Cours Langage C/C++
Mémoire et allocation dynamique

Thierry Vaira

BTS IRIS Avignon

tvaira@free.fr ☮ v0.1

La pile et le tas

- La mémoire dans un ordinateur est une **succession d'octets (soit 8 bits)**, organisés les uns à la suite des autres et **directement accessibles par une adresse**.
- En C/C++, la mémoire pour stocker des variables est organisée en deux catégories :
  1. la pile (*stack*)
  2. le tas (*heap*)
- Remarque : Dans la plupart des langages de programmation compilés, la pile (*stack*) est l'endroit où sont stockés les paramètres d'appel et les variables locales des fonctions.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

La pile (stack)

- La pile (*stack*) est un **espace mémoire réservé au stockage des variables désallouées automatiquement**.
- Sa taille est limitée mais on peut la régler (appel POSIX `setrlimit`).
- La pile est bâtie sur le modèle **LIFO** (*Last In First Out*) ce qui signifie "Dernier Entré Premier Sorti". Il faut voir cet espace mémoire comme une pile d'assiettes où on a le droit d'empiler/dépiler qu'une seule assiette à la fois. Par contre on a le droit d'empiler des assiettes de taille différente. Lorsque l'on ajoute des assiettes on les empile par le haut, les unes au dessus des autres. Quand on les "dépile" on le fait en commençant aussi par le haut, soit par la dernière posée. Lorsqu'une valeur est dépilée elle est effacée de la mémoire.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Le tas (heap)

- Le tas (*heap*) est l'autre **segment de mémoire utilisé lors de l'allocation dynamique** de mémoire durant l'exécution d'un programme informatique.
- Sa taille est souvent considére comme illimitée mais elle est en réalité limitée.
- Les fonctions `malloc` et `free`, ainsi que les opérateurs du langage C++ `new` et `delete` permettent, respectivement, d'allouer et désallouer la mémoire sur le tas.
- La mémoire allouée dans le tas doit être désallouée explicitement.

Sous Linux, on peut visualiser facilement les valeurs du tas et de la pile :

```
$ ulimit -a
...
data seg size           (kbytes, -d) unlimited
...
stack size              (kbytes, -s) 8192
...
```

La taille de la pile étant limitée (ici à 8Mo), cela peut provoquer des écrasements de variables et surtout des "**Erreur de segmentation**" en cas de dépassement. Il est évidemment recommandé d'allouer dans le tas les "grosses" variables sous peine de surprise !

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

malloc et free

- L'allocation d'une nouvelle zone mémoire se fait dans un endroit particulier de la mémoire appelée le **tas** (*heap*).
- Elle se fait par la fonction `malloc` : `void *malloc(size_t taille) ;`
- L'argument transmis correspond à la **taille en octets** de la zone mémoire désirée.
- La valeur retournée est un **pointeur void *** sur la zone mémoire allouée, ou **NULL** en cas d'échec de l'allocation.
- Si vous devez redimensionner un espace mémoire qui a été alloué dynamiquement, il faudra utiliser la fonction `realloc()`.
- La mémoire allouée doit, à un moment ou un autre, être libérée. Cette libération mémoire se fait par la procédure `free` : `void free(void *pointeur) ;`

Exemple : allocation mémoire

```c
int *p; // pointeur sur un entier
int *T; // pointeur sur un entier

// allocation dynamique d'un entier
p = (int *)malloc(sizeof(int)); // alloue 4 octets (= int) en mémoire
*p = 1; // ecrit 1 dans la zone mémoire allouée

// allocation dynamique d'un tableau de 10 int
T = (int *)malloc(sizeof(int) * 10); // alloue 4 * 10 octets en mémoire
// initialise le tableau avec des 0 (cf. la fonction memset)
for(int i=0;i<10;i++) {
   *(T+i) = 0;   // les 2 écritures sont possibles
   T[i]   = 0;   // identique à la ligne précèdente
}
// ou plus directement
memset(T, 0, sizeof(int)*10); // il faudra alors inclure string.h

free(p);
free(T);
```

Une fois qu'une zone mémoire a été libérée, il ne faut sous aucun prétexte y accéder, de même qu'il ne faut pas tenter de la libérer une seconde fois.

Règles de bonne conduite

Un certain nombre de règles, quand elles sont observées, permettent de se mettre à l'abri de problèmes :

- Toute déclaration de pointeur s'accompagne de son initialisation à `NULL`
- Avant tout appel de `malloc()`, on doit s'assurer que le pointeur à allouer est bien `NULL`
- Après tout appel de `malloc()`, on s'assure qu'aucune erreur ne s'est produite
- Avant de libérer une zone mémoire, on s'assure que son pointeur n'est pas `NULL`
- Dès qu'une zone mémoire vient d'être libérée par `free()`, on réinitialise son pointeur à `NULL`

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

new et delete

- Pour allouer dynamiquement en C++, on utilisera l'opérateur `new`.
- Celui-ci renvoyant une adresse où est crée la variable en question, il nous faudra un pointeur pour la conserver.
- Manipuler ce pointeur, reviendra à manipuler la variable allouée dynamiquement.
- Pour libérer de la mémoire allouée dynamiquement en C++, on utilisera l'opérateur `delete`.

Pour allouer dynamiquement en C++, on utilisera l'opérateur `new`.

Exemple : allocation dynamique

```
#include <iostream>
#include <iostream>
#include <new>

using namespace std;

int main ()
{
   int * p1 = new int; // pointeur sur un entier

   *p1 = 1; // ecrit 1 dans la zone mémoire allouée
   cout << *p1 << endl; // lit et affiche le contenu de la zone mémoire allouée

   delete p1; // libère la zone mémoire allouée

   return 0;
}
```

Exemple : allocation dynamique d'un tableau

```
#include <iostream>
#include <new>

using namespace std;

int main ()
{
   int * p2 = new int[5]; // alloue un tableau de 5 entiers en mémoire

   // initialise le tableau avec des 0 (cf. la fonction memset)
   for(int i=0;i<5;i++)
   {
      *(p2 + i) = 0;  // les 2 écritures sont possibles
      p2[i]     = 0;   // identique à la ligne précèdente
      cout << "p2[" << i << "] = " << p2[i] << endl;
   }

   delete [] p2; // libère la mémoire allouée
   return 0;
}
```

Pour allouer dynamiquement un **objet**, on utilisera l'opérateur `new`.

Exemple : allocation dynamique d'un objet

```
#include <iostream>
#include "point.h"
using namespace std;

int main()
{
   Point *pointC; // je suis pointeur sur un objet de type Point

   pointC = new Point(2,2); // j'alloue dynamiquement un objet de type Point

   pointC->afficher(); // Comme pointC est une adresse, je dois utiliser l'opérateur -> pour accéder aux
         membres de cet objet

   pointC->setY(0); // je modifie la valeur de l'attribut _y de pointB

   (*pointC).afficher(); // cette écriture est possible : je pointe l'objet puis j'appelle sa méthode
         afficher()

   delete pointC; // ne pas oublier de libérer la mémoire allouée pour cet objet

   return 0;
}
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Fuite de mémoire

- L'allocation dynamique dans le tas **ne permet pas la désallocation automatique**.
- Chaque allocation avec "`new`" doit impérativement être libérée (détruite) avec "`delete`" sous peine de créer une **fuite de mémoire**.
- La fuite de mémoire est une zone mémoire qui a été allouée dans le tas par un programme qui a omis de la désallouer avant de se terminer. Cela rend la zone inaccessible à toute application (y compris le système d'exploitation) jusqu'au redémarrage du système. Si ce phénomène se produit trop fréquemment la mémoire se remplit de fuites et le système finit par tomber faute de mémoire.

Ce problème est évité en Java en introduisant le mécanisme de "ramasse-miettes" (*Garbage Collector*).