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

TD N°10

Exercice 1: Vector

On voudrait effectuer une implémentation de `vector` en faisant attention à la gestion des exceptions.

Pour représenter un vecteur, on utilisera 4 champs de base :

- `elem` : un pointeur sur le début de la zone allouée.
- `space` : un pointeur sur le premier élément libre.
- `last` : un pointeur sur l'élément qui suit le dernier élément.
- `alloc` : un allocateur

On pose alors les questions suivantes :

1. Rappeler le fonctionnement de base d'un allocateur.
2. Donner une implémentation possible de l'allocateur standard.
3. Quelles sont les exceptions pouvant être lancé par cet allocateur ?
4. Quelle serait alors la déclaration de la classe `vector` ?
5. Soit l'implémentation suivante du constructeur qui initialise un vecteur de `n` éléments, chacun initialisé à la valeur `val`.

```
template<class T, class A>
vector<T,A>::vector(size_t n, const T& val, const A& a = std::allocator())
:alloc{a} {
    elem = alloc.allocate(n);
    space = last = elem+n;
    for (T* p = elem; p!=last; ++p) a.construct(p,val);
}
```

   Quels sont les problèmes potentiels de ce code ?
6. Quelles sont alors les conséquences ?
7. Proposer alors une version du code de ce constructeur qui résout ces problèmes (avec une garantie d'exception forte).
8. Y-a-t-il un moyen d'utiliser le principe RAII pour la définition de la classe ?
9. Quelle doivent être les propriétés de la classe `vector_base` chargée de la gestion des ressources de la classe ?
10. Donner la définition de la classe interne `vector_base` chargée de la gestion des ressources de la classe.
11. Donner la définition du constructeur (= acquisition des ressources pour `n` éléments) et du destructeur pour la classe `vector_base`
12. Donner la définition du constructeur et de l'assignation par déplacement pour la classe `vector_base`.
13. Donner alors la définition de la classe `vector` utilisant cette classe de stockage.
14. Donner la définition complète de l'itérateur de cette classe. Elle devra permettre l'appel aux méthodes construct ou destruct à la position de l'itérateur.

15. Donner la définition d'une fonction template `void uninitialized_fill(It beg, It end, const T& x)` dont le but est de construire par copie des éléments de type `T` dans une zone de mémoire `[beg,end[` déjà allouée et non initialisée de façon à ce qu'elle ait une garantie d'exception forte.
16. En déduire une nouvelle version du constructeur qui initialise un vecteur de `n` éléments initialisés à la valeur `val` avec une garantie d'exception forte.
17. Donner la définition d'une fonction template `void uninitialized_copy(It1 beg1, It1 end1, It2 beg2)` dont le but est de construire par copie les éléments de `[beg1,end1[` dans la zone mémoire déjà allouée commençant par `beg2` de façon à ce qu'elle ait une garantie d'exception forte.
18. En déduire le constructeur par copie qui possède une garantie d'exception forte.
19. Donner le constructeur par déplacement et l'assignation par déplacement.
20. Que peut-on dire en terme d'exception sur les deux méthodes précédentes ?
21. Écrire une méthode `destroy_elements` qui détruit les éléments du vecteur.
22. Que peut-on dire en terme d'exception de la méthode `destroy_elements` ?
23. Définir les méthodes `size()` et `capacity()`.
24. Définir l'assignation par copie. On étudiera sa garantie d'exception.
25. Définir la méthode `reserve(size_t n)` qui permet de faire en sorte que la mémoire allouée est au moins de `n`.
26. Définir la méthode `resize(size_t n, const T& val = T())` qui fixe la taille du vecteur à n en initialisant toutes les nouvelles valeurs créées (s'il y en a à `val`).
27. Définir la méthode `push_back(const T&x)` qui permet d'ajouter un élément au vecteur avec une garantie forte.
28. Conclure sur la gestion des exceptions effectuées.

Solution:

1. Un allocateur possède 4 méthodes :
   - `T* allocate<T>(size_t n)` : alloue de la place de stockage pour n objets de type `T` (sans constructeur), et retourne un pointeur vers la zone allouée.
   - `void deallocate<T>(T *p, size_t n` : désalloue la zone mémoire sur laquelle pointe `p` (zone de `n` éléments de type `T`).
   - `void construct<T>(T *p, const T& v)` : construction par copie en place d'un objet `T` initialisé avec `v` à l'adresse mémoire `p`.
   - `void construct<T>(T *p, T&& v)` : construction par déplacement en place d'un objet `T` avec `v` à l'adresse mémoire `p`.
   - `void destroy<T>(T *p)` : destruction en place de l'objet à l'adresse `p` (sans désallouer l'espace qu'il occupe).
2. ce sont les suivants :

```
template <typename T> class Allocator {
public:
  T* allocate(size_t n) {
        return reinterpret_cast<T*>(::operator new(n * sizeof (T)));
  }
  void deallocate(T* p, size_t n) {
        ::operator delete( reinterpret_cast<void*>(p) );
  }
  static void construct(T* p, const T& v) {
    ::new (static_cast<void*>(p)) T(v);
  }
  static void construct(T* p, T&& v) {
    ::new (static_cast<void*>(p)) T(std::move(v));
  }
  static void destroy(T* p) { p->~T(); }
};
```

3. `allocate` peut lancer `std::bad_alloc` si l'allocation échoue, le constructeur par copie peut échouer (construire un `T` peut conduire à des allocations mémoires externes).

4. On propose le code suivant :

```
template<class T, class A = allocator<T>>
class vector {
private:
    T* elem;  // star t of allocation
    T* space;  // end of element sequence,
               // star t of space allocated for possible expansion
    T* last;  // end of allocated space
    A alloc;  // allocator
public:
    // ...
};
```

5. Deux problèmes potentiels dans ce code :
   - `allocate` peut échouer s'il n'y a plus de mémoire disponible.
   - le constructeur par copie de `T` peut échouer s'il ne peut pas copie `val`.

   Note : on pourrait aussi parler de la construction par copie de l'allocateur, mais le standard indique qu'il ne le fait pas.

6. **Rappel :** un objet n'est pas considéré comme construit tant que son constructeur ne s'est pas terminé. En conséquence, si une exception est lancée avant la fin de l'exécution du constructeur, l'objet n'est pas construit, et son destructeur ne sera pas appelé (= pas de destruction partielle associée à une construction partielle).

   Donc, l'objet `vector` ne sera pas construit, et son destructeur ne sera pas appelé.

   Dans le détail maintenant :
   - si `allocate` échoue, le `throw` fera sortir du constructeur avec qu'aucune ressource ne soit allouée. Donc, tout va bien.
   - si le CC de `T` échoue, des ressources mémoires ont été allouée (par `allocate`) qui doivent être libérés pour ne pas produire de fuite mémoire.
     si ceci se produit après que quelques éléments de type `T` ait été construit (donc sans les construire tous), ces objets `T` peuvent détenir des ressources provoqueront aussi des fuites.

7. On propose le code suivant :

```
template<class T, class A>
vector<T,A>::vector(size_t n, const T& val = T(), const A& a = A())
:alloc{a} {
    elem = alloc.allocate(n); // get memory for elements
    iterator p;
    try {
        iterator end = elem+n;
        for (p=elem; p!=end; ++p)
            alloc.construct(p,val);  // construct element
        last = space = p;
    }
    catch (...) { // pas de souci car rethrow l'exception
        for (iterator q = elem; q!=p; ++q)
            alloc.destroy(q);  // destroy constructed elements
        alloc.deallocate(elem,n);  // free memory
        throw; // rethrow
    }
}
```

8. Oui, en définissant un classe de base interne qui sera chargé de l'acquisition des ressources.
9. Ses propriétés doivent être les suivantes :
   — elle contient l'allocateur et les pointeurs,
   — elle ne s'occupe que de l'acquisition des ressources et de leurs libérations.
   — elle ne peut être copiée (ni par construction, ni par assignation) : les valeurs sont copiées, et non les pointeurs, donc la ressource elle-même ne peut pas être copiée.
   — elle peut être déplacée (par construction et par assignation) : transfert de la propriété de la ressource.
10. On propose le code suivant :

```
template<class T, class A = allocator<T>>
struct vector_base {
  A alloc;
  T *elem, *space, *last;
  vector_base(const A& a) : alloc(a), elem(nullptr), space(nullptr), last(nullptr) {}
  vector_base(const A& a, size_t n);
  ~vector_base();
  vector_base(const vector_base&) = delete;
  vector_base& operator=(const vector_base&) = delete;
  vector_base(vector_base&& a);
  vector_base& operator=(vector_base&& a);
};
```

11. On propose le code suivant :

```
vector_base(const A& a, size_t n)
  : alloc(a), elem(alloc.allocate(n)), space(elem+n), last(elem+n) { }
~vector_base() { alloc.deallocate(elem, last - elem); }
```

Cette classe s'occupe uniquement de l'allocation/desallocation des ressources, la construction/destructi des éléments est une attribution de `vector`.

12. On propose le code suivant pour transférer la propriété

```
vector_base(vector_base&& a)
  : alloc(a.alloc), elem(a.elem), space(a.space), last(a.last) {
  a.elem = a.space = a.last = nullptr;
}
vector_base& operator=(vector_base&& a) {
  std::swap(elem, a.elem);
  std::swap(space, a.space);
  std::swap(last, a.last);
  std::swap(alloc, a.alloc);
  return *this;
}
```

13. On propose le code suivant :

```
template<class T, class A = allocator<T> >
class vector {
    vector_base<T,A> vb; // the data is here
public:
    ...
};
```

14. On propose le code suivant (code dans la partie publique de `vector`) :

```
class iterator : public std::iterator<std::random_access_iterator_tag, T> {
private:
  T*  i;
  A*  a;
  iterator(T *Vi, A *Va) : i(Vi), a(Va) {}
    // privé : utilisé plus loin
  void construct(const T& v) { a->construct(i, v); }
  void construct(T&& v) { a->construct(i, std::move(v)); }
  void destroy() { a->destroy(i); }
public:
  iterator() : i(nullptr), a(nullptr) {}
  iterator(const iterator &it) : i(it.i), a(it.a) {}
  ~iterator() {}
  iterator& operator=(const iterator&it) { i = it.i; a = it.a; return *this;
}
  iterator& operator++() { ++i;  return *this;  }
  const T& operator*() const { return *i; }
  T& operator*() { return *i; }
  bool operator!=(const iterator &it) const { return (it.i != i); }
  iterator operator+(size_t n) { return iterator(i + n,a); }
  friend class Vector;
};
iterator begin() { return iterator(vb.elem, &vb.alloc); }
iterator end() { return iterator(vb.space, &vb.alloc); }
iterator here(size_t n = 0) { return iterator(vb.elem + n, &vb.alloc); }
const_iterator begin() const { return const_iterator(vb.elem); }
const_iterator end() const { return const_iterator(vb.space); }
const_iterator here(size_t n = 0) const
    { return const_iterator(vb.elem + n); }
```

L'itérateur constant est le suivant :

```
// itérateur constant
class const_iterator
    : public std::iterator<std::random_access_iterator_tag,const T> {
private:
  const T*  i;
  const_iterator(T *Vi) : i(Vi) {}
public:
  const_iterator() : i(nullptr) {}
  const_iterator(const const_iterator &it) : i(it.i) {}
  ~const_iterator() {}
  const_iterator& operator=(const const_iterator&it)
        { i = it.i; return *this; }
  const_iterator& operator++() { ++i;  return *this; }
  bool operator<(const const_iterator &it) const { return (it.i < i); }
    // < utilisé par std::copy vs
  const_iterator operator+(size_t n) { return const_iterator(i + n); }
  long int operator-(const const_iterator &it) { return i - it.i; }
    // - utilisé par std::copy linux
  const T& operator*() const { return *i; }
  bool operator!=(const const_iterator &it) const { return (it.i != i); }
  friend class Vector;
};
const_iterator cbegin() const { return const_iterator(vb.elem); }
const_iterator cend() const { return const_iterator(vb.space); }
const_iterator chere(size_t n = 0) const
    { return const_iterator(vb.elem + n); }
```

15. On propose le code suivant (hypothèse : l'itérateur sur un objet permet l'accès à construct/destroy de l'allocateur) :

```
template <class T, class It>
void uninitialized_fill(It beg, It end, const T& x) {
  It p;
  try { for (p = beg; p != end; ++p) p.construct(x); }
  catch (...) {
    for (It q = beg; q != p; ++q) q.destroy();
    throw;
  }
}
```

Noter que, à l'allocateur prêt, cette fonction fait partie de la bibliothèque standard.

16. On propose le code suivant :

```
Vector(size_t n, const T& val = T{}, const A& a = A())
: vb(a,n) {
  uninitialized_fill(begin(),end(), val);
}
```

17. On propose le code suivant :

```
template <class It1, class It2>
void uninitialized_copy(It1 beg, It1 end, It2 beg2) {
  It2   p2 = beg2;
  try {
    for (It p = beg; p != end; ++p) {
      p2.construct(*p);
      ++p2;
    }
  }
  catch (...) {
    for (It2 q2 = beg2; q2 != p2; ++q2) q2.destroy();
    throw;
  }
}
```

Noter qu'à l'allocateur prêt, cette fonction fait partie de la bibliothèque standard.

18. On propose le code suivant :

```
Vector(const Vector& v) : vb(v.vb.alloc,v.size()) {
  uninitialized_copy(v.begin(), v.end(), begin());
}
```

19. On propose le code suivant :

```
// construction par déplacement
Vector(Vector&& a)
: vb { std::move(a.vb) } {}

// assignation par déplacement
Vector& operator=(Vector&& a) {
    std::swap(vb, a.vb);
    return *this;
}
```

20. elle ne lancent pas d'exception, à savoir, on pourrait les qualifier `noexcept`.

21. On propose le code suivant :

```
void destroy_elements() {
  for (iterator p = begin(); p != end(); ++p) p.destroy();
  vb.space = vb.elem;
}
~vector() { destroy_elements(); }
```

22. Appelle explicitement le destructeur de `T` pour chacun des éléments du tableau. En conséquence, si la destruction d'un élément lève une exception, la destruction du `vector` échoue.

    Rappel : si une exception à lieu lors du `stack unwinding`, le programme se termine immédiatement.

    En conséquence, si un objet de type `vector<T>` est détruit lors d'un stack unwinding, et que le destructeur de `T` lève une exception, alors le programme se termine.

    Conséquences :

    — aucune garantie ne peut être apportée sur `destroy_element`,
    — même chose dans la bibliothèque standard dans toutes les méthodes qui font appel à un destructeur.
    — aucun moyen de se prémunir contre la conséquence de la levée d'une exception dans un destructeur : on doit toujours faire en sorte qu'un destructeur ne lève jamais d'exception.

23. On propose le code suivant :

```
size_t size()     const { return vb.space - vb.elem; }
size_t capacity() const { return vb.last - vb.elem; }
```

24. On propose le code suivant :

```
// version 1 : garantie forte
Vector& operator=(const vector& a) {
    // 1) get memory
  vector_base   b(a.vb.alloc, a.size());
    // 2) copy elements
  uninitialized_copy(a.begin(), a.end(), iterator(b.elem,&b.alloc) );
    // 3) destroy old elements
  destroy_elements();
    // 4) transfer ownership
  std::swap(vb, b);
    // 5) implicitly destroy the old value
    return *this;
} // 6
```

Remarque : `swap` fonctionne pour `vector_base` car le constructeur par déplacement est défini (= T v(move(x)) ; x=move(y) ; y=move(v) ;).

Pourquoi la garantie est-elle forte ?

- si 1/ lance une exception, l'opération est sans effet (voir `vector_base`).
- 2/ ne lance jamais d'exception.
- 3/ si `destroy_elements` lance une exception, le RAII garantit que les ressources de `b` seront libérées. En revanche, une fuite mémoire est possible pour les ressources mémoires stockées à l'extérieur de `T`, mais inévitable puisque le destructeur ne "marche" plus (on ne peut donc pas faire plus).
- 4/ ne lance jamais d'exception : la ressource est affectée ici
- 5/ ne lance jamais d'exception
- 6/ fin de portée, ressource `b` (celle du swap) automatiquement libérée.

Par, maintenant, en réutilisant les fonctions déjà écrites :

```
// version 2 : garantie forte
Vector& operator=(const Vector& a) {
    Vector temp {a}; // copie de la ressource utilisant le constructeur
                     // par copie (garantie forte)
    std::swap(*this,temp);  // swap representations (pas d'exception)
    return *this;  // pas d'exception
}
```

Note : on a vu que cette version pouvait aussi s'écrire :

```
// version 2 (bis) : garantie forte
Vector operator=(Vector a) { // passage par valeur
    std::swap(*this,a);  // swap representations (pas d'exception)
    return *this;
}
```

Remarque : cette version ne tient pas compte des optimisations suivantes :

- si la capacité du vecteur est assez grande pour contenir le vecteur assigné, la mémoire n'a pas besoin d'être réallouée.
- l'assignation d'un élément peut être plus efficace que la destruction d'un élément suivie par la construction d'un élément.
- le cas v=v n'est pas géré.

```
// version 3: garantie de base
Vector& operator=(const Vector& v) {
  if (capacity() <= v.size()) {
    Vector temp{ v };
    std::swap(*this, temp);
    return *this;
  }

  if (this == &v) return *this;

  size_t  sz = size(), vsz = v.size();
  vb.alloc = v.vb.alloc;
  if (vsz <= sz) {
    std::copy(v.begin(), v.here(vsz), begin() ); // STL
    for (iterator p = here(vsz); p != end(); ++p) p.destroy();
  }
  else {
    std::copy(v.begin(), v.here(sz), begin()); // STL
    uninitialized_copy(v.here(sz), v.end(), end());
  }
  vb.space = vb.elem + vsz;
  return *this;
}
```

Le code n'offre qu'une garantie de base car :

- `copy` n'offre pas une garantie forte car l'assignation par copie peut lever une exception.
- dans ce cas, la copie des éléments est partielle (jusqu'à la levée de l'exception). Le vecteur reste dans un état cohérent et les valeurs originales du vecteur ne sont pas conservées : il y a un effet de bord.
- en raison de cet effet de bord, nous n'avons plus qu'une garantie de base. L'optimisation s'est donc effectuée au détriment de la garantie du code.

Il peut donc être envisageable d'avoir deux versions de l'assignation : une version optimisée (garantie de base) et un `safe_assign` (garantie forte).

25. On propose le code suivant :

```
// version 1
template<class T, class A>
    void vector<T,A>::reserve(size_type newalloc)  // flawed first attempt
{
    if (newalloc<=capacity()) return; // never decrease allocation
    vector v(capacity());         // 1) make a vector with the new capacity
    copy(begin(),end(),v.begin())  // 2) copy elements
    swap(*this,v);                // install new value
} // implicitly release old value
```

Analyse de ce code :

- 1) alloue et initialise les éléments avec la valeur par défaut `T()` (cf prototype ci-dessus).
- 2) écrase ces valeurs par défaut avec la copie du tableau.
- donc on parcourt le tableau deux fois : une fois pour l'initialiser, et une fois pour écraser les valeurs.
- mais, il a une garantie forte.

Deuxième essai : en optimisant :

```
// version 2
void reserve(size_t newalloc) {
  if (newalloc <= capacity()) return;
  vector_base   b{ vb.alloc,newalloc };
  uninitialized_move( begin(), end(), iterator(b.elem,&b.alloc));
  b.space = b.elem + size();
  std::swap(vb, b);
} // implicitly release old space
```

mais il faut écrire `uninitialized_move` :

```
template<class It1, class It2>
It2 uninitialized_move(It1 b, It1 e, It2 oo) {
  for (; b != e; ++b, ++oo) {
    oo.construct(std::move(*b));
    b.destroy();
  }
  return b;
}
```

`Analyse du code:`

- si la construction de `b` lance une exception le vecteur est laissé inchangé.
- Rappel : un déplacement ne doit pas lever d'exception (ne devrait pas, mais peut être le cas, si le sens du déplacement est perverti pour la classe `T`; dans tous les cas, essayer absolument de l'éviter car annule des garanties).
- si `uninitialized_move` lance une exception (la destruction peuvent lever une exception), le RAII libère les ressources, mais les éléments déjà déplacés sont perdus (effet de bord : déplacement partiel), mais pas de fuite mémoire (garantie de base).
- donc ce `reserve()` n'apporte qu'une garantie de base.

26. On écrit tout d'abord une fonction permettant de détruire un intervalle d'éléments :

```
template<typename It> void destroy(It b, It e) {
  for (; b != e; ++b) b.destroy();
}
```

On propose ensuite le code suivant :

```
void resize(size_t newsize, const T& val = T()) {
  reserve(newsize);
    // construct new elements: [size():newsize)
  if (size() < newsize) uninitialized_fill(end(), here(newsize), val);
    // destroy sur plus elements: [newsize:size())
  else destroy(here(newsize), end());
  vb.space = vb.last = vb.elem + newsize;
}
```

27. On propose ensuite le code suivant :

```
void push_back(const T& val) {
  if (capacity() == size()) reserve(size() ? 2 * size() : 8);
  vb.alloc.construct(&vb.elem[size()], val);
  ++vb.space;
}
```

`Analyse du code:`

- pour avoir une garantie forte, il faut déjà utiliser la version de `reserve()` offrant une garantie forte.
- si `reserve` lance une exception, le vecteur est inchangé.

— si le constructeur par copie lance une exception, le vecteur est aussi inchangé. Néanmoins, si cela est le cas, un `reserve()` peut avoir eu lieu inutilement.
— la taille n'est incrémentée que si toutes les opérations précédentes ont réussies.
— donc, `push_back` offre une garantie forte.

28. quasiment aucun bloc `try-catch` dans le code. Tout (à l'exception de `uninitialized_*`) utilisent le RAII afin de gérer la desallocation automatique lors du stack unwinding, et en faisant attention à l'ordre dans lequel on fait les différentes opérations. Ce type d'approche est en général plus élégant et efficace.

    Notons que plus un code est simple, plus sa gestion d'exception l'est aussi.