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

TD N°8

Exercice 1: Vector

On s'intéresse au conteneur `vector`. Soit `A` le type que l'on souhaite stocker dans le conteneur.

1. Quelle est la différence entre la capacité et la taille d'un vecteur ?
2. On écrit : `vector<A> v;` quelle est la taille réservée pour le vecteur `v` ?
3. Que fait `vector<A> v(100);` ?
4. Rappeler les différents moyens de parcourir la totalité d'un conteneur `vector`.
5. Quelle est la différence entre : `v.push_back(x)` et `v.emplace_back(x)` ?
6. On veut initialiser un tableau de `size` éléments. Comparer les 3 implémentations suivantes :

   (a)
   ```
   vector<A>     v;
   for (size_t i = 0; i < size; i++) v.push_back(val);
   ```

   (b)
   ```
   vector<A>     v(size);
   for (size_t i = 0; i < size;i++) v[i]=val;
   ```

   (c)
   ```
   vector<A>     v;
   v.reserve(size);
   for (size_t i = 0; i < size; i++) v.emplace_back(val);
   ```

7. Pourquoi faut-il privilégier les insertions/suppressions en fin plutôt qu'aléatoire ?
8. Trier le tableau.
9. On voudrait les éléments du tableau dans l'ordre, mais en faisant en sorte de conserver l'ordre original du tableau. Comment faire ?

Solution:

1. la capacité est le nombre d'éléments qui peut être stockés dans le vecteur, la taille est le nombre d'éléments qui sont actuellement stockés.
2. capacité = 0, taille = 0.
3. crée un vecteur `v` dont la taille et la capacité sont de 100, et dont les éléments sont initialisés avec le constructeur par défaut, sauf pour les BITS où on a une garantie d'initialisation à 0 (utilise le contructeur `T{}`).
4. pour un type `A` pour lequel l'affectation à 0 produit une initialisation, ce sont les suivants :
   ```
   A   a(2);
   // accès par l'opérateur [] (spécifique au conteneur à accès aléatoire)
   for (size_t i = 0; i < v.size();i++) v[i]=a;
   // avec un itérateur
   // ci-dessous auto=vector<A>::iterator
   for(auto x=v.begin();x!=v.end();++x) *x = a;
   // avec un range-for
   for(A &x : v) x = a;
   ```
5. `push_back` lance le constructeur par copie ou par déplacement pour copier l'élément x. `emplace_back` lance directement le constructeur en place avec les arguments du constructeur.

```
// on suppose que A a un constructeur à 3 arguments A(int,int,int)
// et à un argument A(int)
vector<A> v;
A  a(1,2,3);
v.push_back(a);          // Cc
v.push_back(A(1,2,3)); // Cxxx + Cm + D
v.push_back(1);          // Cx+Cm+D équivalent de v.push_back(A(1))
// v.push_back(1,2,3); // push back ne prend qu'un argument
v.emplace_back(1,2,3); // Cxxx (construction en place)
v.emplace_back(x);       // Cc (pas d'intérêt)
v.emplace_back(X(1));   // Cx + Cm + D (pas d'intérêt)
```

Noter que dans la construction par copie/déplacement, l'objet (ses champs) est initialisé (copié/déplacé) à partir d'un objet déjà existant. Pour la construction sur la place, le constructeur remplit directement les champs de l'objet final.

6. Les différences sont les suivantes :
   (a) utilise la construction par copie/déplacement + réallocation régulière du tableau dès que la capacité maximale est atteinte.
   (b) préallocation du tableau + initialisation dans le cas des bits, et copie/déplacement par assignation dans un élément déjà existant.
   (c) préallocation du tableau, puis utilise la construction en place élément par élément.

   L'approche 3 est normalement la plus rapide (dès lors qu'il y a des copies couteuses à effectuer), mais la 2 peut être optimisée (pipeling) pour les BITs.

   Exemple :
   - Type=int, 1/ 110% plus lente, 2/ 68% plus rapide.
   - Type=struct int x[2]; , 1/ 90% plus lent, 2/ 15% plus rapide
   - Type=struct int x[4]; , 1/ 90% plus lent, 2/ 5% plus rapide
   - Type=struct int x[8]; , 1/ 86% plus lent, 2/ 15% plus lent
   - Type=struct int x[16]; , 1/ 90% plus lent, 2/ 40% plus lent
   - Type=struct int x[32]; , 1/ 84% plus lent, 2/ 65% plus lent
7. à la fin, ne concerne que la fin du tableau (au pire avec une réallocation). Aléatoire, toutes les données après le point d'insertion/suppression doivent être déplacées.
8. code :
```
sort(v.begin(),v.end());  // (défaut = ordre croissant)
```
9. code :
```
vector<int>   u = { 1,3,4,8,7 };
vector<int*>  v;
v.resize(u.size());
transform(u.begin(), u.end(), v.begin(), [](int &x) {return &x; });
sort(v.begin(), v.end(), [](int *x, int *y) { return *x < *y; });
// affichage du tableau trié
for(int *x : v) cout << *x << ", ";
```

Exercice 2: Itérateur

1. Sur `u` et `v` deux conteneurs. Indiquer comment fusionner dans `u` le contenu des deux vecteurs (= ajouter à la fin de `u` le contenu de la liste `v`).
2. Que faire maintenant si je veux déplacer les éléments du vecteur `v` dans la liste `u` ?
3. Est-il possible d'utiliser l'algorithme `copy` (resp. `move`) de la STL pour pour copier (resp. déplacer) les éléments de `v` dans `u` ?

4. Donner une alternative pour la copie en utilisant un itérateur d'insertion.
5. Donner une alternative pour la copie en utilisant un itérateur de déplacement.
6. Écrire le code permettant d'afficher le contenu du conteneur, il devra être fonctionnel pour tout type de conteneur.
7. Donner une alternative en utilisant un itérateur de flux.
8. Dans le code suivant, à quoi correspondent les itérateurs après le `swap` ?

```
std::vector<A> v1({1,2,3}), v2({4,5,6,7});
auto it1b=v1.begin(), it1e=v1.end(), it2b=v2.begin(), it2e=v2.end();
v2.swap(v1);
```

9. Dans le code suivant, à quoi correspond l'itérateur après le `push_back` ?

```
std::vector<A> v1({1,2,3,4});
auto it=v1.begin();
v1.push_back(5);
```

10. Qu'est sensé faire ce code ? Puis, donner l'ensemble des erreurs qu'il produit.

```
std::vector<A> v({1,2,3,4});
for(auto it=v.begin();it!=v.end();++it) v.push_back(*it);
```

Solution:

1. Commencer par redimensionner le vecteur d'accueil à sa taille finale :

```
u.reserve(u.size()+v.size());
```

Deux méthodes :

```
for (auto i = v.begin(); i != v.end(); i++) u.push_back(*i);
```

```
for (auto &x : v) u.push_back(x);
```

2. il faut ajouter `move`, à savoir :

```
for (auto i = v.begin(); i != v.end(); i++) u.push_back(move(*i));
v.clear();
```

Attention, ceci ne change pas le contenu du vecteur `v`, cela utilise le constructeur par déplacement s'il existe, sinon effectue la copie. Le vecteur `v` doit donc ensuite être vidé manuellement.

3. `copy` (resp. `move`) ne permet de ne copier (resp. déplacer) des valeurs que dans des valeurs du conteneur cible déjà construite en utilisant l'assignation par copie (resp. par déplacement).

En conséquence, l'écriture suivante ne doit jamais être utilisée si les éléments de `u` ne doivent pas être écrasée par celle de `v` :

```
// code faux: ne copie pas à la fin car rien n'est construit + corruption mémoire
copy(v.begin(), v.end(), u.end());  // utilise l'assignation par copie
// traces: initialisation des vecteurs
v1[10/10] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
v2[5/5] 10, 11, 12, 13, 14,
// après la copie.
v1[10/10] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
munmap_chunk(): invalid pointer
```

La solution consiste à écrire :

```
v1.reserve(v1.size()+v2.size()); // taille inchangée mais alloue assez de mémoire
auto iv1 = v1.end();             // le pointeur sur la fin est déplacé par resize
v1.resize(v1.size()+v2.size());  // ajout des nouveaux éléments (constr. défaut)
copy(v2.begin(),v2.end(),iv1);   // copie à la fin
```

résultats de ce code :

```
// traces: valeurs de départ
v1[10/10] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
v2[5/5] 10, 11, 12, 13, 14,
// après le reserve
v1[10/15] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
// après le resize
v1[15/15] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0,
// après le copy
v1[15/15] 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
```

4. L'itérateur d'insertion permet d'ajouter de nouveaux éléments, `back_inserter` insère dans le conteneur à la fin de la liste en utilisant `push_back` :

```
u.reserve(u.size()+v.size()); // évite la réallocation pendant la copie.
copy(v.begin(), v.end(), back_inserter(u));
```

5. Utiliser en plus `make_move_iterator` sur les itérateurs de la source, et `back_inserter` sur le conteneur cible :

```
copy(make_move_iterator(v.begin()),
  make_move_iterator(v.end()), back_inserter(u));
v.clear();
```

6. voir code suivant :

```
// méthode 1
for (auto i = v.begin(); i != v.end(); i++) cout << *i << ",";
// méthode 2
for (auto &x : u) cout << x << ",";
```

7. voir code suivant :

```
copy(u.begin(), u.end(), ostream_iterator<A>(cout, "␣,␣"));
```

Attention, le type interne au conteneur doit être passé à l'itérateur (argument `<A>` du template), et la surcharge `<<` doit exister pour le type `A`.

8. Dans le cas d'un `vector`, les itérateurs sont équivalents à des pointeurs. Après permutation, on a donc :

```
// v1.begin() == it2b    // v1.end() == it2e
// v2.begin() == it1b    // v2.end() == it1e
```

9. Toujours réfléchir au sens de l'opération sur le conteneur. Le `push_back` peut provoquer une réallocation mémoire si la capacité est atteinte. Si la réallocation a lieu, la zone de mémoire contenant les données du vecteur est changée, et l'itérateur (=un pointeur) pointe alors sur l'ancienne zone de données et est invalide.

   Il est donc essentiel de comprendre le sens du conteneur et son fonctionnement interne afin d'avoir une idée des effets d'une opération sur un conteneur.

10. Ce code est censé dupliquer son contenu dans lui-même. Il fonctionne s'il est précédé par `v.reserve(2*v.size())`. si `v.reserve() < v.size()`, alors :
    - quand se produit la réallocation, l'itérateur `it` sur le conteneur devient invalide, mais il pointe toujours sur l'ancien tableau.
    - au même moment, la valeur de `v.end()` est modifiée. En conséquence, l'itérateur ne peut jamais devenir égal à `v.end()`, même si l'allocation a lieu à proximité, car pour l'atteindre, il faudrait qu'il se déplace de plus que la nouvelle taille allouée pour le tableau. Or, ceci va provoquer au moins autant de `push_back` ce qui va conduire à une nouvelle réallocation.

— les valeurs lues par l'itérateur finissent par sortir de la zone de mémoire initialement allouée, provoquant ainsi le remplissage du tableau avec des valeurs indéterminées.

En conséquence, ce code produit presque certainement une boucle infinie, ou un segmentation fault (si on atteint une zone invalide mémoire).

Exercice 3: Opérations sur une chaine de caractères

On considère une chaine de caractères contenant un texte en français. Les réponses proposées devront utiliser autant que possible les fonctionnalités de string et/ou les algorithmes de la STL.

1. Remplacer les lettres accentuées par les lettres non accentuées correspondantes.
2. Convertir cette chaine en majuscule.
3. Supprimer les espaces dans la chaine.
4. Écrire un code calculant le nombre d'occurrences de chaque lettre.
5. Afficher les valeurs de l'histogramme.
6. Afficher les valeurs de l'histogramme sous forme de barres de texte.
7. Récupérer la liste de tous les caractères à partir du conteneur contenant les fréquences.
8. Même question à partir de la chaîne de caractères obtenue après la question 3.
9. Vérifier si un caractère x fait partie de cette chaine.

Solution:

1. voir code suivant :

```
string str = "...";
map<char, char> cmap = { {'à','a'},{ 'é','e' },{ 'ê','e' } };
// version 1
// complexité (nb de boucles) = log2(cmap.size()) * str.size()
for (auto &x : str) {
  auto it = cmap.find(x);
  if (it != cmap.end()) x = it->second;
}
// version 2 (moins performant)
// complexité (nb de boucles) = str.size() * cmap.size()
for (auto &x : cmap)
  replace(str.begin(), str.end(), x.first , x.second);
```

2. voir code suivant :

```
// version 1
transform(str.begin(), str.end(), str.begin(), ::toupper);
// version 2
for (auto &x : str) x = toupper(x);
```

3. voir code suivant :

```
// version 1: résultat faux
remove(str.begin(), str.end(),' ');
// version 2: correct
str.erase(remove(str.begin(), next(str.end()),' '),str.end());
```

Correction : remove renvoie un caractère sur le dernier caractère remplacé. Comme on réécrit dans la même chaine, il faut veilleur à conserver le '
0', et supprimer le reste de la chaine .

Exemple de trace avec l'utilisation ou pas de certaines fonctions :

```
// chaine originale, où [x]=taille renvoyée par size, 0=caractère de fin de chaine
// caractères entre | = caractères stockés en mémoire
[21] |AB CD EFGH IJKLM NOP 0|
// remove sans next
[21] |ABCDEFGHIJKLMNOP NOP 0|
// remove avec next
[21] |ABCDEFGHIJKLMNOP0NOP 0|
// remove sans next + erase
[16] |ABCDEFGHIJKLMNOP|
// remove avec next + erase
[17] |ABCDEFGHIJKLMNOP0|
```

4. voir code suivant :

```
map<char, int>   hist;
// version 1
for (char &x : str) ++hist[x];
```

note : ceci fonctionne seulement parce lorsqu'une clef n'existe pas, la valeur mappée de type `A` est initialisée par valeur (i.e. utilise le constructeur `A{}` = constructeur par défaut s'il existe, et initialisation à 0 pour les BITs).

5. voir code suivant :

```
// méthode 1
for(auto &x : hist) cout << x.first << ":" << x.second << "␣";
cout << endl;
```

méthode 2 : définir une surcharge de « pour la paire (pas de surcharge générique)

```
namespace std {
    ostream& operator<<(ostream &os, const pair<char, int> &val) {
      return os << val.first << ":" << val.second << "␣";
    }
}
```

Obligation de le définir dans le namespace std en raison de l'ADL (argument-dependent lookup) qui dit que si un argument est dans un certain namespace, alors la fonction appelée est recherchée dans ce namespace (ajouter l'ADL au cours).

Exemple :

```
namespace ADL {
  struct A {
    int a;
    A(int u) : a(u) {}
  };
  int funADL(A x) { return x.a + 1; }
}
int main() {
  ADL::A  a1(2);
  int x1 = funADL(a1);
  int x2 = funADL(ADL::A(2));
  // int x3 = funADL(1);  // introuvable
}
```

Dans le cas ci-dessus, la surcharge n'est recherchée que dans le namespace std. Conséquence : il est plus prudent de placer les surcharges pour les opérateurs ou les fonctions de la STL dans le namespace std (sinon, risque de ne pas être trouvée).

Le code obtenu :

```
for(auto &x : hist) cout << x;
// ou
copy(hist.begin(), hist.end(), ostream_iterator<pair<const char,int>(cout, ",␣"));
```

6. pour le max, voir code suivant :

```
// version 1
int vMax = 0;
for (auto &x : hist) if (x.second > vMax) vMax = x.second;
cout << vMax << endl;
```

```
// version 2
auto iMax = max_element(hist.begin(), hist.end(),
  [](const pair<char, int> &x, const pair<char, int> &y)
      { return x.second < y.second; });
cout << iMax->first << ":" << iMax->second << endl;
```

Pour l'histogramme :

```
// version 1
for (auto &x : hist) {
  cout << x.first << ": " << string(40 * x.second / vMax, '*') << endl;
}
```

```
// version 2
for (auto &x : hist) {
    cout << x.first << ": ";
    fill_n(ostream_iterator<char>(cout), 40 * x.second / vMax, '*');
    cout << endl;
}
```

7. voir code suivant :

```
// version: type 1
vector<char>  car(hist.size());
// version: type 2
string       car(hist.size(),0);
// commun
transform(hist.begin(), hist.end(), car.begin(),
          mem_fn(&pair<char, int>::first));
```

8. la liste étant triée, on peut utiliser une recherche par dichotomie. Voir code suivant :

```
// rechercher par dichotomie de x dans un ensemble trié: retourne un booléen
if (binary_search(car.begin(),car.end(),x) cout << "found" << endl;
// idem, mais récupère la plus petite valeur inférieure ou égale à x
// la récupération d'un itérateur vers la valeur se fait avec lower_bound
auto it = lower_bound(car.begin(),car.end(),x);
if (it != car.end() && *it==x) cout << "found" << endl;
```

Exercice 4: Liste et Vecteur

On veut dans cette partie manipuler une liste.

1. Remplir une liste avec des valeurs aléatoires uniformes entre 0 et 255.
2. Trier le vecteur par ordre croissant. On proposera deux solutions.
3. Pourquoi la solution de trier la liste peut être quand même plus rapide ?
4. Enlever les éléments dupliqués.
5. Inverser la liste chainée.

Solution:

1. voir code suivant :

```
const size_t nElts = 100;
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(0, 255);
auto RandGen = [&gen, &dis]() { return dis(gen); };
list<int> l(nElts); // l'allocation de la place est ici
generate_n(l.begin(), nElts,RandGen);
```

2. voir code suivant :

```
// méthode interne de la classe liste
l.sort();
```

```
vector<int> v(l.size());
// passage par le tri d'un tableau
copy(l.begin(), l.end(), v.begin());
sort(v.begin(), v.end());
copy(v.begin(), v.end(), l.begin());
// 35% plus rapide
```

3. lorsque les éléments sont triés avec la méthode interne, les éléments ne sont jamais déplacés ni copiés, seuls les pointeurs sont modifiés.

   Ici, le coût de la copie étant faible, les recopies et tri dans le tableau sont plus performants. Mais, dès que la taille de la structure grandit, la tri direct de la structure est plus avantageux.

4. voir code suivant :

```
l.unique();
```

```
// plus rapide
copy(l.begin(), l.end(), v.begin());
unique(v.begin(), v.end());
copy(v.begin(), v.end(), l.begin());
```

   Mêmes remarques qu'à la question précédente.

5. voir code suivant :

```
// méthode interne (pointeur)
l.reverse();
```

```
// à la main
list<int> l2;
copy(l.rbegin(),l.rend(), back_inserter(l2));
swap(l, l2);
```

Exercice 5: Algorithmes

1. En utilisant `transform`, écrire trois façons de multiplier par deux les éléments d'un conteneur.
2. Ecrire un code permettant de mettre les éléments aux positions pairs du conteneur à 0.
3. Reprendre le cours et voir les exemples d'algorithmes en détail.

Solution:

1. voir code suivant :

```
transform(u.begin(), u.end(), u.begin(), [](int x) { return 2 * x; });
```

```
struct { int operator()(int x) { return x * 2; } } mul2;
transform(u.begin(), u.end(), u.begin(), mul2 );
```

```
using namespace std::placeholders;
function<int(int)> fmul2 = bind(multiplies<int>(), _1, 2);
transform(u.begin(), u.end(), u.begin(), mul2);
```

2. voir code suivant :

```
struct {
    int i = 0;
    int operator()(int x) { return ((++i)%2 ? x : 0); }
} zero2;
transform(u.begin(), u.end(), u.begin(), zero2);
```