Programmation C++ Avancée

Joel Falcou Guillaume Melquiond

10 décembre 2019

1 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 ?

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 `~node` pour qu'il fasse `delete` sur chacun des pointeurs contenus dans `children`. Pourquoi est-ce que le programme ainsi obtenu plantera ?

2 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. Dans quel ordre les destructeurs sont-ils appelés ?

Ajoutez un cycle au graphe précédent avec la ligne suivante :

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

Pourquoi y a-t-il à nouveau une fuite mémoire ?

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, le destructeur de l'objet pointé est appelé et sa mémoire désallouée.

3 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 `node::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 implanter correctement `node::add_child`.

Pourquoi le champ `node::parents` introduit-il nécessairement une fuite mémoire et cela même pour un graphe acyclique ?

Modifiez la déclaration de `node::parents` pour qu'il soit de type `std::vector<std::weak_ptr<node>>` et adaptez le programme. 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> node::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.