans le conteneur.
- Exemple:

```
i=c6.begin();
*i =6; // place 6 dans le premier élément
i++; i*=7; // l’itérateur est incrémenté et pointe sur le second élément.
// On place donc 7 dans le second element
```

- l’itérateur s’utilise vraiment comme un pointeur. On peut utiliser ”[]” pour faire la même chose.
- Les itérateurs sont par contre essentiels pour des manipulations plus lourdes des conteneurs.

Vector VI : insertion/suppression

- Contrairement aux tableaux classiques, on peut modifier dynamiquement les conteneurs avec l'insertion/suppression d'élément.
- Insertion/suppression en $o(n)$ opérations sauf si c'est en fin de vector. Dans ce cas $O(1)$.
- Pour utiliser les fonctions d'insertion/suppression il faut les itérateurs.

```
1  vector < complexe<double> > :: iterator ic;
2    for (ic = c4.begin(); ic < c4.end(); ic++){
3      cout <<"c4: "<< *ic<<endl; }
4
5    ic = c4.begin()+2;   cout<<"////// "<<endl;
6
7    c4.insert(ic,2.0*a);
8    for (ic = c4.begin(); ic < c4.end(); ic++){
9      cout <<"c4: "<< *ic<<endl;}
10   return 0;
```

- La méthode "insert" prend comme paramètre une position qui est le type de retour de l'itérateur. On ne peut pas utiliser un entier.
- C'est l'insertion en milieu de vecteur qui justifie l'utilisation de ces conteneurs et des itérateurs.

Vector VI : insertion/suppression

- Contrairement aux tableaux classiques, on peut modifier dynamiquement les conteneurs avec l'insertion/suppression d'élément.
- Insertion/suppression en $o(n)$ opérations sauf si c'est en fin de vector. Dans ce cas $O(1)$.
- Pour utiliser les fonctions d'insertion/suppression il faut les itérateurs.

```
user $ c4: 0.5+i 0.1
user $ c4: 0.5+i 0.1
user $ c4: 0.5+i 0.1
user $ //////
user $ c4: 0.5+i 0.1
user $ c4: 0.5+i 0.1
user $ c4: 1+i 0.2
user $ c4: 0.5+i 0.1
```

- La méthode "insert" prend comme paramètre une position qui est le type de retour de l'itérateur. On ne peut pas utiliser un entier.
- C'est l'insertion en milieu de vecteur qui justifie l'utilisation de ces conteneurs et des itérateurs.

Vector VII : insertion/suppression

- Type d'insertions:

```
 cout<<"////// "<<endl;
  ic = c4.begin()+2;
  c4.insert(ic,3,3.0*a);
  for (ic = c4.begin(); ic < c4.end(); ic++){
    cout <<"c4: "<< *ic<<endl;}

cout<<"////// "<<endl;
  c4.erase(c4.begin());
  c4.erase(c4.begin()+1,c4.begin()+3);
  for (ic = c4.begin(); ic < c4.end(); ic++){
    cout <<"c4: "<< *ic<<endl;}
  return 0;
```

- ”insert(ic,n,m)” insère n fois ”m” avant ic. ”insert(ic1,ic2,ic3)” insère les valeurs entre ic1 et ic2 d'un conteneur avant ic3 dans le conteneur courant.
- Pour supprimer on utilise ”erase”. Le ”end” et ”begin” sont mis à jour.

Vector VII : insertion/suppression

- Type d'insertions:

```
user $ c4: 0.5+i 0.1
user $ c4: 0.5+i 0.1
user $ c4: 1.5+i 0.3
user $ c4: 1.5+i 0.3
user $ c4: 1.5+i 0.3
user $ c4: 1+i 0.2
user $ c4: 0.5+i 0.1
user $ //////
user $ c4: 0.5+i 0.1
user $ c4: 1.5+i 0.3
user $ c4: 1+i 0.2
user $ c4: 0.5+i 0.1
```

- ”insert(ic,n,m)” insère n fois ”m” avant ic. ”insert(ic1,ic2,ic3)” insère les valeurs entre ic1 et ic2 d'un conteneur avant ic3 dans le conteneur courant.
- Pour supprimer on utilise ”erase”. Le ”end” et ”begin” sont mis à jour.

Vector VIII : insertion/suppression

- Accès en fin de ”vector”. opération en $O(1)$.
- Il existe des accès spécifiques pour la fin d’un ”vector”

```
c4.back() = 4.0*a;
for (ic = c4.begin(); ic < c4.end(); ic++){
    cout <<"c4: "<< *ic<<endl;}

c4.pop_back();
for (ic = c4.begin(); ic < c4.end(); ic++){
    cout <<"c4: "<< *ic<<endl;}

c4.push_back(5*a);
for (ic = c4.begin(); ic < c4.end(); ic++){
    cout <<"c4: "<< *ic<<endl;}
```

- ”back” pour accéder au dernier élément, ”push_back” pour ajouter un élément en fin de vecteur, ”pop_back()” pour supprimer le dernier élément.

Vector VIII : insertion/suppression

- Accès en fin de ”vector”. opération en $O(1)$.
- Il existe des accès spécifiques pour la fin d’un ”vector”

```
user $ c4: 0.5+i 0.1
user $ c4: 1.5+i 0.3
user $ c4: 1+i 0.2
user $ c4: 2+i 0.4
user $ //////
user $ c4: 0.5+i 0.1
user $ c4: 1.5+i 0.3
user $ c4: 1+i 0.2
user $ //////
user $ c4: 0.5+i 0.1
user $ c4: 1.5+i 0.3
user $ c4: 1+i 0.2
user $ c4: 2.5+i 0.5
```

- ”back” pour accéder au dernier élément, ”push_back” pour ajouter un élément en fin de vecteur, ”pop_back()” pour supprimer le dernier élément.

Vector IX: gestion mémoire

- Afin de bien comprendre comment utiliser efficacement un vecteur, il faut comprendre comme la mémoire est gérée.
- Allocation mémoire:
  - La mémoire allouée est parfois supérieure à la mémoire utilisée en pratique. Cela permet de ne pas ré-allouer à chaque élément inséré.
  - La ré-allocation à lieu quand la mémoire additionnelle est épuisée.
  - Compromis mémoire/cout. Meilleur pour la mémoire: réallouer à chaque fois, meilleur pour le calcul: ne jamais réallouer.

```
1  vector<rationnel> d1(4);
2  vector<rationnel> d2(7);
3  cout<<"size d1 :"<<d1.size()<<" capacity: "<<d1.capacity()<<endl;
4  cout<<"size d2 :"<<d2.size()<<" capacity: "<<d2.capacity()<<endl;
5  d1.push_back(rationnel(1,1));
6  d2.push_back(rationnel(1,1));
7  cout<<"size d1 :"<<d1.size()<<" capacity: "<<d1.capacity()<<endl;
8  cout<<"size d2 :"<<d2.size()<<" capacity: "<<d2.capacity()<<endl;
9  d1.push_back(rationnel(1,1));
10 d2.push_back(rationnel(1,1));
11 cout<<"size d1 :"<<d1.size()<<" capacity: "<<d1.capacity()<<endl;
12 cout<<"size d2 :"<<d2.size()<<" capacity: "<<d2.capacity()<<endl;
```

Vector IX: gestion mémoire

- Afin de bien comprendre comment utiliser efficacement un vecteur, il faut comprendre comme la mémoire est gérée.
- Allocation mémoire:
    - La mémoire allouée est parfois supérieure à la mémoire utilisée en pratique. Cela permet de ne pas ré-allouer à chaque élément inséré.
    - La ré-allocation à lieu quand la mémoire additionnelle est épuisée.
    - Compromis mémoire/cout. Meilleur pour la mémoire: réallouer à chaque fois, meilleur pour le calcul: ne jamais réallouer.

```
user $ size d1 :4 capacity: 4
user $ size d2 :7 capacity: 7
user $ size d1 :5 capacity: 8
user $ size d2 :8 capacity: 14
user $ size d1 :6 capacity: 8
user $ size d2 :9 capacity: 14
```

- Lors de la première allocation, elle est faite au plus juste. Puis lors de la première insertion plus de mémoire est allouée.

Deque

Deque

- Il s’agit d’une structure similaire à la structure ”vector”:
  - l’accès est en $O(1)$,
  - l’insertion/suppression en $O(n)$,
  - l’insertion/suppression en fin en $O(1)$,
  - l’insertion/suppression début en $O(1)$.
- Les constructeurs, méthodes etc sont les mêmes.

```
1 rationnel f(2,3);
2 deque<rationnel> e1(4,f);
3 e1.push_front(f);
4 deque<rationnel> :: iterator ie;
5 for (ie = e1.begin(); ie < e1.end(); ie++){
6    cout <<"e1: "<< *ie<<endl;}
```

- On a en plus ”push_front” et ”pop_front” pour insérer/supprimer au début.
- **Remarque**: sur une opération en $O(1)$ ”deque” est plus lent que ”vector”. ”Capacity” n’existe pas pour ”deque”.
- Les éléments ne sont pas tous contiguë. Séquence de tableaux de taille fixe

List I

List

- Il s'agit d'une structure type "liste doublement chainée".
  - itérateur bi-directionnel,
  - l'insertion/suppression en $O(1)$,
  - Pas d'itérateur à accès direct.
- Les constructeurs, méthodes etc sont les mêmes.

- **Itérateur**: situation différente.
  - on peut définir un "iterator" et un "reverse_iterator"
  - ils peuvent être incrémentés ou décrémentés (++ ou --),
  - pas d'incrémentation d'un pas quelconque, pas d'accès avec [].
- Pour accéder à un élément, il faut avoir parcouru la liste jusqu'à lui une première fois.

```
1 double  tl[] = {2.4,-1.2,3,1.2,2.4};
2 list<double> l1(t,t+5);
3 list<double> :: iterator il1;
4 list<double> :: reverse_iterator il2;
```

LIst II

- Les méthodes d'insertion/suppression ainsi que beaucoup d'autres sont les mêmes. Cependant les listes possèdent des méthodes supplémentaires.
- Exemple:

```
1 for (il1 = l1.begin();il1!= l1.end();il1++){cout <<"l1: "<< *il1<<endl;}
2 l1.remove(2.4);
3 cout<<"///////// "<<endl;
4 for (il1 = l1.begin();il1!= l1.end();il1++){cout <<"l1: "<< *il1<<endl;}
```

```
user $ l1: 2.4
user $ l1: -1.2
user $ l1: 3
user $ l1: 1.2
user $ l1: 2.4
user $ /////////
user $ l1: -1.2
user $ l1: 3
user $ l1: 1.2
```

- "remove" permet de supprimer toutes les valeurs "2.4" dans le cas présent.

LIst III

- Il y a d'autres outils comme:
  - "sort()" qui permet de trier la liste,
  - "unique()" qui supprime tous les doublons,
  - "merge(list)" qui fusionne deux listes,
  - "slice(position,list)" qui met après "position" dans la liste courante éléments de "list"

```
1 list<double> l3(tl,tl+1);
2 list<double> l4(tl,tl+2);
3 for (il1 = l3.begin();il1!= l3.end();il1++){cout <<"l3: "<< *il1<<endl;}
4 il1 = l3.begin();
5 l3.splice(il1,l4);
6 cout<<"///////// "<<endl;
7 for (il1 = l3.begin();il1!= l3.end();il1++){cout <<"l3: "<< *il1<<endl;}
```

```
user $ l3: 2.4
user $ /////////
user $ l3: 2.4
user $ l3: -1.2
user $ l3: 2.4
```

Exemple: système dynamique I

- Exemple d'utilisation de la classe "vector": système dynamique.
- Equation de Lokta Voltera:

$$\begin{cases} \dfrac{dx(t)}{dt} = x(t)(\alpha - \beta y(t)) \\[2ex] \dfrac{dy(t)}{dt} = y(t)(\gamma x(t) - \delta) \end{cases}$$

- $x(t)$ les proies et $y(t)$ les prédateurs. α taux de naissance des proies, δ taux de mortalité des prédateurs,
- β taux de mortalité des proies du au prédateur, γ taux de naissance des prédateurs en fonction des proies mangées.
- **Etat stationnaire**: calcul d'un état stationnaire $==>$ on ne connaît pas l'avance la taille de la boucle en temps et du tableau de stockage.
- **Structure de données**: vector. On peut ajouter à la fin en $O(1)$ et on optimise raisonnablement la mémoire.

Exemple: système dynamique II

- Classe ”lv” pour Lokta Voltera

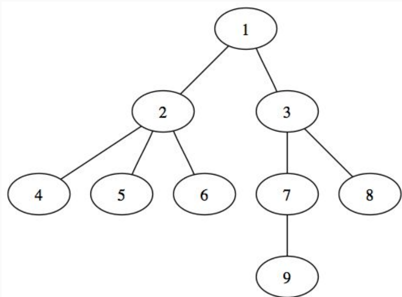
```
class lv {
  double a; double b;
  double c; double d;
  vector<double> x;
  vector<double >y;
  double eps=0;
public:
  tv(){
    a=0.0; b=0.0;
    c=0.0; d=0.0;
    eps=1.0; }
  tv(double al, double be, double ga, double de, double e){
    a=al;  b=be;
    c=ga;  d=de;
    eps=e;  }
  ~tv (){
    x.clear();  y.clear(); }
  void init_LV(double,double);
  void loop_LV(double);
  bool etat_stationnaire(double,double);
};
```

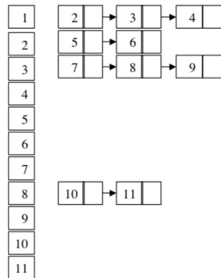
Exemple: système dynamique III

```
1  void lv::init_LV(double x0,double y0){
2    x.push_back(x0);
3    y.push_back(y0);
4  }
5  void lv::loop_LV(double dt,int nmax){
6    double xn=0.0,yn=0.0;
7    bool stop=false;
8    int n=0;
9    while (stop==false && n<nmax){
10     xn = x.back() + dt*(a-b*y.back())*x.back();
11     yn = y.back() + dt*(c*x.back()-d)*y.back();
12     n++;
13     stop=etat_stationnaire((xn-x.back())/x.back(),(yn-y.back())/y.back());
14     x.push_back(xn); y.push_back(yn);  }
15   cout<<" convergence après:"<<n<<" iteration"<<endl;
16 }
17 bool lv::etat_stationnaire(double dx,double dy){
18   bool res= false;
19   double residue= abs(dx)+abs(dy);
20   if(residue< eps) { res = true;}
21   return res;
22 }
```

Exemple: Arbre I

- Les arbres (binaire, n-aire), les forêts (ensemble d'arbres) sont des structures de données importantes en informatique (recherche, indexation) et mathématiques (algorithme décisionnaire, apprentissage et IA, etc).





- Gauche: exemple d'arbre, droite: structure de données pour un arbre.

Exemple: Arbre II

- Une classe noeud contenant: la valeur du noeud, le numéro et la liste des fils.
- Une classe arbre contenant: un vecteur de noeud, le nombre total de noeuds.

```
1  class noeud{
2    double val;
3
4    int num;
5    friend class arbre;
6  public:
7    noeud(){
8      val = 0.0; num = 0; };
9    noeud(const noeud & n){
10     val = n.val;   fils = n.fils;
11     num = n.num;  };
12   ~noeud (){
13     fils.clear(); };
14   ....
15    void ajout_fils(int nbnoeud){
16     fils.push_back(nbnoeud);
17   }
18   void suprime_fils(int nbnoeud){
19     fils.remove(nbnoeud);
20   }
```

Exemple: Arbre III

- La ”list” simplifie la suppression notamment.
- Classe arbre:

```
class arbre{
  vector<noeud> noeuds;
  int nbnodes;
public:
  arbre(){
    nbnodes = 0;
  };
  arbre(const arbre & a){
    noeuds = a.noeuds;
    nbnodes =a.nbnodes;
  };
  ~arbre (){
    noeuds.clear();
  };
 .....
}
```

- Les opérateurs d’affectation sont redéfinis pour les ”vector” et ”list”.
- Le ”clear” permet de nettoyer totalement la structure.

Exemple: Arbre IV

- Deux opérations de base: ajout et suppression de noeud

```
 void ajout_noeud(double x, int pere){
    noeud n; // on crée le noeud
    n.ajout_val(x);
    n.ajout_num(nbnodes+1);
    noeuds.push_back(n); // on ajoute le noeud
    if(nbnodes != 0){ // on precise que ce noeud est le fils du noeud "pere"
      noeuds[pere-1].ajout_fils(nbnodes+1);
    }
    nbnodes++;
}

void supprime_noeud_et_fils(int n){
    vector<noeud>::iterator i,i0;

    for(i = noeuds.begin(); i != noeuds.end(); i++){
      if(i->num == n){ i0=i; break;}  }
    noeuds.erase(i0);
  }
```

- Ici on supprime un noeud et tous les fils. Parfois on voudrait supprimer juste un noeud pas les fils.

Exemple: Arbre V

- Suppression d'un noeud sans supprimer les fils.
- Les fils sont associés au père du noeud supprimé.

```
void supprime_noeud(int n){
   vector<noeud>::iterator i,i0, ip;
   list<int> :: iterator il;
   int np;
   noeud copy;
   // on cherche le numéro du père et la position de l'objet à supprimer
   for(i = noeuds.begin(); i != noeuds.end(); i++){
     if(i->num == n){ i0=i;}
     for(il=i->fils.begin(); il!= i->fils.end(); il++){
       if(*il == n){ np = i->num; }
     }
   }
   for(il=noeuds[n-1].fils.begin(); il!= noeuds[n-1].fils.end(); il++){
    // on copie les fils du noeud supprimé dans le père
     noeuds[np-1].fils.push_back(*il); }
   noeuds[np-1].supprime_fils(n);
   // on supprime le lien au noeud supprimé chez le père
   noeuds.erase(i0); // on supprime le noeud
   nbnodes--;
 }
```

Exemple: Arbre VI

- Exemple:

```
arbre A;
  A.ajout_noeud(2.5,0); A.ajout_noeud(3.0,1);
  A.ajout_noeud(2.0,1); A.ajout_noeud(-2.0,2);
  A.ajout_noeud(1.2,2); A.ajout_noeud(2.4,2);
  A.ajout_noeud(1.7,3); A.ajout_noeud(1.9,3);
  A.ajout_noeud(2.0,7);

  cout<<A<<endl;
  cout<<"////////////////"<<endl;
  A.supprime_noeud(7);
  cout<<A<<endl;

  cout<<"////////////////"<<endl;
  A.supprime_noeud(9);
  cout<<A<<endl;
```

- **En cours**: compiler le code et l'utiliser en cours.
- **Possible opération**: re-numérotation des noeuds.

Auto et iterateur

- Pour simplifier les codes sur les iterateurs: on peut utiliser auto.
- Exemple de base:

```
1 list<int> :: iterator il;
2 list<int> :: ma_list;
3 for(il = ma_list.begin(); il != ma_list.end(); il++){
4    ...
5 }
```

- Autre solution:

```
1 list<int> :: ma_list;
2 for(auto il = ma_list.begin(); il != ma_list.end(); il++){
3    ...
4 }
```

Conclusion

- On a plusieurs structures de données séquentielles: ”list”, ”deque” et ”vector”
- Il en existe d’autre: ”forward_list” (liste simplement chaînée), ”array” (tableau simple etc)
- On peut aussi adapter ces structures de données avec une interface différente (permettant de manier cela comme des piles LIFO ou files FIFO)

Important

Utiliser ces structures de données simplifie la gestion de certains problèmes, mais il faut bien choisir laquelle. Un ”conteneur” de type ”list” est intéressant en terme de coût calcul, mais plus difficile à manier. Le conteneur ”vector” peut assez régulièrement remplacer les tableaux et les tableaux dynamiques.