Laboratoire : Éléments du langage C++ et la classe `vector`

1 Objectifs

Le but de ce laboratoire est de pousser à fond des concepts très techniques du C++, afin d'encourager les étudiants à s'efforcer de maîtriser tout ça sur le bout des doigts. Cette maîtrise fournira une base solide pour expérimenter les concepts plus théoriques imminents.

Important : Il est recommandé de faire de petits programmes afin de vérifier vos réponses. N'oubliez pas d'utiliser les opérations d'entrée/sortie du C++.

2 Surcharge de fonctions

On souhaite donner le même nom à trois fonctions. La première additionne deux entiers (type `int`), la deuxième deux réels (type `float`) et la troisième deux tableaux de dix entiers. Donner le prototype de ces fonctions. Montrer que les appels sont correctement réalisés. Que se passe-t-il lorsqu'un appel est fait avec comme arguments deux `short`. Pourquoi ?

3 Utilisation de références

Écrire une fonction qui incrémente la date d'une structure (`struct`) `Date`, tout en respectant les contraintes du calendrier grégorien. La structure `Date` contient 3 attributs : `jour`, `mois`, `annee`. Il y a trois solutions à ce problème (trois manières d'écrire le prototype). Les mettre en oeuvre et expliquer les différences. Indice : de quelles (3) façons peut-on passer un paramètre à une fonction ?

Écrire le prototype d'une fonction qui reçoit une grande structure sachant qu'on ne veut pas perdre de place mémoire et on ne souhaite pas la modifier. Que se passe-t-il lorsqu'on

retourne dans une fonction une variable locale et que la fonction a pour type de retour une référence ? Que faudrait-il faire pour solutionner ce problème ? Donner un exemple.

4 L'attribut `const` et la manipulation de pointeurs

- Définir un pointeur sur un entier, nommé `p`, un pointeur sur un entier constant, nommé `q`, un pointeur constant sur un entier, nommé `r`. Faites des allocations avec l'opérateur `new`. Que constatez-vous ?
- Définir un tableau de dix pointeurs sur des entiers. Faites une allocation de l'ensemble et une désallocation. Vérifiez l'état de la mémoire.
- Définir une référence sur un entier, nommée `s` puis une référence constante vers un entier, nommée `t`. Essayez toutes les affectations entre `p`, `q`, `r`, `s` et `t`, et expliquez les résultats obtenus.

5 Les attributs `const` et `static`

- Corrigez les erreurs de compilation dans le programme se trouvant dans le fichier `exerciceConstStatic.cpp` joint avec cet énoncé, **sans enlever les mots-clé `const`**.
- Corrigez le code pour que le nombre de fois que la fonction `distanceDeLOrigine()` a été appelé soit bel et bien incrémenté à chaque appel.
- Qu'arrive-t-il si on déclare `x` et `y` `static`? Peut-on toujours initialiser `x` et `y` dans le constructeur ? À quoi pourrait servir l'instruction suivante : `Point::x` ou `Point::y` ?

6 Tableaux dynamiques à plusieurs dimensions

Soit les huit versions de tableaux à trois dimensions suivantes (un tableau 2x3x4), donnez les lignes `new` nécessaires à la création de ces tableaux, puis donnez les lignes `delete` nécessaires à la libération de leur mémoire.

La solution des deux premières façons de faire vous est donnée à titre d'exemple.

Truc : Si vous avez de la difficulté à trouver la solution plus les cas les plus tordus, vous pouvez commencer par ajouter des déclarations typedef pour simplifier le tout. Ensuite, essayez de retirer ces typedefs (ce n'est pas si facile non plus). Si vous réussissez les 5 premières versions mais ne réussissez pas les versions 6, 7 et 8 ne vous inquiétez pas, il est très rare de devoir faire ce type de dans la « vraie vie ».

1.
```
int x[2][3][4];
// Rien à faire pour les new ni pour les delete,
// car tout est statique.
```

2.
```
int *x[2][3];
for(int i=0;i<2;i++)
    for(int j=0;j<3;j++)
        x[i][j] = new int[4];
// [...]
for(int i=0;i<2;i++)
    for(int j=0;j<3;j++)
        delete[] x[i][j];
```

3.
```
int **x[2];
```

4.
```
int ***x;
```

5.
```
int (*x)[3][4];
```

6.
```
int (**x)[4];
```

7.
```
int (*x[2])[4];
```

8.
```
int *(*x)[3];
```

7 La classe `vector` de la librairie standard

Nous vous fournissons une implémentation de la classe `vector` telle qu'on la retrouve dans la librairie standard du C++. L'interface n'est pas complète, mais nous avons implémenté suffisamment de méthodes pour vous aider à mieux comprendre le développement orienté objet. La syntaxe employée dans les fichiers sources suit les normes de programmation du cours et les commentaires *Doxygen* sont présents. Nous vous fournissons également les tests unitaires utilisant le framework *Google Test* afin que vous puissiez vous inspirer de ces tests pour vos vérifications plus tard durant la session.

Le but de cet exercice est de vous familiariser avec les éléments nécessaires pour le développement d'une classe. Vous devez lire le code et vous poser un maximum de questions par rapport à tous les éléments que vous y trouverez. En ce sens, nous vous avons préparé une série de questions pour vous aider dans vos réflexions.

Remarque : n'oubliez pas d'ajouter les librairies nécessaires pour le fonctionnement de Google Test (voir labo 1 pour plus de détails).

1. Regardez l'interface de la classe dans le fichier `Vector.h`. L'interface d'une classe qui est bien construite vous permettra en un coup d'oeil de comprendre ce qu'elle fait sans avoir à regarder les détails. Est-ce le cas ici ? Trouvez-vous des améliorations à apporter à l'interface ?

2. Pourquoi le nom des paramètres ne figurent pas dans le prototype des méthodes de l'interface ?
3. Pourquoi l'utilisation d'un *template* est nécessaire ?
4. Que remarquez-vous par rapport à l'utilisation du `const`?
5. Pourquoi `Vector.hpp` est inclut à la fin de la définition de la classe ?
6. Pourquoi `using namespace std` ne doit jamais être déclaré dans un `.h` ?
7. À quoi sert la liste d'initialisation que l'on retrouve dans l'implémentation du constructeur ?
8. Dans `Vector.hpp`, pourquoi doit-on utiliser `Vector<T>::` au début de chaque nom de méthode ?
9. Pourquoi doit-on utiliser `typename` pour les fonctions `begin()` et `end()` ?
10. Trouvez-vous que les tests unitaires sont exhaustifs ? Programmez un test qui n'est pas présent.

Bon travail !