Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°6

Exercice 1: classe `vector` et itérateur

On veut définir un vecteur générique `vector` similaire à celui de la bibliothèque standard. Ce vecteur est dynamique, redimensionnable, et déplaçable.

On donne dans un premier temps la définition de base de la classe `vector`.

1. Donner la définition des champs privés de cette classe qui stocker des éléments de type générique `T`.
2. Donner les accesseurs au nombre d'éléments stocké et le nombre d'éléments alloués.
3. Définir deux fonctions privées : `alloc` qui alloue la mémoire pour un tableau de n éléments de type `T` (sans constructeur), `destroy` qui lance le destructeur sur tous les éléments.
4. Définir les constructeurs avec la taille allouée (et taille nulle), par défaut (avec une taille allouée par défaut).
5. Définir les constructeurs par copie et par déplacement.
6. Définir les assignations par copie et par déplacement.
7. Définir le destructeur.
8. Définir la méthode `push_back` qui ajoute un élément à la fin du vecteur. On donnera la version par copie ou par déplacement du paramètre.
9. Définir la méthode `emplace_back` qui permet de passer directement les arguments du constructeur à `push_back`.
10. Définir le méthode `pop_back` qui retourne par déplacement le dernier élément du vecteur.
11. Définir le constructeur à partir d'une liste d'initialisation. On donnera un exemple d'utilisation.
12. Définir les opérateurs `[]` afin de permettre un accès direct (en lecture et en écriture) sur les éléments du vecteur. Il devra être fonctionnel sur un vecteur constant.

On veut maintenant définir un itérateur de cette classe.

13. Rappeler le rôle d'un itérateur, et comment il est utilisé.
14. Quels sont les données stockées dans l'itérateur ?
15. Donner l'ensemble des méthodes que l'itérateur doit définir afin d'être fonctionnel.
16. Donner les constructeurs (privé avec un pointeur, public par copie). Quel est l'intérêt d'avoir un constructeur privé ?
17. Comment sont définies les méthodes begin() et end() et où sont-elles définies ?
18. Donner la surcharge de l'opérateur `++`
19. Donner la surcharge des opérateurs `==` et `!=`.
20. Donner la surcharge de l'opérateur `*`.
21. Reprendre alors l'utilisation de l'itérateur, et voir comment chaque élément ci-dessus participe à son fonctionnement.
22. Même question pour définir un itérateur constant, à savoir sur lequel il n'est pas possible de modifier les objets du `vector`.

Solution:

1. champs privés de la classe de type générique `T` :

```
template <class T> class vector {
protected:
  static const size_t      DefSize = 4;
  size_t   Size,Allocated;
  T        *Data;
public:
  ...
};
```

2. Accesseurs :

```
size_t size() { return Size; }
size_t allocated() { return Allocated; }
```

3. fonctions privées : `alloc` et `destroy`

```
void allocate(size_t n) {
  Data = reinterpret_cast<T*>(:: operator new(n * sizeof(T)));
  Allocated = n;
}
void destroy() {
  for (size_t i = 0; i < Size; i++) Data[i].~T();
  Size = 0;
}
```

Ces fonctions sont privés car se sont des fonctions techniques qui ne devraient jamais être appelée par l'utilisateur.

4. constructeurs par taille allouée et par défaut :

```
vector(size_t n) : Size(0) { allocate(n); }
vector() : vector(DefSize) {}
```

5. constructeurs par copie et par déplacement.

```
vector(const vector& orig) : vector(orig.Size) {
  for (size_t i = 0; i < orig.Size; i++) new(&Data[i]) T(orig.Data[i]);
  Size = orig.Size;
}
vector(vector&& orig) :
  Size(orig.Size), Allocated(orig.Allocated), Data(orig.Data) {
  orig.Data = nullptr;
  orig.Size = orig.Allocated  = 0;
}
```

6. assignations par copie et par déplacement.

```
vector& operator=(const vector& orig) {
  if (this != &orig) {
    if (Allocated < orig.Size) {
      destroy();
      ::operator delete(Data);
      allocate(orig.Size);
    }
    if (Size < orig.Size) {
      for (size_t i = 0; i < Size; i++) Data[i] = orig.Data[i];
      for (size_t i = Size; i < orig.Size; i++) new(&Data[i]) T(orig.Data[i]);
    } else {
      for (size_t i = 0; i < orig.Size; i++) Data[i] = orig.Data[i];
      for (size_t i = orig.Size; i < Size; i++) Data[i].~T();
    }
    Size = orig.Size;
  }
  return *this;
}
```

```
vector& operator=(vector&& orig) {
  if (this != &orig) {
    std::swap(Size,orig.Size);
    std::swap(Allocated, orig.Allocated);
    std::swap(Data, orig.Data);
  }
  return *this;
}
```

7. destructeur.

```
~vector() {
  destroy();
  ::operator delete(Data);
  Allocated = 0;
};
```

8. `push_back` :

```
template <class U> void push_back(U&& x) {
  if (Size < Allocated) {
    T  *tmp = Data;
    allocate(Allocated * 1.2f); // +20%
    for(size_t i=0;i<Size;++i) Data[i] = std::move(tmp[i]);
    ::operator delete(tmp);
  }
  new(&Data[Size]) T( std::forward<T>(x) );
  ++Size;
}
```

9. `emplace_back` : pousse une rvalue (issue du constructeur) en faisant suivre les paramètres variables.

```
template <class... Args> void emplace_back(Args&&... args ) {
  push_back(T(std::forward<Args>(args)...));
}
```

Exemple d'utilisation :

```
struct A {
  int x,y;
  A() : x{}, y{} {}
  A(int u) : x(u), y(u) {}
  A(int u, int v) : x(u), y(v) {}
};
int main() {
  vector<A> vA;
  vA.emplace_back();
  vA.emplace_back(1);
  vA.emplace_back(2,3);
  for(vector<A>::const_iterator i=vA.cbegin();i != vA.cend();++i)
  std::cout << "(" << (*i).x << "," << (*i).y << ")␣";
  std::cout << std::endl;
  // sortie: (0,0) (1,1) (2,3)
}
```

10. `pop_back`

```
T&& pop_back() {
  assert(Size!=0);
  --Size;
  return std::move( Data[Size] );
}
```

11. constructeur avec liste d'initialisation

```
vector(std::initializer_list<T> list) : vector(list.size()){
  for(const T &x : list) push_back(x);
}
```

Noter que une `initializer_list` se passe toujours par valeur. C'est cette définition qui permet l'écriture :

```
vector<int>      v5 = {2, 4, 7, 10};
}
```

12. opérateurs `[]`

```
const T& operator[](int i) const { assert(Data!=nullptr); return Data[i]; }
T& operator[](int i) { assert(Data!=nullptr); return Data[i]; }
```

On veut maintenant définir un itérateur de cette classe.

13. rôle d'un itérateur : manière générique de parcours les éléments contenus dans un conteneur.

```
vector<int>  v = {1,2,3,4};
for(vector<int>::iterator i=v.begin(); i!=v.end(); ++i)
  std::cout << i << "␣";
std::cout << std::endl;
```

`vector<int>::iterator` est un classe interne permettant d'itérer sur la classe. A noter que l'on peut avoir autant d'itérateur qu'on le souhaite sur un containeur.

14. données stockées dans l'itérateur : juste un pointeur vers l'élément courant.

```
class iterator {
private:
  T       *ptr;
public:
};
```

15. méthodes itérateur :

— méthode `begin()`/`end()` : méthodes de `vector` renvoyant l'itérateur sur le premier et le dernier élément de la liste.
— constructeur par copie.
— opérateur `!=` qui permet de comparer deux itérateurs.
— opérateur `++` sur un itérateur
— opérateur `*` qui retourne la valeur de l'itérateur.

16. constructeurs (privé avec un pointeur, public par copie)

```
private: iterator(T *v) : ptr(v) {}
public : iterator(const iterator &v) : ptr(v.ptr) {}
```

L'intérêt d'avoir un constructeur privé est de permettre la création d'un itérateur à partir d'un objet `vector` (déclarée comme `friend` dans la classe `iterator`).

17. `begin()` et `end()` :

méthodes de `vector` :

```
iterator begin() { return iterator(Data); }
iterator end() { return iterator(Data+Size); }
```

18. surcharge de l'opérateur `++`

```
iterator& operator++() { ++ptr; return *this; }
iterator operator++(int) { iterator tmp(ptr); ++ptr; return tmp; }
```

19. surcharge des opérateurs `==` et `!=`.

```
bool operator==(const iterator &other) const { return ptr == other.ptr; }
bool operator!=(const iterator &other) const { return ptr != other.ptr; }
```

20. surcharge de l'opérateur `*`.

```
T& operator*() { return *ptr; }
```

21.

22. itérateur constant

```
class const_iterator {
private:
  T       *ptr;
  const_iterator(T *v) : ptr(v) {}
public:
  const_iterator(const const_iterator &v) : ptr(v.ptr) {}
  const_iterator& operator++() { ++ptr; return *this; }
  const_iterator operator++(int) { const_iterator tmp(ptr); ++ptr; return tmp; }
  bool operator==(const_iterator other) const { return ptr == other.ptr; }
  bool operator!=(const_iterator other) const { return ptr != other.ptr; }
  const T& operator*() { return *ptr; }
  friend class vector;
};
const_iterator cbegin() { return const_iterator(Data); }
const_iterator cend() { return const_iterator(Data+Size); }
```

Exercice 2: classe `list` et itérateur

On veut définir un vecteur générique `list` similaire à celui de la bibliothèque standard. Cette liste est déplaçable.

On donne dans un premier temps la class `Cell` interne de `list` :

1. Donner la définition de la structure `Cell` et des champs privés de list.
2. Donner les constructeurs de `Cell`.

3. Définir la méthode privée `clear` qui détruit la liste passée en paramètre.
4. Définir la méthode privée `duplicate` qui duplique la liste passée en paramètre
5. Définir la méthode privée `copy` qui copie une liste dans une autre.
6. Écrire les constructeurs par défaut, par copie et par déplacement, et destructeur.
7. Écrire l'assignation par copie et par déplacement.
8. Écrire la méthode `push_front` en utilisant une référence universelle et `emplace_front`.
9. Écrire le constructeur avec une liste d'initialisation.
10. Écrire la méthode `pop_front`.
11. Écrire l'itérateur sur la liste chainée.

Solution:

1. structure `Cell`

```
class list {
private:
  struct Cell {
    T     val;
    Cell  *nxt;
    template <class U> explicit Cell(U&& v)
      : val(std::forward<T>(v)), nxt(nullptr) {};
    template <class U> Cell(U&& v, Cell *t)
      : val(std::forward<U>(v)), nxt(t) {};
  };
  // only field
  Cell  *head;
}
```

2. constructeurs Cell

```
template <class U> explicit Cell(U&& v)
  : val(std::forward<T>(v)), nxt(nullptr) {};
template <class U> Cell(U&& v, Cell *t)
  : val(std::forward<U>(v)), nxt(t) {};
```

3. méthode privée `clear`

```
Cell *clear(Cell *start) {
  while(start != nullptr) {
    Cell *tmp = start;
    start = start->nxt;
    delete tmp;
  }
  return nullptr;
}
```

4. méthode privée `duplicate`

```
Cell *duplicate(Cell *from) {
  Cell *start = nullptr;
  if (from != nullptr) {
    start = new Cell(from->val, nullptr);
    Cell  *cur = start, *curc = from;
    while (curc->nxt != nullptr) {
      curc=curc->nxt;
      cur->nxt = new Cell(curc->val, nullptr);
      cur=cur->nxt;
    }
  }
  return start;
}
```

5. méthode privée `copy`

```
Cell *copy(Cell *dst, Cell *src) {
  Cell *start = dst;
  if (src == nullptr) return clear(dst);
  if (dst == nullptr) return duplicate(src);
  while ((src->nxt !=nullptr) && (dst->nxt != nullptr)) {
    dst->val = src->val;
    src=src->nxt;
    dst=dst->nxt;
  }
  if (dst->nxt == nullptr) dst->nxt = duplicate(src);
  if (src->nxt == nullptr) dst->nxt = clear(dst->nxt);
  return start;
}
```

6. constructeurs par défaut, par copie et par déplacement, et destructeur.

```
list() : head(nullptr) {};
list(const list& other) { head = duplicate(other.head); }
list(list&& other) : head(other.head) { other.head = nullptr; }
~list() { head = clear(head); }
```

7. assignation par copie et par déplacement

```
list& operator=(const list& other) {
  if (this != &other) head = copy(head,other.head);
  return *this;
}
list& operator=(list&& other) {
  if (this != &other) std::swap(head,other.head);
  return *this;
}
```

8. méthode `push_front` et `emplace_front`

```
template <class U> void push_front(U&& x) {
  head = new Cell(std::forward<U>(x), head);
}

template <class... Args> void emplace_front(Args&&... args ) {
  push_front(T(std::forward<Args>(args)...));
}
```

9. constructeur avec une liste d'initialisation.

```
list(std::initializer_list<T> init) : list() {
  for(const T &x : init) push_front(x);
}
```

10. `pop_front`

```
T&& pop_front() {
  if (head == nullptr) {
    static T invalid;
    return move(invalid);
  } else {
    Cell *tmp = head;
    head=head->nxt;
    T&&  val = std::move(tmp->val);
    ::operator delete(tmp);
    return std::move(val);
  }
}
```

11. itérateur

    A intégrer dans la class `list` :

```
class iterator {
  private:
  Cell *ptr;
  iterator(Cell *v) : ptr(v) {}
  public:
  iterator(const iterator &v) : ptr(v.ptr) {}
  iterator& operator++() { ptr=ptr->nxt; return *this; }
  iterator operator++(int) { iterator tmp(ptr); ptr=ptr->nxt; return tmp; }
  bool operator==(iterator other) const { return ptr == other.ptr; }
  bool operator!=(iterator other) const { return ptr != other.ptr; }
  T& operator*() { return ptr->val; }
  friend class list;
};
iterator begin() { return iterator(head); }
iterator end() { return iterator(nullptr); }
```

Exercice 3: classe `set` et itérateur

On veut définir un ensemble générique `set` similaire à celui de la bibliothèque standard. Un `set` sera déplaçable.

1. Donner la définition de la structure `node` et des champs privés de `set`. `node` contiendra trois pointeurs, vers ses deux fils et son parent.
2. Donner les constructeurs de `node` : avec valeur + parent, et par copie d'un autre arbre.
3. Donner le destructeur de `node`. Il devra aussi détruire tout la hiérarchie en dessous.
4. Donner la méthode `insert` qui permet d'insérer un n ?ud dans l'arbre en fonction de sa valeur. Aucun équilibrage ne sera fait.
5. Expliquer comment on peut effectuer itérativement le parcours préfixe complet d'un arbre en utilisant deux pointeurs (un sur la position courante, un sur la position précédente).
6. Donner la fonction qui permet, à partir d'un élément, de passer à l'élément suivant.
7. Donner les constructeurs par défaut, par copie et par déplacement.
8. Donner les assignations par copie et par déplacement.
9. Donner la fonction d'insertion d'un élément.
10. Donner le destructeur.
11. Donner l'itérateur.

Solution:

1. structure `node` et `set`.

```
template <typename T> class set {
private:
  struct node {
    node    *top, *left, *right;
    T       value;
    // méthodes de node
    ...
  };
  node *root;
public:
  // méthodes de set
  ...
}
```

2. constructeurs de `node`

```
node(const T& v, node *up)
  : value(v), top(up), left(nullptr), right(nullptr) {}
node(node *n, node *parent) :
  top( parent ),
  left( new node(value->left ,this) ),
  right( new node(value->right ,this) ),
  value(n->value) {}
```

3. destructeur de `node`

```
~node() {
  if (left != nullptr)  { left->~node();  delete left;  }
  if (right != nullptr) { right->~node(); delete right; }
  // on ne delete pas this: c'est fait par le parent (cf lignes ci-dessus)
}
```

4. méthode `insert` de `node`

```
void insert(const T& v) {
  node *cur = this;
  while (1) {
    if (v < cur->value) {
      if (cur->left == nullptr) { cur->left = new node(v,cur); break; }
      else cur = cur->left;
    }
    else if (v > cur->value) {
      if (cur->right == nullptr) { cur->right = new node(v,cur); break; }
      else cur = cur->right;
    }
    else break; // déjà présent
  }
}
```

5. parcours préfixe non récursif

   — on part de la racine : on descend à gauche tant que c'est possible.

   — lorsque l'on arrive à la feuille la plus en bas à gauche, on affiche sa valeur.

   — en remontant d'un niveau, on affiche la valeur au noeud.

   — on recommence en descendant à droite le plus à gauche possible.

   — lorsque l'on remonte depuis le fils droit, on continue à remonter.

6. fonction next.

```
static node* go_up   (node* &cur)
  { node* lst=cur; cur=cur->top;    return lst; }
static node* go_left (node* &cur)
  { node* lst=cur; cur=cur->left;   return lst; };
static node* go_right(node* &cur)
  { node* lst=cur; cur=cur->right;  return lst; };
static bool next(node* &cur, node* &lst) {
  bool    found=false;
  if (cur == nullptr) return found;
  if (lst == cur->top) { // vient du haut
    if (cur->left == nullptr) { // rien à gauche
      found = true;
      if (cur->right == nullptr) lst = go_up(cur);
// rien à droite: on remonte
      else lst = go_right(cur); // fils droit: on descend
    } else lst = go_left(cur);  // fils gauche: on descend

  } else if (lst == cur->left) { // vient de gauche
    found = true;
    if (cur->right == nullptr) lst = go_up(cur);  // rien à droite: on remonte
    else lst = go_right(cur); // fils droit: on descend
  } else if (lst == cur->right) { // vient de droite
    lst = go_up(cur); // on remonte
  } else throw;
  return found;
}
```

7. constructeurs par défaut, par copie et par déplacement de `set`.

```
set() : root(nullptr) {}
set(const set& s) : root( new node(s.root) ) {}  // construction récursive
set(set &&s) : root(s.root) { s.root = nullptr; }
```

8. assignations par copie et par déplacement de `set`.

```
set& operator=(const set &s) {
  if (this != &s) { // plus simple:
    delete root;
    root = new node(s.root);
  }
  return this;
}
set& operator=(set &&s) {
  std::swap(root,s.root);
}
```

9. insertion d'un élément de `set`.

```
void insert(int v) {
  if (root == nullptr) root = new node(v,nullptr);
  else root->insert(v);
}
```

10. destructeur de `set`.

```
~set() {
  delete root;
}
```

11. itérateur de `set`.

```
class iterator {
private:
  node    *cur,*lst;
  iterator(node *c, node *l) : cur(c), lst(l) {}
  iterator() : cur(nullptr), lst(nullptr) {}
public:
  /// constructeurs
  iterator(const iterator &s) : cur(s.cur), lst(s.lst) {}
  /// itération vers l'élément suivant
  iterator& operator++() {
    while (cur != nullptr) {
      bool found = next(cur,lst);
      if (found) break;
    }
    return *this;
  }
  /// déréférencement
  const T& operator*() const {
    return lst->value;
  }
  T& operator*() { return lst->value; }
  /// comparaison de deux itérateurs
  friend bool operator!=(const iterator &it1, const iterator &it2)
    { return (it1.cur != it2.cur) || (it1.lst != it2.lst); }
  friend class set;
};
/// itérateur vers le premier élément de l'arbre
iterator begin() { return ++iterator(root,nullptr); }
/// itérateur vers le dernier élément (parcours fini = pile vide)
iterator end() { return iterator(nullptr,root); }
```