Programmation C++ Avancée

TP 2

1 Création, destruction, copie et élision

Créez une classe `T` possédant un constructeur sans argument, un constructeur de copie, un opérateur d'assignation par copie et un destructeur. Pour chacun de ces quatre membres, ajoutez une ligne de code qui affiche une ligne sur la sortie standard quand il est appelé. On pourra afficher le pointeur `this` pour éliminer toute ambiguïté entre les différents objets :

```
T(T const &t) { std::cout << this << ": constructed from " << &t << '\n'; }
```

Testez votre code sur différents scénarios :

```
void f1(T const &t) {}
void f2(T t) {}
T f3() { return T(); }
T f4() { T t; return t; }
void f5(T &t) { t = T(); }

struct U {
  T v1, v2;
  U(T const &t): v2(t) { v1 = t; }
};

int main() {
  T a;
  f1(a);
  f2(a);
  T b = a;
  T c = f3();
  T d = f4();
  f5(d);
  U e(a);
}
```

Expliquez à quel moment chaque objet est construit, détruit, copié, etc. Soyez particulièrement attentif aux cas où le compilateur a élidé une ou plusieurs copies.

2 Accès à des fichiers

Le but est ici de fournir une encapsulation en C++ des fonctions C d'accès aux fichiers : `fopen`, `fwrite`, `fclose`, disponibles dans `<cstdio>`. Pour une description de ces fonctions, consultez leurs pages de manuel, par exemple en tapant `man fopen` en ligne de commande. Remarque : les fonctions `fopen` et `fwrite` attendent des chaînes C, on pourra utiliser la méthode `std::string::c_str` pour les obtenir à partir de chaînes C++.

Créez une classe `file` possédant un champ de type `FILE*`. Définissez un constructeur prenant une chaîne de caractères, ouvrant le fichier de ce nom et stockant le descripteur obtenu dans le champ.

Définissez une méthode `file::write` prenant une chaîne de caractère en argument et l'écrivant dans le fichier ouvert précédemment.

Testez votre classe sur un exemple du genre suivant. Vérifiez que les fichiers ainsi créés contiennent bien les chaînes attendues.

```
int main() {
  file f("test1.txt");
  f.write("first string for test1\n");
  file g("test2.txt");
  g.write("first string for test2\n");
  f.write("second string for test1\n");
}
```

Définissez un destructeur `file::~file` qui appelle `fclose` pour fermer proprement le fichier ouvert par le constructeur.

Question : quel est le comportement du constructeur de copie créé par défaut par le compilateur pour la classe `file` ? Quel est le problème avec ce comportement ?

Écrivez un programme qui plante violemment à l'exécution à cause d'une copie.

Appliquez l'attribut `=delete` au constructeur de copie de la classe `file` pour le désactiver. Vérifiez que le compilateur rejette maintenant votre programme incorrect.

Constatez que le même problème existe avec l'opérateur d'assignation `file::operator=`. Corrigez-le de la même manière que le constructeur de copie.

3 Sémantique de transfert

La classe `file` précédente ne permet ni construction par copie ni assignation par copie. Il est par contre facile de fournir une sémantique de transfert.

Expliquez en quoi consiste un transfert entre deux objets de la classe `file`. Cela nécessite-t-il de modifier le code du destructeur ?

Ajoutez un constructeur par transfert `file::file(file &&)`. Testez-le sur le code suivant. Remarque : ce code est volontairement incorrect et doit être corrigé.

```
int main() {
  file f("test3.txt");
  f.write("first string for test3\n");
  file g = f;
  g.write("second string for test3\n");
}
```

Question : est-il possible d'écrire un code qui plante en utilisant la classe `file` ?

4 Pointeurs C

L'objectif est de construire des graphes dirigés en s'appuyant sur la classe suivante qui représente un nœud d'un tel graphe.

```
class node;
typedef node* node_ptr;
class node {
  std::vector<node_ptr> children;
};
```

Pourquoi le vecteur `children` contient-il des pointeurs vers des nœuds et non pas les nœuds eux-mêmes ? À quelle catégorie de graphes serait-on limité si c'était le cas ?

Ajoutez à la classe `node` un constructeur qui prend une chaîne de caractère en argument (l'étiquette du nœud). Ajoutez un destructeur qui affiche l'étiquette du nœud détruit.

Ajoutez une méthode `node::add_child(node_ptr)` qui ajoute au vecteur `children` le pointeur passé en argument.

Testez votre classe avec le code suivant :

```
int main() {
  node_ptr a(new node("a"));
  node_ptr b(new node("b"));
  node_ptr c(new node("c"));
  node_ptr d(new node("d"));
  a->add_child(b);
  a->add_child(c);
  d->add_child(b);
  return 0;
}
```

Constatez qu'aucun destructeur n'est appelé. Pourquoi la mémoire occupée par les nœuds n'est-elle pas libérée ?

Ajoutez la ligne suivante avant la commande `return` :

```
delete a;
```

Constatez que le destructeur de `a` est appelé mais pas ceux de ses enfants `b` et `c`. Pourquoi y a-t-il toujours une fuite mémoire ?

Pour boucher cette fuite, il serait possible de modifier le destructeur de `node` pour qu'il fasse `delete` sur chacun des pointeurs contenus dans `children`. Pourquoi est-ce que le programme ainsi obtenu plantera ?

5 Pointeurs « intelligents » C++

Reprenez le programme précédent et remplacez la définition de `node_ptr` par la suivante :

```
typedef std::shared_ptr<node> node_ptr;
```

Constatez que les quatre objets sont maintenant détruits et qu'il n'y a donc plus de fuite mémoire. Expliquez l'ordre dans lequel les destructeurs sont appelés.

Ajoutez un cycle au graphe précédent avec la ligne suivante :

```
b->add_child(a);
```

Expliquez pourquoi une fuite mémoire refait son apparition.

Rappel : Tous les `shared_ptr` pointant vers un même objet partagent un compteur qui indique combien ils sont à pointer vers cet objet. Chaque copie d'un `shared_ptr` incrémente ce compteur ; chaque destruction le décrémente. Quand ce compteur atteint zéro, l'objet pointé est détruit puisqu'il n'y a plus aucun `shared_ptr` qui pointe vers lui.

6 Pointeurs faibles

On souhaite maintenant enrichir la classe `node` pour qu'un nœud sache non seulement vers qui il pointe mais aussi qui pointe vers lui dans le graphe. Ajoutez pour cela un champ `node::parents` de type `std::vector<node_ptr>`.

La méthode `add_child` doit maintenant être modifiée pour aussi ajouter son pointeur `this` au champ `parents` de son argument. Pourquoi est-ce une très mauvaise idée de convertir `this` en un `node_ptr` ?

Utilisez la méthode `shared_from_this` de la classe `std::enable_shared_from_this` pour implémenter correctement `add_child`.

Expliquez pourquoi le champ `parents` introduit nécessairement une fuite mémoire et cela même pour un graphe acyclique.

Modifiez la déclaration du champ `parents` pour qu'il soit de type `std::vector<std::weak_ptr<node>>`. Vérifiez que la fuite mémoire a été bouchée.

Rappel : Le type `weak_ptr` partage le même compteur que `shared_ptr` mais ne modifie pas sa valeur. En particulier, le compteur peut atteindre zéro même en présence d'un `weak_ptr`.

Ajoutez une méthode `node::get_parents` ayant la signature suivante :

```
std::vector<node_ptr> get_parents() const;
```

Quel intérêt présente le type de retour `std::vector<node_ptr>` par rapport au type `std::vector<std::weak_ptr<node>>` ?

Testez votre programme en affichant le nombre d'éléments renvoyés par `b->get_parents()`.

Forcez la libération de certains nœuds en exécutant `a.reset()` juste avant que `b->get_parents()` soit appelé. Que se passe-t-il ?

Corrigez la méthode `get_parents` pour qu'elle ne renvoie que les parents encore vivants.