Allocation dynamique

Objectifs

— Comprendre l'allocation dynamique de mémoire.

Rappel : Comme pour tous les TP, il faut commencer par faire un « svn update » 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  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

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

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).

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

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. Décommenter l'appel à cette procédure dans le programme principal. Compiler et exécuter le programme pour vérifier les résultats.

3. Exécuter avec valgrind et comprendre les erreurs signalées.

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.

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.

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.

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 ?
2. Décommenter l'appel à `Illustrer_Pile_Locale` dans le programme principal puis compiler et exécuter le programme avec valgrind.
3. Corriger le programme et le vérifier grâce à valgrind.

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 ?
3. Exécuter avec valgrind et corriger les erreurs signalées (dans l'ordre !).
4. Indiquer les leçons à tirer de cet exemple.

Exercice 7 : Affectation entre structures chaînées
Considérons le programme `illustrer_affectation_pile.adb`.
1. Lire le programme et répondre aux questions qu'il contient.
2. Compiler puis exécuter le programme, d'abord sans valgrind/valkyrie puis avec, pour confirmer les réponses aux questions.
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`**.

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.