MOOC Init Prog C++
Corrigés semaine 5

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 15 : échauffement avec les tableaux dynamiques

> Cet exercice correspond à l'exercice n°17 (pages 55 et 218)
> de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

A)

Le code fourni remplit le vecteur `tab` (de taille 10) d'éléments allant de 0 à 9.

En effet, `push_back` ajoute un élément à la fin du tableau. Au moment de l'ajout `tab.size()` vaut la taille du vecteur **avant** l'ajout (puisque l'élément n'est pas encore ajouté).

Vérification : Le code suivant (C++98) :

```
for (int i(0); i < tab.size(); ++i) cout << tab[i] << endl;
```

ou comme ceci en C++11 :

```
for (auto x : tab) cout << x << endl;
```

affiche(nt) :

```
0
1
2
3
4
5
6
7
8
9
```

B)

Ajoute à la fin de `tab2` un vecteur de même taille que le vecteur `tab` et contenant que des

éléments de même valeur : la valeur du premier élément de `tab`.

Dans le cas où `tab2` est un tableau vide (et dans ce cas **seulement**), on pourrait aussi faire :

```
vector<int> tab2(tab.size(), tab[0]);
```

Exercice 16 : produit scalaire

Cet exercice correspond à l'exercice n°18 (pages 55 et 218) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Nous savons depuis la leçon sur les fonctions qu'il ne faut **jamais** faire de copier-coller dans du code, mais faire des fonctions ! C'est par exemple le cas ici pour la saisie des vecteurs (cet aspect est optionnel, mais c'est bien d'y penser !).

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

double scalaire(vector<double> u, vector<double> v);
void saisie(const string& titre, vector<double>& vecteur);

int main()
{
  cout << "Quelle taille pour les vecteurs ? ";
  size_t n;
  cin >> n;

  vector<double> v1(n);
  vector<double> v2(n);

  saisie("premier", v1);
  saisie("second",  v2);

  cout << "Le produit scalaire de v1 par v2 vaut "
       << scalaire(v1, v2) << endl;

  return 0;
}

// ---------------------------------------------------------
double scalaire(vector<double> u, vector<double> v)
{
  double somme(0.0);

  for (size_t i(0); i < u.size(); ++i) {
    somme += u[i] * v[i];
  }

  return somme;
}
```

```cpp
// ----------------------------------------------------------
void saisie(const string& titre, vector<double>& vecteur)
{
  cout << "Saisie du " << titre << " vecteur :" << endl;
  for (size_t i(0); i < vecteur.size(); ++i) {
    cout << " coordonnée " << i+1 << " = ";
    cin >> vecteur[i];
  }
}
```

Exercice 17 : multiplication de matrices

Cet exercice correspond à l'exercice n°20 (pages 57 et 222)
de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Le code fourni ici est en C++11. Pour une version compilant avec l'ancien standard (C++98) voir ci-dessous. Voir aussi les commentaires en fin de corrigé.

```
#include <iostream>
#include <vector>
using namespace std;

vector<vector<double>> lire_matrice();
void  affiche_matrice(const vector<vector<double>>& M);
vector<vector<double>> multiplication(const vector<vector<double>>& M
                                      const vector<vector<double>>& M

// ----------------------------------------------------------------
int main()
{
  vector<vector<double>> M1(lire_matrice()), M2(lire_matrice());

  if (M1[0].size() != M2.size()) {
    cout << "Multiplication de matrices impossible !" << endl;
  } else {
    cout << "Résultat :" << endl;
    affiche_matrice(multiplication(M1, M2));
  }

  return 0;
}

// ----------------------------------------------------------------
vector<vector<double>> lire_matrice()
{
  cout << "Saisie d'une matrice :" << endl;

  unsigned int lignes;
  do {
    cout << "  Nombre de lignes : ";
    cin >> lignes;
  } while (lignes < 1);

  unsigned int colonnes;
  do {
    cout << "  Nombre de colonnes : ";
```

```
    cin >> colonnes;
  } while (colonnes < 1);

  vector<vector<double>> M(lignes, vector<double>(colonnes));

  for (unsigned int i(0); i < lignes; ++i) {
    for (unsigned int j(0); j < colonnes; ++j) {
      cout << "  M[" << i+1 << "," << j+1 << "]=";
      cin >> M[i][j];
    }
  }
  return M;
}

// ---------------------------------------------------------------------
vector<vector<double>> multiplication(const vector<vector<double>>& M
                                      const vector<vector<double>>& M
{
  // crée la Matrice prod à la bonne taille *et* l'initialise
  // avec des zéros :
  vector<vector<double>> prod(M1.size(), vector<double>(M2[0].size())

  for (size_t i(0); i < M1.size(); ++i)
    for (size_t j(0); j < M2[0].size(); ++j)
      for (size_t k(0); k < M2.size(); ++k)
        prod[i][j] += M1[i][k] * M2[k][j];

  return prod;
}

// ---------------------------------------------------------------------
void  affiche_matrice(const vector<vector<double>>& M)
{
  for (auto ligne : M) {
    for (auto element : ligne) {
      cout << element << " ";
    }
    cout << endl;
  }
}
```

Les deux changements à faire au code ci-dessus pour qu'il compile avec les anciennes versions des compilateurs (C++98) sont

- de séparer les deux '>' dans les déclarations de matrices : remplacer partout

  ```
  vector<vector<double>>
  ```

par

`vector<vector<double> >`

(i.e. ajouter un espace)

- dans la dernière fonction (`affiche_matrice`), de remplacer les itérations `for` C++11 par des itérations C :

```
void  affiche_matrice(const vector< vector<double> >& M)
{
  for (unsigned int i(0); i < M.size(); ++i) {
    for (unsigned int j(0); j < M[i].size(); ++j) {
      cout << M[i][j] << " ";
    }
    cout << endl;
  }
}
```

Commentaires

Nous avons avec ce corrigé une belle illustration des progrès introduits dans C++11. En C++98, le code fourni produit en effet plusieurs copies inutiles : les fonctions `lire_matrice` et `multiplication` créent leur propre `Matrice`, laquelle est **recopiée** en sortie (échange de l'information entre le `return` de la fonction et son appel). Il y a donc à chaque fois deux `Matrices` : celle de l'instruction qui fait l'appel et celle de la valeur de retour de la fonction appellé. Cela est inutile et coûteux (surtout si les matrices sont de grande taille).

Une solution pour éviter cette duplication des `Matrices` est, en C++98, d'utiliser les **pointeurs** (c'était d'ailleurs à l'époque l'objet d'un exercice en soit).

Tout ceci **n'est plus nécessaire** en C++11. Cette nouvelle version du langage introduit en effet la notion de «sémantique de déplacement» (*move semantics*), laquelle permet entre autres au compilateur, avec le **MÊME** code d'éviter ces copies (techniquement c'est parce que les `vector` ont maintenant un «constructeur de déplacement» (*move constructor*)). Cela permet aux programmeurs d'écrire des codes plus efficaces, sans avoir rien de particulier à faire !

Vous pouvez même expliciter cette optimisation (mais ce n'est pas nécessaire, le compilateur devrait pouvoir le faire pour vous), en transformant `M1` et `M2` en des «références vers des transitoires» (*r-value refences*) :

```
  const Matrice&& M1(lire_matrice()), M2(lire_matrice());
```

(et c'est tout !!) Cela impose que les seules matrices existant dans tout le programme soient celles crées dans les deux appels de `lire_matrice` et celle résultant de la multiplication. Plus aucune copie inutile supplémentaire...
Vive la *move semantics* !

[Niveau avancé] Arcanes de la *move semantics*

[Niveau avancé] J'en profite pour pousser encore un peu plus loin pour ceux que cela intéresse (mais cela est vraiment avancé, hors des objectifs du cours) : peut on encore faire mieux et ne pas introduire la 3e matrice (le résultat de la multiplication) ?

La réponse serait «oui» avec un autre algorithme pour le calcul de la multiplication permettant le calcul «sur place» (trop compliqué pour être montré ici), mais imaginons que l'on veuille faire une addition au lieu d'une multiplication : on sait qu'on peut alors faire cette addition «sur place», c'est-à-dire calculer le résultat de `A+B` dans `A` lui-même (ou dans `B`), sans necessiter une troisième matrice pour stocker le résultat.

Peut-on coder cela en C++ ? i.e. éviter de créer une 3e `Matrice` ? La réponse est «oui» en C++11.

Il faut pour cela tout d'abord se rendre compte qu'on ne peut le faire que si `A` ou `B` sont des «transitoires» (*r-value reference*). En effet si `A` et `B` sont des «vraies données» (= des variables, *l-values* en termes techniques) alors on est obligé de créer une 3e `Matrice` car on ne peut (souhaite) pas toucher à `A` ni à `B`. Il faut donc *expliciter* le cas où `A` ou `B`, ou les deux, sont des *r-value references*. Cela nous conduit aux quatres prototypes suivants :

```
// deux données stockées, à préserver
Matrice addition(const Matrice& M1  , const Matrice& M2);

// M1 "transitoire", M2 "à préserver"
Matrice&& addition(Matrice&& M1     , const Matrice& M2);

// M2 "transitoire", M1 "à préserver"
Matrice&& addition(const Matrice& M1, Matrice&& M2     );

// deux "transitoires"
Matrice&& addition(Matrice&& M1     , Matrice&& M2     );
```

Cependant, il est clair qu'on ne vas pas écrire quatre fois l'addition !! (**jamais** de copier-coller !) Comment faire ?

Commençons par choisir un cas de référence, par exemple lorsque `M1` est transitoire. Que faut-il faire alors ?

Simplement faire l'addition dans `M1` et retransmettre `M1` plus loin, toujours en tant que transitoire.

Il y a ici encore une subtilité : il faut savoir qu'une *r-value reference* nommée (par exemple en argument de fonction) devient une ***l-value*** ! La fonction pour transformer une *l-value* en *r-value* pour la «passer plus loin» (cela la rend alors invalide dans la suite locale, i.e. dans la suite de sa portée), est la fonction `move`.

Cela nous donne donc :

```
/* ------------------------------------------------------------------
 * Cas ou M1 est un temporaire.
```

```
 * On code ici la "vraie" addition et on utilisera cette version dans
 * les autres cas.
 * Note : on suppose ici que M1 et M2 sont de même taille. Dans un
 * programme plus complet, il faudrait bien sûr s'en assurer !

Matrice&& addition(Matrice&& M1, const Matrice& M2)
{
  for (size_t i(0); i < M1.size(); ++i)
    for (size_t j(0); j < M1[i].size(); ++j)
        M1[i][j] += M2[i][j];
  return move(M1);
}
```

Reste maintenant à utiliser cette addition de référence pour écrire les autres (sans duplication de code).

Le plus simple est sûrement lorsque `B` est transitoire et `A` non : il suffit d'inverser ! Attention cependant à ne pas oublier la conversion automatique d'une *r-value reference* nommée en *l-value reference* !!

Pour le cas où les deux sont des transitoires : il suffit d'en déplacer une...

Le cas le plus compliqué pour réutiliser l'addition déjà écrite sans la réécrire est peut être le cas où `A` et `B` sont des *l-values*. Dans ce cas, il est nécessaire d'introduire une 3^{e} `Matrice` pour ne pas les écraser. L'idée est alors simplement d'introduire cette 3^{e} `Matrice`, d'y recopier l'une des deux autres (e.g. `A`) puis de la «déplacer» dans l'appel de notre addition de référence.

Tout ceci conduit au code suivant :

```
// ----------------------------------------------------------------
Matrice&& addition(const Matrice& M1, Matrice&& M2)
{
  return addition(move(M2), M1);
}

// ----------------------------------------------------------------
Matrice&& addition(Matrice&& M1, Matrice&& M2)
{
  return addition(move(M1), M2);
}

// ----------------------------------------------------------------
Matrice addition(const Matrice& M1, const Matrice& M2)
{
  Matrice resultat(M1);                 // copie M1 dans resultat pour
  return addition(move(resultat), M2); // ...utiliser la version r-va
}
```

Voilà ! Mais peut on faire encore mieux ?

La réponse est «peut être, mais sûrement oui». Si le compilateur que vous utilisez fait de l'élision de copie (*copy elision*, une optimisation du compilateur qui évite les copies superflues), alors on peut faire mieux (il est très certainement probable que votre compilateur fasse de l'élision de copie (g++ le fait), mais ce n'est pas garantie par la norme).

Si l'on peut compter sur l'élision de copie, alors on pourra ne faire qu'*UNE* seule version de l'addition qui prenne en compte trois des 4 cas précédents (il faudra conserver la version `addition(const Matrice& M1, Matrice&& M2))`:

```
Matrice addition(Matrice M1, const Matrice& M2)
{
  for (size_t i(0); i < M1.size(); ++i)
    for (size_t j(0); j < M1[i].size(); ++j)
        M1[i][j] += M2[i][j];
  return M1;
}
```

Notez qu'ici le premier argument est passé par valeur. Si cet argument est un temporaire, et si le compilateur fait de l'élision de copie, cet argument ne sera pas copié mais déplacé (*moved*. Note : ceci est vrai car la classe `vector` possède justement cette sémantique. L'approche décrite ici ne fonctionnerait pas pour des objets n'ayant pas de «*move semantic*» et doit donc être utilisée à bon escient !). De même, la copie de retour ne sera pas faite dans un contexte de temporaire (*return value optimization*). Bref pour résumer, trois des 4 cas précédents seront correctement traités par cette seule fonction.

En conclusion, ce qu'il faut peut être retenir de tout ceci est que l'utilisation de la *move semantics* et des *r-value references* n'est vraiment que pour de l'optimisation et qu'il vaut peut être mieux dans un premier temps ne pas y toucher explicitement, mais se contenter d'utiliser (peut être même sans le savoir) ce qu'elle apporte déjà dans le langage de base (cf commentaires précédents).
