Allocation dynamique
Corrigé

Objectifs

— Comprendre l'allocation dynamique de mémoire.

Rappel : Comme pour tous les TP, il faut commencer par faire un « `git pull` » depuis votre dossier « pim/tp » pour récupérer les fichiers fournis pour cette séance.

Attention : La partie concernant l'utilisation des outils valgrind et valkyrie est à faire sur les machines de l'N7 sauf à installer ces outils sur vos propres machines.
Remarque : Le paquet valkyrie n'est plus disponible sur Ubuntu 20.04 car il utilise encore la bibiliothèque QT4 et non QT5. Nous utiliserons donc valgrind.

1 Comprendre l'allocation dynamique de mémoire

Dans ces exercices, nous nous appuyons sur le programme `exemples_memoires_dynamique.adb` pour illustrer différents aspects liés à l'allocation dynamique de mémoire.

Exercice 1 : Libération de la mémoire
Le compilateur GNAT utilisé ne fournit pas de ramasse-miettes [1]. C'est donc au programmeur de libérer explicitement la mémoire qu'il a alloué dynamiquement. Il doit la libérer dès que cette mémoire n'est plus utilisée. S'il le fait trop tard, la mémoire n'est pas disponible pour d'autres demandes d'allocation. S'il le fait trop tôt, c'est une erreur car on ne doit pas continuer à utiliser une zone mémoire libérée ; le programme est faux et son comportement indéterminé.

Pour libérer la mémoire allouée dynamiquement, Ada fournit `Ada.Unchecked_Deallocation`, procédure générique avec deux paramètres de généricité : le type de l'objet dont la mémoire doit être libérée (*Object*) et le nom du type pointeur qui permet d'y accéder (*Name*). Il faut donc commencer par l'instancier en précisant la valeur de ces deux paramètres.

1. On parle aussi de *ganneur de cellules*. Le terme anglais est *garbage collector*, *GC*.

```
1  type T_Pointeur_Integer is access Integer;
2
3  procedure Free
4          is new Ada.Unchecked_Deallocation
5              (Object => Integer, Name => T_Pointeur_Integer);
6          -- ou is new Ada.Unchecked_Deallocation (Integer, T_Pointeur_Integer);
```

La procédure `Illustrer_Memoire_Dynamique` montre la gestion normale de la mémoire dynamique en absence de ramasse-miettes. On alloue dynamiquement de la mémoire quand on en a besoin, on l'utilise, et dès qu'on n'en a plus besoin on la libère [2]. La procédure de libération met le pointeur à `null` pour éviter que le programmeur se serve du pointeur après sa libération.

Dans la procédure `Illustrer_Memoire_Dynamique_Sans_Free`, la mémoire allouée dynamiquement n'est pas libérée. Elle est définitivement perdue. On parle de *fuite de mémoire*.

Compiler (en utilisation les options `-gnatwa -gnata -g`) et l'exécuter. Tout semble correct.

Exercice 2 : Détecter les fuites de mémoires

Pour identifier les fuites de mémoires (et plus généralement les mauvaises utilisations de la mémoire dynamique) on peut utiliser des outils tels que *valgrind* et son interface graphique *valkyrie*.

1. Exécuter le programme avec valgrind :

```
valgrind ./exemples_memoire_dynamique
```

2. Lire et comprendre les messages affichés valgrind (et en particulier les lignes du code source [3] concernées). En lisant le rapport affiché par valgrind, on apprend qu'il faut utiliser l'option `--leak-check=full` pour voir les détails concernant les fuites de mémoire. Il faut donc utiliser cette option.

3. Corriger l'erreur. Compiler et exécuter avec valgrind pour vérifier qu'elle est bien corrigée.

Solution : valgrind nous indique qu'un bloc de 4 octets n'a pas été libéré. Ce bloc a été alloué à la ligne 31. Il manque l'appel à free dans ce sous-programme.

Notons que dans le cas présent le recours à l'allocation dynamique n'était pas judicieux puisqu'on alloue et on libère la mémoire dans le même bloc : une variable locale de type Integer aurait été suffisante !

Exercice 3 : Danger lié aux pointeurs

Les pointeurs ajoutent une difficulté par rapport à la mémoire automatique [4] : il faut être sûr que l'objet pointé par un pointeur non nul existe encore quand on utilise ce pointeur. Voyons le sur un petit exemple.

1. Lire la procédure `Illustrer_Memoire_Dynamique_Erreur` et répondre aux questions posées.

2. En général, on l'alloue dans un premier sous-programme, on l'utilise dans d'autres sous-programmes et on la libère dans encore un autre sous-programme. Allouer et libérer de la mémoire dynamique dans le même sous-programme n'a pas d'intérêt : on pourrait utiliser une variable locale dont la mémoire est gérée automatiquement.

3. Le programme doit avoir été compilé avec l'option `-g`.

4. On appelle mémoire automatique la mémoire qui est allouée et libérée automatiquement, sans intervention du programmeur. C'est par exemple le cas des variables locales d'un sous-programme. Leur mémoire est allouée automatiquement au début de l'appel du sous-programme et libérée, toujours automatiquement, à la fin de l'exécution de cet appel.

2. Décommenter l'appel à cette procédure dans le programme principal. Compiler et exécuter le programme pour vérifier les résultats.
Solution : Le résultat de l'exécution avec GNATMAKE 10.4.0 sur Ubuntu 22.04 donne :

```
Ptr1.all =  5
Ptr1.all =  5
Ptr2.all =  1536825449
```

Sur les machines de l'N7 on obtient :

```
Ptr1.all =  5
Ptr1.all =  5
Ptr2.all =  0
```

On remarque que dans les deux cas, la valeur de Ptr2.all n'est pas 5. Ce n'est pas anormal car la mémoire a été libérée via Ptr1 et il n'y a plus aucune garantie que le contenu précédent soit conserver. Notons que ce pourrait être le cas et dans bien des cas c'est effectivement le cas. Ici nous avons plutôt de la chance : l'exécution du programme et la lecture attentive des résultats montrent qu'il y a un problème. Le programme est effectivement faux : il faut pas utiliser la mémoire libérée.
3. Exécuter avec valgrind et comprendre les erreurs signalées.
Solution : valgrind signale explicitement l'instruction qui a accédé à un bloc de mémoire dynamique libéré. L'erreur de programmation est donc signalée explicitement. Conclusion, il faut toujours exécuter son programme avec valgrind quand on utilise la mémoire dynamique pour détecter des erreurs de mauvaise utilisation de cette mémoire dynamique.
4. Ne pas corriger ce sous-programme et mettre en commentaire son appel dans le programme principal.

Exercice 4 : Passage de paramètre in et pointeurs

Lire la procédure `Illustrer_Pointeur_In`. Indiquer ce que le programme affichera. Décommenter l'appel à cette procédure dans le programme principal et l'exécuter pour vérifier les résultats.
Solution : Seul le pointeur est en « in ». On ne peut donc pas dans le sous-programme l'associer à une autre zone de mémoire. Cependant, rien n'empêche depuis la fonction de modifier le contenu de la zone mémoire référencée par ce pointeur. Nous avons donc une fonction qui a un effet de bord ! Ceci est à éviter !

Exercice 5 : Structures chaînées et variables locales

Les fichiers `piles.ads` et `piles.adb` proposent une implantation avec des structures chaînées d'une pile. Par rapport à la version tableau, nous avons supprimé la fonction `Est_Pleine` et les préconditions qui l'utilisaient. Ces opérations peuvent échouer s'il n'y plus de mémoire. L'exception `Storage_Error` le signalera.

On a également défini une procédure `Detruire` qui libère la mémoire dynamique utilisée par la pile. Elle doit être utilisée dès que l'on n'aura plus besoin d'une pile.
1. Considérons la procédure `Illustrer_Pile_Locale`. La variable P est une variable locale du sous-programme, la mémoire qu'elle occupe sera automatiquement libérée à la fin du sous-programme. Dans ce cas, faut-il utiliser `Detruire` ou peut-on s'en passer ?

Solution : Il faut utliser `Detruire` car la variable locale qui est libérer automatique ne contient que l'adresse de la première cellule de la pile (ou `null`). Ce n'est donc que la zone occupée par cette adresse qui sera libérée. Il faut au préalable libérer la mémoire dynamique utilisée par les cellules de la pile et c'est ce que fait `Detruire`.

Une exécution avec valgrind signalera ces fuites de mémoire.

2. Décommenter l'appel à `Illustrer_Pile_Locale` dans le programme principal puis compiler et exécuter le programme avec valgrind.

Solution : Les fuites de mémoires sont bien signalées par valgrind.

3. Corriger le programme et le vérifier grâce à valgrind

Solution : Ilf aut ajouter un appel à `Detruire` à la fin du sous-programme (et plus précisément dès que la pile ne sera plus utilisée).

Exercice 6 : Erreurs difficiles à trouver liées à la mauvaise utilisation des pointeurs

La mauvaise utilisation de la mémoire dynamique peut conduire à des erreurs difficiles à trouver. Essayons [5] de le voir.

1. Décommenter l'appel à `Illustrer_Memoire_Dynamique_Erreur` dans le programme principal.

2. Exécuter le programme. Que se passe-t-il ?

Solution : Dans mon cas, il y a une erreur sur le deuxième empilé de la pile. Ceci n'est pas forcément reproductible car le programme est faux : accès et modification d'une mémoire allouée dynamiquement **après** l'avoir libérée !

3. Exécuter avec valgrind et corriger les erreurs signalées (dans l'ordre !).

Solution : Deux corrections possibles :

1. On supprime toutes les instructions qui utilise `Ptr2.all` après l'appel à `Free (Ptr1)` car la mémoire référencée par `Ptr2` a alors été libérée.
2. On alloue de la mémoire pour `Ptr2` pour pouvoir à nouveau utiliser `Ptr2.all`.

4. Indiquer les leçons à tirer de cet exemple.

Solution : Il est important de tester un programme au fur et à mesure pour identifier et localiser plus facilement les erreurs.

Il est important de s'appuyer sur des outils tels que valgrind et valkyrie pour vérifier la bonne utilisation de la mémoire dynamique.

Exercice 7 : Affectation entre structures chaînées

Considérons le programme `illustrer_affectation_pile.adb`.

1. Lire le programme et répondre aux questions qu'il contient.

Solution : Faire `P2 := P1;` signifie que les deux pointeurs vont référencer la même cellule, celle de l'élément en sommet de la pile `'B'`. Ceci n'est pas à faire car quand on va dépiler `P2`, on va libérer la mémoire occupée par la cellule contenant `'B'`. `P2` référencera bien la cellule contenant `'A'` mais `P1` contiendra toujours l'adresse de la cellule que l'on vient de libérer. Il ne faudra plus utiliser `P1` sinon le comportement du programme n'est pas connu. L'utilisation de valgrind permettra de détecter ce type de problème.

5. « Essayons » car suivant le compilateur, l'architecture, etc. l'erreur ne se manifestera pas de la même façon. En effet, le programme étant faux, son comportement n'est pas spécifié par la norme Ada.

2. Compiler puis exécuter le programme, d'abord sans valgrind/valkyrie puis avec, pour confirmer les réponses aux questions.

Solution : L'affichage de P1 peut être « `[A,  >` », « `[ ,  >` » ou encore d'autres choses... Le programme est faux, on ne peut pas prédire ce qui va se passer !

3. Modifier la déclaration du type `T_Pile` dans la partie publique de la spécification du module *Piles* pour remplacer **`private`** par **`limited private`**.

Compiler le programme et en déduire la signification et l'intérêt de **`limited private`**.

Solution : Un type déclaré **`private`** propose l'affectation (`:=`) et le test d'égalité (`=` et `/=`). Un type **`limited private`** (que l'on pourrait traduire par « très privé » en algorithme) ne les fournit pas.

Les types s'appuyant sur des structures chaînées devraient donc toujours être déclarés très privés (**`limited private`**).