Programmation C
TP 9

De l'allocation dynamique.

Fonctions de <stdlib.h> :

- `void *malloc(size_t size);`
- `void free(void *ptr);`
- `void *calloc(size_t nmemb, size_t size);`
- `void *realloc(void *ptr, size_t size);`

Sur nos machines récentes en 64 bits, il faut penser que `size_t` n'est rien d'autre que `unsigned long int`. C'est un long entier positif qui représente le nombre d'octets d'une zone mémoire. L'intérêt du type `size_t` est qu'il s'adapte à la machine. C'est globalement toujours un entier positif mais il se pourrait qu'une machine avec une architecture exotique ou à ressources limitées utilise une donnée plus petite (ou plus grande) pour représenter les tailles de zones mémoires. Comme un bon code est un code portable, on utilisera `size_t` au besoin.

Pour tous les exercices, l'utilitaire d'Unix `valgrind` peut vous donner une indication sur la manière dont votre programme manipule la mémoire. en lançant la commande:

```
valgrind ./mon_executable arg_de_l_executable
```

`valgrind` va encapsuler l'exécution et faire un fin bilan des zones mémoires allouées et libérées. A la fin, il donne un tableau vous disant si votre exécutable se comporte bien vis à vis de la mémoire manipulée. Même si la lecture fine de la verbose de `valgrind` est fastidieuse, le message important à voir est le suivant :

```
All heap blocks were freed - no leaks are possible
```

Cela signifie que le programme à tout libéré avant sa sortie d'exécution.

Exercice 1 Premiers pas

L'objectif de premier exercice est d'allouer et de libérer un simple tableau composé d'entiers. La taille de ce tableau ne sera connu qu'à l'exécution et sera donné par l'utilisateur comme premier argument de votre exécutable. Une fois alloué, votre tableau devra être rempli par des entiers en commençant par 1 et en incrémentant progressivement. Une fois rempli, faites un affichage du contenu du tableau. Une exécution possible est ainsi la suivante :

```
nborie@perceval$ ./exo1 12
1 2 3 4 5 6 7 8 9 10 11 12
nborie@perceval$
```

Une fois votre programme fonctionnel, utilisez `valgrind` pour faire un constat de l'utilisation de la mémoire. Vérifiez que vous ne créer pas de fuite mémoire.

```
nborie@perceval$ valgrind ./exo1 12
==20436== Memcheck, a memory error detector
==20436== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==20436== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==20436== Command: ./exo1 12
==20436==
1 2 3 4 5 6 7 8 9 10 11 12
==20436==
==20436== HEAP SUMMARY:
==20436==     in use at exit: 0 bytes in 0 blocks
==20436==   total heap usage: 1 allocs, 1 frees, 48 bytes allocated
==20436==
==20436== All heap blocks were freed -- no leaks are possible
==20436==
==20436== For counts of detected and suppressed errors, rerun with: -v
==20436== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
nborie@perceval$
```

Rappelez vous qu'une bonne pratique de programmation est de faire une tâche, une fonction! Ici, les tâches atomiques sont les suivantes :

- Allocation d'un tableau
- Libération d'un tableau
- Affichage d'un tableau
- Remplissage d'un tableau

Une bonne fonction `main` est une fonction courte qui articule les tâches atomiques en déléguant le boulot aux autres fonctions. Cette démarche a pour objectif la lisibilité et le débogage. Quand les fonctions réalisent des tâches simples et qu'elles ne fonctionnent pas, on débogue alors des petites fonctions.

Exercice 2 Allocation à deux dimensions

L'objectif de cet exercice est d'allouer proprement puis de remplir un tableau à deux dimensions de caractères. Pour rappel, un tableau à deux dimensions est un tableau de tableaux lignes(ou colonnes) et aura ainsi ici pour type `char**`. Les deux tailles seront données non pas à la ligne de commande mais durant l'exécution du programme (un bon `scanf` doit suffire).

Il faudra aussi songer à libérer correctement la mémoire en fin de programme. Le remplissage du tableau devra être fait avec les lettres minuscules de l'alphabet en commençant avec un 'a' en haut à gauche du tableau puis, pour chaque déplacement vers le bas ou vers la droite, on devra aller vers la lettre suivante de l'alphabet. Voici un exemple d'exécution :

```
nborie@perceval$ ./exo2
Donnez deux dimensions entières : 18 36
a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j
```

```
b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k
c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l
d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m
e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n
f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o
g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p
h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q
i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r
j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s
k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t
l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u
m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v
n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w
o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x
p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y
q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z
r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z a

nborie@perceval$
```

Exercice 3 **Allouer et recopier les arguments d'un programme**

Écrire un programme qui alloue un tableau de chaînes de caractères de taille adapté et recopie chacun des arguments donné au programme dans ce tableau. Dans ce tableau de chaînes, chaque chaîne devra avoir la longueur optimale (la plus courte possible). Faites aussi une fonction d'affichage et une fonction de libération, testez vos fonctions dans un `main` adapté.

```
nborie@perceval:$ gcc exo3.c -o test -Wall -ansi
nborie@perceval:$ valgrind ./test quelques arguments 124 45.321
==24770== Memcheck, a memory error detector
==24770== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==24770== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==24770== Command: ./test quelques arguments 124 45.321
==24770==
arg[0] : ./test
arg[1] : quelques
arg[2] : arguments
arg[3] : 124
arg[4] : 45.321
==24770==
==24770== HEAP SUMMARY:
==24770==     in use at exit: 0 bytes in 0 blocks
==24770==   total heap usage: 6 allocs, 6 frees, 77 bytes allocated
==24770==
==24770== All heap blocks were freed -- no leaks are possible
==24770==
==24770== For counts of detected and suppressed errors, rerun with: -v
==24770== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
nborie@perceval:$
```

Exercice 4 **Pour aller plus loin...**

Si vous vous sentez à l'aise avec `Malloc`, essayez d'en percer les secrets. Si la fonction `free` ne produit pas de segfault, c'est que `free` à le pouvoir de connaître la taille des zones mémoires allouées (et libère d'ailleurs juste le bon nombre d'octet sans qu'on lui donne une taille). Pourtant, en C, les tableaux ne connaissent pas leur taille.

Voici une fonction pour afficher des informations contenues autour des zones mémoire allouées :

```
1  void print_info_zone(void* adr){
2    /* Pour lire la zone en paquet de 8 octets */
3    /* et recuperer des tailles de zones */
4    size_t* info_adr = (size_t*)adr;
5    /* Affichage de l'adresse hexadecimale de la zone */
6    printf("Zone a l'adresse : %p\n", adr);
7    /* Affichage d'une taille juste avant la zone */
8    printf("%lu \n", info_adr[-1]);
9    /* Affichage d'une taille deux cases avant la zone */
10   printf("%lu \n", info_adr[-2]);
11 }
```

En utilisant cette fonction et en allouant et libérant plusieurs zones, tentez de comprendre le fonctionnement de `malloc` et `free`.