Containers et itérateurs

Définition

Un **container** (conteneur en français) est une **classe qui contient une collection de données** (objets ou types de base).

Notions de structures de données :

Les structures de données classiques du C++ (et du C pour la plupart) sont :

- Les vecteurs — Structures statiques
- Les listes
- Les piles
- Les files
- Les arbres binaires

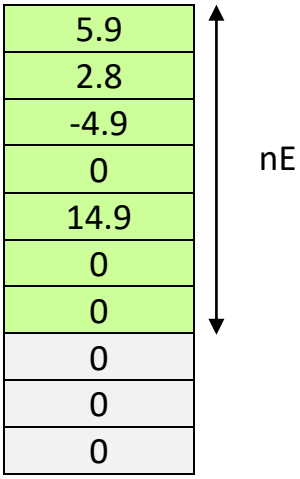
(Les listes, piles, files et arbres binaires : Structures dynamiques)

- Etc…

En C++, la gestion de chacune de ces structures, ou plutôt de chacun de ces types de structures, se fait au sein d'une classe dédicacée à cette tâche.

Un premier exemple : un classe « Vecteur de réels »

```
class CVecteur
{
  private :
    float v[10];
    int nE;
    char checkIndex (const int i);

  public :
    CVecteur (void);
    CVecteur (const int valInit);
    CVecteur (unsigned int nEl) { nE = nEl; }

    int getDim (void) { return nE; }
    void setDim (const int nEl);

    float getVal (int i);
    void setVal (int i, float val);

    void afficheVec (void);
    void entreVec (void);
};
```



Une telle classe est consistante et cohérente, car elle fournit à l'utilisateur toute l'interface qui permet d'incorporer des objets de cette classe dans un programme.

Exemple d'utilisation :

```
int main()
{
  CVecteur v((unsigned) 7) ;

  v.setVal(0,5.9) ;
  v.setVal(1,2.8) ;
  v.setVal(2,-4.9) ;
  v.setVal(4,14.9) ;
  v.afficheVec() ;

  return 0 ;
}
```

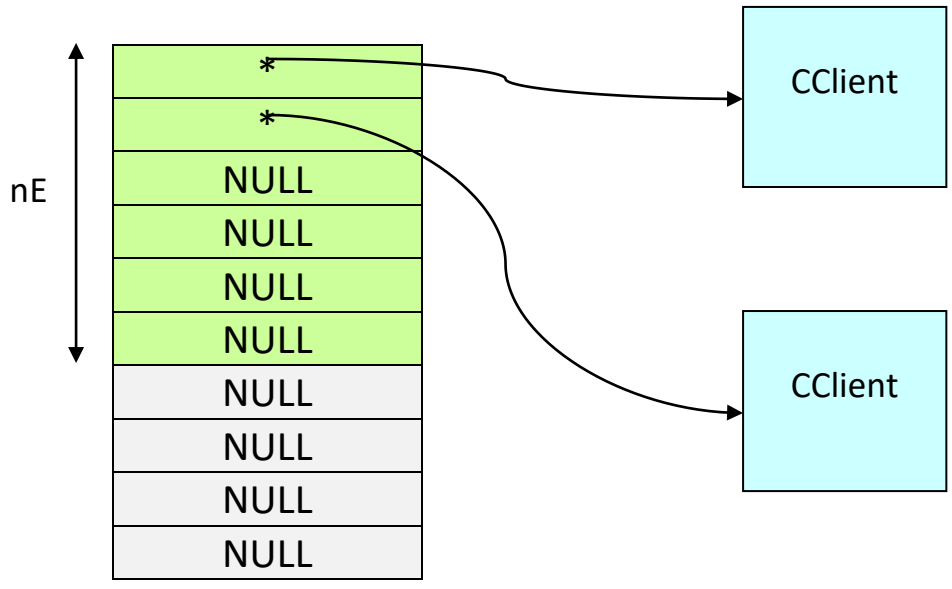
Un deuxième exemple : une classe « Vecteur d'objets CClient »

```
class CVecClient
{
  private :
    CClient * v[10];
    int nE;
    char checkIndex (const int i);

  public :
    CVecClient (void);
    ~CVecClient (void);

    int getDim (void) { return nE; }
    void setDim (const int nEl);

    void afficheVec (void);
    void entreVec (void);
    void traiteVec (void);
};
```

CClient

Nom
Sinistre
numPoli
somRemb

qui peut se représenter ainsi :



Attention au **destructeur** de cette classe !

Une classe Pile d'entiers : différentes implémentations

Tableau statique

```
class pile
{
  private:
    int nbreElem ;
    int tab[N_MAX];
    int pTop;

  public:
    pile(int n=N_MAX);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Une version un peu « moins statique »

```
class pile
{
  private:
    int tab[N_MAX];
    int pTop;


  public:
    pile(int n=N_MAX);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Tableau dynamique

```
class pile
{
  private:
    int *tab;
    int *pTop;

  public:
    pile(void);
    ~pile(void);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Liste chaînée dynamique

```
class pile
{
  private:
    noeud *pPile;

  public:
    pile(void);
    ~pile(void);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};

struct noeud {
  int valeur;
  noeud *suiv;
}
```

Les 4 versions ont **la même interface** !!!

Ces 4 versions de la classe pile pourront donc être utilisées au sein du programme suivant :

```
int main()
{
  pile maPile;

  int nbre = 5;

  for (int i=0 ; i<5 ; i++)
  {
    if (maPile.push(nbre++) == false)
      cout << "Pile pleine !" << endl;
  }

  maPile.affContenu();

  while (!maPile.empty())
  {
    cout << maPile.top() << endl ;
    maPile.pop();
  }
  return 0;
}
```

Cela est dû au fait que les 4 versions diffèrent seulement dans leur **implémentation** et pas dans leur **interface** (les méthodes ont conservé le même prototype) → c'est l'intérêt majeur de l'**encapsulation** !

Hiérarchie de structures de données

On se rend bien compte que les différents types de structures de données comme les listes, les listes triées, les piles, les files, les vecteurs, les listes circulaires, les arbres binaires ont des points communs et des différences.

Points communs :

- Stockage des données au sein de la classe « container »
- Allocation et désallocation du container

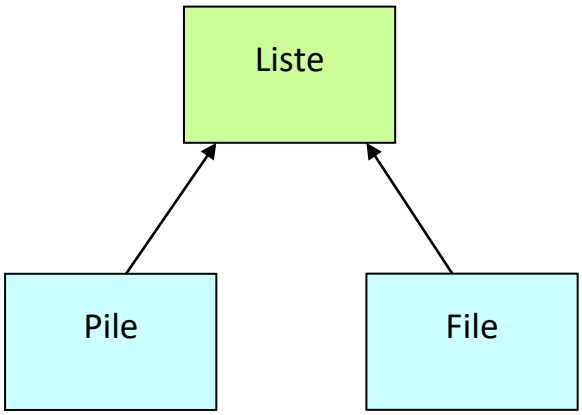
Différence principale :

- La manière dont les données sont insérées dans le container.

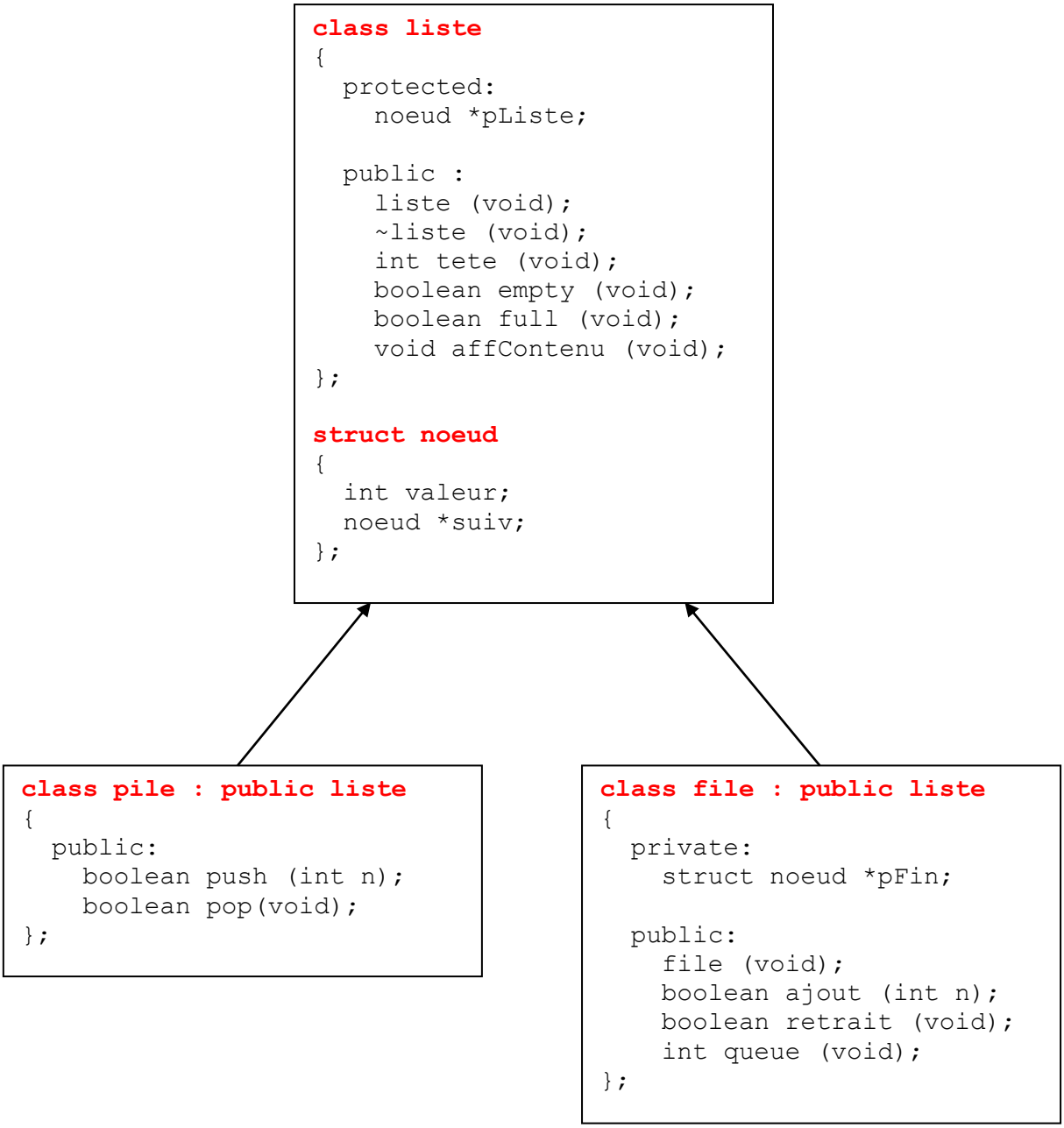
Il n'est donc pas absurde d'envisager de créer une **hiérarchie de structure de données** :

- Une **classe de base** reprendrait les **points communs** : on pourrait parler d'une classe Liste simple, sans méthode d'ajout et de retrait, mais comportant notamment un constructeur et destructeur qui seront utilisés par les classes dérivées
- **Dériver la classe pile et la classe file** (par exemple) **de cette classe de base commune**

Une première hiérarchie :



En entrant dans les détails :



Les classes dérivées ajoutent les fonctionnalités, ainsi que les données nécessaires pour les remplir, correspondant au type de collection qu'elles représentent.

Code de la classe de base : Liste

```
liste::liste (void)
{
  pListe = NULL;
}

liste::~liste (void)
{
  struct noeud *pCur, *pPrec;
  pCur = pListe;
  while (pCur != NULL)
  {
    pPrec = pCur; pCur = pCur->suiv;
    delete pPrec;
  }
}

int liste::tete (void)
{
  return pListe->valeur;
}

boolean liste::empty (void)
{
  return pListe == NULL? true : false;
}

boolean liste::full(void)
{
  struct noeud *pTr = new noeud;
  if (!pTr) return true;
  else return false;
}

void liste::affContenu (void)
{
  if (empty())
  {
    cout << "Liste vide\n";
    return;
  }
  struct noeud *pCur;
  pCur = pListe;
  while (pCur != NULL)
  {
    cout << pCur->valeur << "\n";
    pCur = pCur->suiv;
  }
}
```

Code de la classe dérivée « pile » :

```
boolean pile::push (int n) // Ajout au début
{
  if (! full())
  {
    struct noeud *pNouv = new noeud;
    pNouv->valeur = n;
    pNouv->suiv = pListe;
    pListe = pNouv;
    return true;
  }
  return false;
}

boolean pile::pop (void)  // Retrait au début
{
  if (!empty())
  {
    struct noeud *pTr = pListe;
    pListe = pListe->suiv;
    free(pTr);
    return true;
  }
  return false;
}
```

Code de la classe dérivée « File » :

```
file::file (void) : liste ()
{
  pFin = NULL;
}

boolean file::ajout (int n) // Ajout à la fin
{
  if (full()) return false;

  struct noeud *pNouv = new noeud;
  pNouv->valeur = n; pNouv->suiv = NULL;
  if (empty())
  {
    pListe = pNouv; pFin = pNouv;
  }
  else
  {
    pFin->suiv = pNouv; pFin = pNouv;
  }
  return true;
}

boolean file::retrait(void)  // Retrait au début
{
  if (!empty())
  {
    struct noeud *pTr = pListe;
    pListe = pListe->suiv;
    free(pTr);
    return true;
  }
  return false;
}

int file::queue (void)
{
  return pFin->valeur;
}
```

Une hiérarchie plus complète

En vue de créer cette hiérarchie, il convient de créer une classe qui sera la **super-classe** de la hiérarchie. Cette classe se devra d'être **la plus générale possible** afin de contenir des informations et des méthodes qui seront utiles à la fois à un vecteur, à une file, à une liste triée ou encore à un arbre binaire.

On créera donc une classe « **structure** » qui contient :

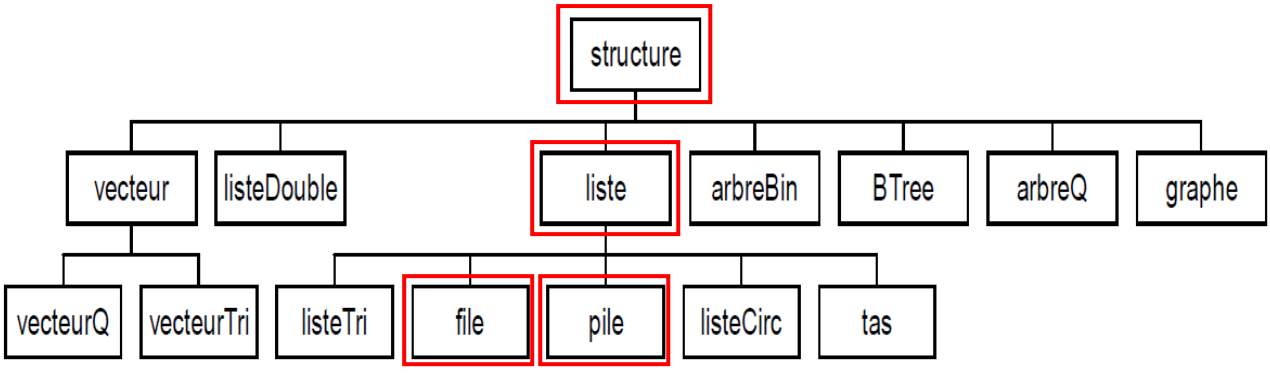
- Le nombre d'éléments dans le container
- Des méthodes indiquant si la structure est vide ou pleine
- Des méthodes renvoyant l'élément de tête, permettant l'ajout ou le retrait d'un élément

En réalité, **on n'instanciera jamais cette classe**. Pour cette raison, on rendra la classe **structure** **abstraite**.

La classe abstraite « structure » :

```
class structure
{
  protected :
    int nbreElem;

  public :
    structure () { nbreElem = 0; }
    virtual int tete (void) = 0;    // devra être redéfinie
    virtual boolean empty (void);   // dans les classes dérivées
    virtual boolean full (void);
    virtual boolean ajout (int n);
    virtual boolean retrait (void);
};
```

Finalement, on pourrait imaginer créer une hiérarchie de données selon le schéma suivant :



En entrant dans les détails :

Classe abstraite pure

```
class structure
{
  protected :
    int nbreElem;

  public :
    structure () { nbreElem = 0; }
    virtual int tete (void) = 0;
    virtual boolean empty (void);
    virtual boolean full (void);
    virtual boolean ajout (int n);
    virtual boolean retrait (void);
};
```

```
class liste : public structure
{
  protected :
    noeud *pListe;

  public :
    liste (void);
    ~liste (void);
    int tete (void);
    boolean empty (void);
    boolean full (void);
    void affContenu (void);
};

struct noeud {
  int valeur;
  noeud *suiv;
};
```

```
class file : public liste
{
  private :
    noeud *pFin;

  public :
    file (void);
    boolean ajout (int n);
    boolean retrait (void);
    int queue (void);
};
```

```
class pile : public liste
{
  public:
    boolean push (int n);
    boolean pop(void);
};
```

La notion d'itérateur

Un **itérateur** est un objet qui permet de **balayer une structure de données**, et d'**accéder ainsi à tous les objets** qu'elle contient, **selon un ordre bien établi**.

A quoi cela sert-il ?

Il est parfois utile de pouvoir avoir accès à tous les éléments contenus au sein d'une structure de données pour, par exemple,

- les afficher un par un
- leur faire subir un traitement via l'appel d'une fonction
- etc…

Sans l'itérateur, il serait indispensable de connaître de manière approfondie l'implémentation du container (indice, pointeurs, …) mais ceci est à l'opposé du principe d'encapsulation.

La notion d'itérateur permet donc de définir un **procédé standardisé d'accès** aux objets contenus au sein d'un container, et cela, sans en connaître l'implémentation.

Les caractéristiques d'un itérateur

1. Un itérateur doit être fortement lié à la classe « container » qu'il doit manipuler. Il doit en connaître les détails d'implémentation et accéder à ses données membres.

   Dès lors, La classe itérateur doit être **amie** de la classe container

2. L'itérateur doit être **initialisé** par son constructeur, sur le début de la structure de données qu'il permet de manipuler. Il doit pouvoir être ramené au début.

3. Il faut pouvoir **détecter la fin** de la structure de données et aussi pouvoir **réinitialiser** l'itérateur.

4. On passe d'un élément de la structure au suivant au moyen de **l'opérateur ++** : celui-ci devra donc être surchargé.

5. L'itérateur doit pouvoir **renvoyer l'élément** (objet) auquel il est parvenu.

Un itérateur pour un tableau (Retour sur la classe vecteur) :

```
class CIntVecteur
{
  private :
    int *tab;
    int taille;

  public :
    CIntVecteur (int n) ;
    ~CIntVecteur (void) { delete [] tab;}
    void affContenu (void);
    friend class CIntVecIter; // la classe itérateur
};

CIntVecteur::CIntVecteur(int n) : taille(n)
{
  tab = new int[n];
  memset(tab, 0, taille * sizeof(int) );
}

class CIntVecIter
{
  private :
    CIntVecteur& p;
    int *pData;

  public :              // Pointeur initialisé au début
    CIntVecIter (CIntVecteur& s) : p(s), pData(s.tab) {};
    bool end() const  // Détecte la fin du vecteur
    {
      if (pData - p.tab >= p.taille) return true;
      else return false;
    }
    void reset() { pData = p.tab; } // Réinitialize l'itérateur
    bool operator++()  // Passe à l'élément suivant
    {
      if (!end())
      {
        pData++;
        return true;
      }
      else return false;
    }
    bool operator++ (int) { return operator++(); }
    operator int() const { return *pData; } // Renvoie l'élément
    void insert(int n) { *pData = n; }
};
```

```
void CIntVecteur::affContenu (void)
{
  CIntVecIter it(*this);
  cout << "( ";
  while (!it.end())
  {
    cout << it << " , ";
    it++;
  }
  cout << "\x8\x8)" << endl;
}

int main()
{
  CIntVecteur v(10);
  CIntVecIter it(v);

  for (int i=0 ; !it.end() ; it++,i++) it.insert(1 << i);
  v.affContenu();

  it.reset();

  cout << "Nouvelle valeur du premier element : ";
  int nb;
  cin >> nb;
  it.insert(nb);
  v.affContenu();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
( 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 , 512 )
Nouvelle valeur du premier element : 55
( 55 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 , 512 )
bash-3.00$
```

Un itérateur pour une liste (chaînée dynamique) :

```
struct noeud
{
  int valeur;
  noeud *suiv;
  noeud (int n, noeud *s) : valeur(n), suiv(s) {};
  noeud (void) : valeur(-9999), suiv (0) {};
};

class CIntListe
{
  private :
    noeud *pListe;

  public :
    CIntListe (void) : pListe(0) {};
    ~CIntListe (void);
    bool empty(void) { return pListe ? false : true; }
    int head(void) { return empty() ? 0 : pListe->valeur; }
    void insert(int n);
    void removeHead (void);
    void removeTail (void);
    void affContenu (void);
    friend class CIntLisIter; // l'itérateur ami
};

CIntListe::~CIntListe (void)
{
  noeud *pCur, *pPrec;
  pCur = pListe;
  while (pCur != NULL)
  {
    pPrec = pCur; pCur = pCur->suiv; delete pPrec;
  }
}

void CIntListe::insert (int n)
{
  pListe = new noeud(n,pListe);
}

void CIntListe::removeHead (void)
{
  if (pListe)
  {
    noeud *pWork(pListe);
    pListe = pListe->suiv; delete pWork;
```

```
  }
}

void CIntListe::removeTail(void)
{
  if (!pListe) return;
  if (!pListe->suiv)
  {
    delete pListe;
    pListe =0;
    return;
  }
  noeud *pWork(pListe), *pPrec;
  while (pWork->suiv)
  {
    pPrec = pWork;
    pWork = pWork->suiv;
  }
  delete pWork;
  pPrec->suiv = 0;
}

class CIntLisIter
{
  private :
    CIntListe& lis;
    noeud *pCur;

  public :                  // Pointeur initialisé au début
    CIntLisIter (CIntListe& l) :lis(l),pCur(l.pListe) {};
    bool end() { return pCur == 0; } // Détecte la fin de la liste
    void reset() { pCur = lis.pListe; } // Réinitialize l'itérateur
    bool operator++ ()// Passe à l'élément suivant
    {
      if (pCur)
      {
        pCur = pCur->suiv;
        return true;
      }
      else return false;
    }
    bool operator++ (int) { return operator++(); }
    operator int() const { return pCur->valeur; }
         // Renvoie l'élément
    int& operator& (void) { return pCur->valeur; }
         // Renvoie l'élément par référence
};
```

```
void CIntListe::affContenu (void)
{
  CIntLisIter it(*this);
  cout << "( ";
  while (!it.end())
  {
    cout << (int)it << " , ";
    it++;
  }
  cout << "\x8\x8)" << endl;
}

int main()
{
  CIntListe l;

  for (int i=0 ; i<4 ; i++) l.insert(1 << i);
  l.affContenu();

  CIntLisIter it(l);
  cout << "Examen de la liste :" << endl;
  for (it.reset() ; !it.end() ; it++)
  {
    cout << "* " << (int)it << endl;
    cout << "--> nouvelle valeur = "; int nb; cin >> nb;
    &it = nb;  // on peut modifier l'élément car retour par réf.
  }

  l.affContenu();

  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ a.out
( 8 , 4 , 2 , 1 )
Examen de la liste :
* 8
--> nouvelle valeur = 10
* 4
--> nouvelle valeur = 20
* 2
--> nouvelle valeur = 30
* 1
--> nouvelle valeur = 40
( 10 , 20 , 30 , 40 )
bash-3.00$
```

On remarque que

- Les itérateurs de vecteur et de liste présentent des fonctionnalités analogues. L'interface est en fait la même → **procédé d'accès standardisé**

- Le programmeur utilisant l'itérateur (par exemple dans le main ou la fonction affContenu) ne sait absolument pas comment le container est implémenté → il se contente d'appeler les méthodes de l'itérateur.

Remarque :

A l'image de notre hiérarchie de structure de données, on peut imaginer une **hiérarchie d'itérateurs** ☺…