IFT1169 - Démonstration #7 - Gestion de la mémoire -

Document rédigé à partir du rapport remis par François Monet et réalisé dans le cadre du travail #03 de la session hiver 2006

L'exercice nécessite la création de deux versions du programme. Pour les différencier lors de la compilation, une variable supplémentaire est utilisée « -DFuite ».

Le drapeau de compilation « -DFuite » sert à activer le détecteur de fuite de mémoire dans le programme. Sa présence est donc essentielle dans le cadre de cet exercice.

Lorsque le drapeau « –Fuite » n'est pas levé, le compilateur ajuste le programme de deux manières :

- Il remplace le mot « New » situé dans la méthode « main » par l'opérateur classique « new ».
- Il définit la fonction « fuite_verif() » comme étant un bloc de code vide.

Lorsque le drapeau est levé à l'intérieur du code C++, le compilateur effectue deux grandes tâches :

- Il remplace le mot «New» par une surdéfinition de l'opérateur «new». Cette surdéfinition permettra au détecteur de fuites de mémoire à procéder à une analyse des adresses des pointeurs.

- Il inclut la surdéfinition des opérateurs globaux de « new », « new[] », « delete » et « delete[] » ainsi qu'une version opérationnelle de la fonction « fuite_verif() ». Les raisons expliquant ses inclusions seront présentées dans la section «Le détecteur de fuites de mémoires».

Au niveau du programme principal

Essentiellement, le programme principal ne fait qu'instancier des variables pour les supprimer par la suite : le pointeur « p1 » pointant vers un objet de classe « A » et le pointeur « p2 » pointant vers un tableau d'objets de la classe « B ».
Ces classes ne font qu'afficher leur adresse dans la mémoire au moment de leur construction et de leur destruction. La classe « B » comprend un pointeur vers un nombre entier. Ainsi, son affichage inclus aussi l'adresse de ce pointeur supplémentaire.

Il existe deux sources de fuites de mémoire dans le programme :

- La première est située dans le destructeur de la classe « B » ; le pointeur défini dans la classe n'est jamais détruit.

- La deuxième se situe dans la fonction principale « main() » ; l'objet de classe A est créé mais jamais supprimé.

Le détecteur de fuites de mémoire

Le détecteur de fuites est, conceptuellement, un registre des pointeurs. Ce registre se remplit et grossit lorsque les constructeurs des classes sont appelés. C'est la surdéfinition des opérateurs « new » et « new[] » qui contient le code provoquant le remplissage du registre. Lorsque les objets dynamiques sont supprimés, leur passage est supprimé du registre. Encore, ce sont les surdéfinitions des opérateurs globaux de « delete » et « delete[] » qui initient la mise à jour du registre.

Avant que le programme ne soit complété, une dernière instruction est exécuté par la fonction « atexit(fuite_verif); » : le contenu du registre est affiché et la description des pointeurs « flottants » dans la mémoire est signalée. Expliqué autrement, si le registre n'est pas vide à la fin du programme, c'est qu'un ou plusieurs pointeurs ne furent pas supprimés de la mémoire.

Fonctionnement sans détection de fuites

Lorsque le drapeau «-DFuite» est absent des arguments de compilation, le programme fonctionne sans effectuer la détection des fuites.

Ainsi, le programme procède normalement à créer les différents objets : « p1 », « p2 » et les deux objets de classe « B » dans son tableau. Ensuite, il supprime les objets du tableau, qui, malheureusement, contiennent chacun un pointeur qui n'est pas supprimé.

Lorsqu'il est temps pour supprimer le tableau, l'opérateur « delete[] » est appelé (« void operator delete[](void *pfs) throw() »). Celui-ci appelle l'opérateur « delete » du registre des pointeurs avec la commande « operator delete(pfs); ». Il ne trouvera pas la copie du pointeur « p2 » dans le registre. C'est normale, l'opérateur « new » n'a pas été surchargé et cela n'a pas provoqué les inscriptions dans le registres. Le programme conclut que des pointeurs n'ont pas été supprimés et génère un affichage en sortie et quitte brutalement le programme (exit(1)).

Fonctionnement avec détection de fuites

Avec le drapeau « -DFuite » inclut lors de la compilation, le programme créera une trace de chaque pointeur créé.
Pour ce faire, les opérateurs « new » et « new[] » sont surchargés en prenant comme argument le pointeur créé, le nom du fichier et le numéro de ligne d'où sa création est codée.
C'est l'opérateur « new » qui a la tâche d'insérer dans le registre la nouvelle trace du pointeur créé.
Ainsi, à la fin du programme, lorsque les pointeurs sont détruits, l'opérateur « delete » sera en mesure de trouver dans le registre les pointeurs et les supprimer.

C'est lorsque le programme a terminé la destruction des pointeurs que la fonction «atexit()» devient utile.
Cette fonction terminale appelle la fonction d'affichage « fuite_verif() ». Cette dernière affiche le contenu du registre. Si un pointeur n'a pas été supprimé (et c'est le cas présent), la position du pointeur dans la mémoire de l'ordinateur et dans le code du programmeur seront envoyées à la sortie d'erreur.

Aspects utilitaires du programme

En pratique, la gestion des pointeurs est critique lorsqu'on programme avec un langage qui n'intègre pas la gestion de la mémoire comme C et C++. Une fuite de mémoire peut nuire à la performance du système informatique en réduisant la taille disponible de la mémoire et la vitesse d'exécution des applications. Comme le présent exercice le démontre, le programme peut aussi facilement arrêter brusquement.

Le travail de développement d'un logiciel peut induire la création des fuites. Étant donné qu'un logiciel commercial est généralement développé par multiples programmeurs ayant parfois des styles de programmation divergents, la colligation du code source peut être imparfaite, voir segmentée. Ainsi, la création d'un registre des pointeurs serait un outil intéressant à développer pour mesurer le niveau d'intégration des codes informatiques.

Le déverminage (« debugging ») est une activité couteuse en temps et en argent pour une entreprise en conception de logiciels. Additionné au coût de perdre des clients à cause d'un produit défectueux, il est clair que la problématique de la gestion de la mémoire doit être omniprésente dans le travail du programmeur.