UNIVERSITÉ DE MONTRÉAL

Travail Pratique n°2

Par
Mehran ASADI - 1047847
Lenny SIEMENI TCHOKOTE - 1055234

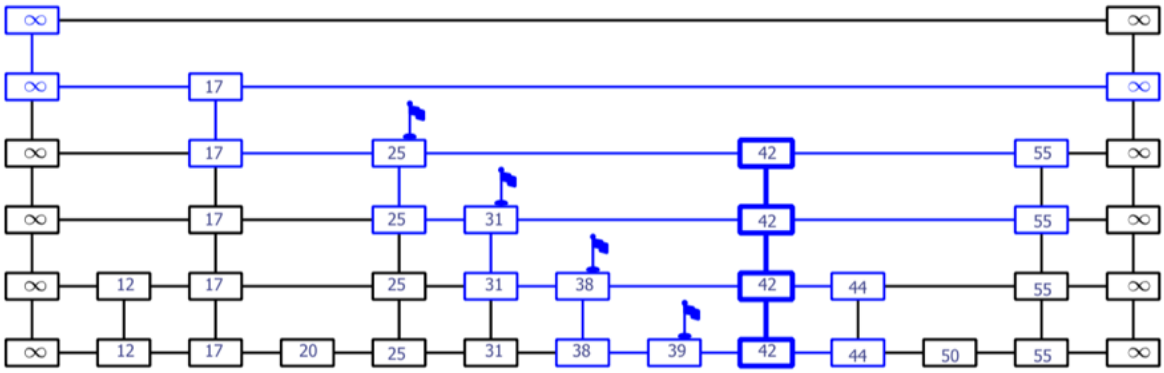
Faculté des arts et des sciences
Département d'Informatique et de Recherche Opérationnelle

Travail présenté à Vincent Archambault-Bouchard
Dans le cadre du cours
IFT2035 – Concept de langages

Juillet 2019

Expérience de développement et compréhension de malloc :

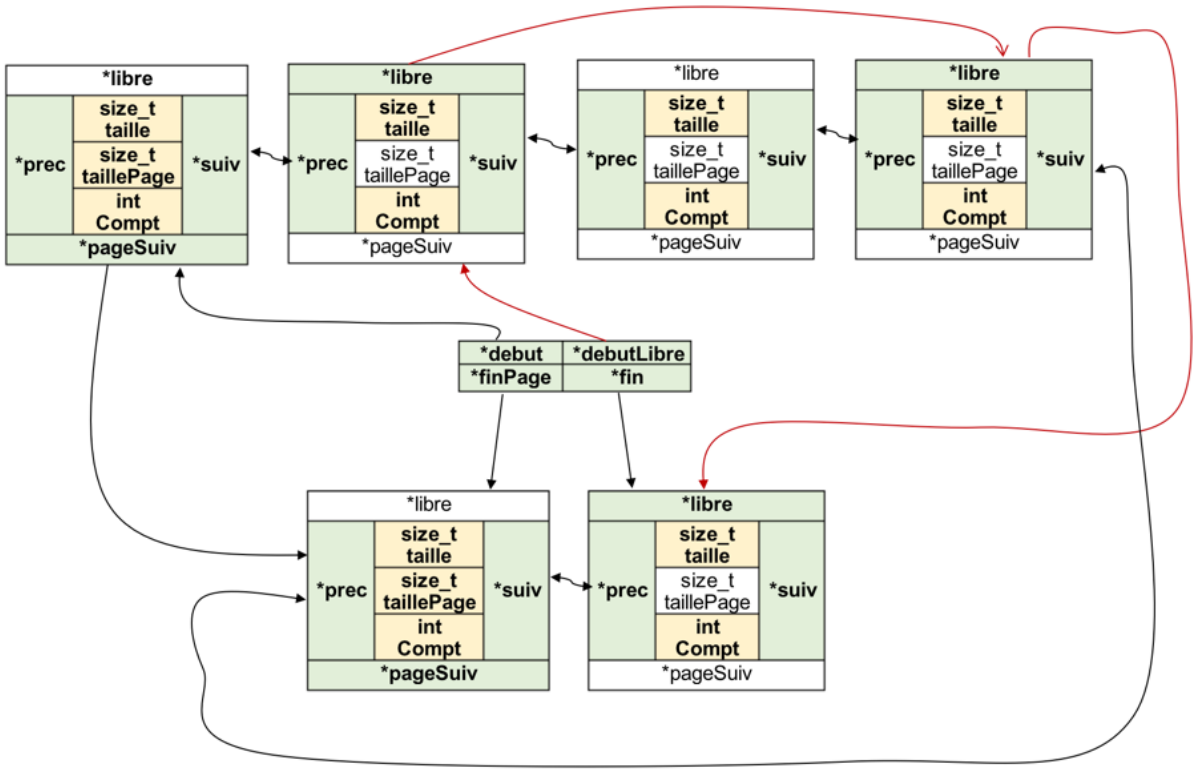
Avant de commencer à implémenter notre algorithme, nous avons effectué quelques recherches académiques sur ce qu'était réellement la fonction « malloc ». Les plus grosses difficultés étaient de savoir comment s'y prendre pour aligner les blocs mémoires dans tous les cas d'utilisations pour maintenir une bonne performance, de savoir quel algorithme de fouille récursif nous allions utiliser pour libérer des blocs mémoires, et si ceux-ci pouvaient facilement s'adapter à notre structure de représentations des données de la mémoire, ainsi que, comme dans les algorithmes vus au cours tels que mark & sweep et « Stop & copy », quelles structures de données on allait utiliser. En effet, on a jugé qu'il était plus simple pour nous de représenter notre structure de la mémoire par une liste doublement chaînée (avec quelques modifications pour se rapprocher d'un map/dictionnaire comme en python), et de jouer avec les différents pointeurs. Cette approche à notre sens nous permet de plus simplement réorganiser les pointeurs plutôt que de manipuler des tableaux, et nous permet de réutiliser des idées d'algorithmes à la manière d'une SkipList que nous avions vus au cours de Structures de données :



1. Schéma Skip List (Extrait des notes du cours IFT2015 – François Major)

L'idée est que nous utilisons la liste doublement chaînée donc, pour représenter la structure même des éléments en mémoire, et une « boussole » ou un « catalogue » qui indique le début de la structure, la fin, la position du premier bloc mémoire libre (offrant un gain de temps important notamment si la mémoire est fragmentée, ainsi que la dernière page mémoire de la structure, puisque notre implémentation supporte donc la pagination.

Le principal inconvénient que l'on peut voir dans notre algorithme, est qu'il est probablement plus gourmand en mémoire que d'autre approches. Les listes chaînées sont des objets et les objets en mémoire nécessitent une plus grande taille pour les représenter. De plus, en général, l'indexation n'est pas supporté par défaut, par conséquent, il faut implémenter un système d'indexation pour augmenter les performances (ainsi qu'un itérateur pour pouvoir parcourir notre liste !), notamment lors des recherches. On ne veut pas qu'à chaque fois qu'on libère un objet en mémoire, on aies à parcourir la liste dans sa totalité, d'où l'inspiration avec la Skip List (évoquée plus haut).



2. La représentation de notre structure de mémoire

Notre approche fut plutôt concluante étant donné que notre malloc passe la majorité des tests de robustesse.

L'algorithme en tant que tel :

Voici donc l'implémentation de notre malloc qui reprends point par point chaque fonction dans notre programme, annoté d'une explication, notre approche ainsi que les forces et faiblesses du code.

```
#define MAXSIZE 16711568
#define ALIGN(size) (((size) + (16-1)) & ~(16-1))
#define CASEALLIGNE ALIGN(sizeof(struct cases))
#define TAILLEPAGE ALIGN(4096) + CASEALLIGNE
#define ALIGNPAGE(size) (((size) + (4096-1)) & ~(4096-1))
struct cases{
    size_t taille, taillePage;
    int compt;
    struct cases *suiv, *prec, *pageSuiv, *libre;
};
```

Structure représentant donc chaque nœud dans la liste doublement chaînée, comme décrite précédemment.

```
struct navigateurs{
    struct cases *debut, *fin, *finPage, *debutLibre;
};
typedef struct navigateurs *Navigateurs;
```

Le navigateur permettant de mapper certaines cases mémoires pertinentes comme le début des cases libres, la fin de la pagination, mapper le début et la fin de la structure.

```
void initNavigateur()
Case pageVideGenerateur(size_t taille);
Case memoireVide(size_t taille);
void trancheCase(Case caseP, size_t taille);
Case trouverCase(void *ptr);
void fusionner(Case actuel);
void detruirepage(Case actuel);
void imprimer();
void imprimervide();
void *mymalloc(size_t taille);
int refinc(void *ptr);
void myfree(void *ptr);
```

void initNavigateur():

Fonction permettant de créer la première page mémoire avec un navigateur
En premier, il alloue la mémoire pour le navigateur.

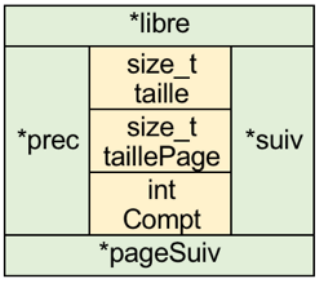
```
void initNavigateur(){
        ///Alloue un peu de mémoire pour le navigateur
        nav = malloc(ALIGN(sizeof(struct navigateurs)));
        if(!nav){
                printf("preobleme d'allocation memoire");
                return;
        }
        ///Creer un nouveau case vide
        Case caseVide = pageVideGenerateur(TAILLEPAGE);
        ///Pointe les éléments du Navigateur vers le nouveau block
        nav->debut = caseVide;          ///Premier case
        nav->fin = caseVide;    ///Dernier case
        nav->debutLibre = caseVide;  ///Premier case libre
        nav->finPage = caseVide;        ///dernier page
}
```

Cette fonction initialise les pointeurs du navigateur, en vue du prochain *mymalloc()*.

Pour répondre à la question « Est-ce que notre algorithme réserve une taille exacte ou bien à un nombre d'octet près ? » -> Oui, de plus, il génère une page mémoire d'une taille de 4096 octets (4 KBytes) . Quant aux cases mémoires à l'intérieur des pages, elles sont alignés sur des blocs de 16 octets.

Case pageVideGenerateur(size_t taille):

Notre générateur de case vide. Rappelons nous que chaque case est représentée comme ceci :



3. Une case mémoire

A l'appel de *mymalloc()*, on va alors créer cette case, lui affecter une taille de page et la taille physique qu'elle représente, demandée par l'utilisateur. Ensuite, ce sera au tour de « *blockInsert()* » d'évaluer cette case et de déterminer où elle soit être insérée dans la liste doublement chaînée, et si celle-ci doit être re-alignée en mémoire si la taille n'est pas un multiple de 4 KiB (nous expliquons ce comportement plus bas dans le rapport).

Case memoireVide(size_t taille):

Cette fonction tente de retourner suite à l'appel de *mymalloc()*, une case mémoire libre d'au moins la taille demandée. On reçoit un pointeur à retirer, on doit donc reculer d'une distance de metadata, pour retrouver le bloc mémoire alloué à cette adresse mémoire de notre malloc en un temps constant.

void trancheCase(Case caseP, size_t taille):

Voici la fonction *trancheCase()* qui va déterminer quels blocs peuvent êtres divisés en vue de les aligner avec le multiple de la taille de notre structure. Par la même occasion, on détermine les références dépendamment de si le bloc est le premier ou le dernier élément de la liste. Cette méthode est appelée uniquement par *mymalloc()* lors de l'allocation d'un bloc mémoire.

Case trouverCase(void *ptr):

Ici, on reçoit un pointeur à retirer. On doit donc reculer d'une distance de metadata, pour retrouver le bloc mémoire alloué à cette adresse mémoire de notre malloc en un temps constant.

void fusionner(Case actuel):

Cette fonction permet de fusionner deux cases libres côte-à-côte dans une même page mémoire. Lorsque toutes les case possibles auront été fusionné, on réarrange les pointeurs des cases-non libres adjacentes dans la page. Bien sûr on n'oublie pas de modifier la valeur de la taille de la nouvelle case ainsi créé.

void detruirepage(Case actuel):
Cette fonction détruit la page libre, si c'est la premiere page de la liste ou la seul page restante de la liste.

imprimer() et imprimervide():

*C*ette fonction est utile simplement pour le programmeur afin de visualiser la structure lors de débogage ou des opérations maintenance, ou simplement pour aider à comprendre ce qu'il se passe dans la mémoire.

void *mymalloc(size_t taille):

Et voici donc notre *mymalloc()*. C'est à ce malloc que l'utilisateur fait appel lorsqu'il veut allouer de la mémoire. Sur l'insertion du bloc dans la liste, si on détermine que le bloc est trop grand, on le divise de manière à ce que sa taille reste un multiple de la taille de la structure représentant notre mémoire.

Au final, ***malloc()*** n'est appelé uniquement lorsque l'on veut créer une nouvelle page mémoire et nous retourne un bloc de 4096 Ko. Tant et aussi longtemps qu'il n'est pas nécessaire d'en créer une autre, c'est notre algorithme qui s'occupe de faire les allocations en interne.

Allocations de grande taille :

Notre algorithme est conçu pour refuser toute allocation de plus de 16 Mo (16711568 bytes), la même limite qui est définie par malloc (IBM - Reserve Storage Block). Même si ceci oblige l'utilisateur à effectuer plusieurs appels à *mymalloc()* pour obtenir son bloc mémoire de 100 Mo (et donc plusieurs appels subséquents à *malloc()*), on préfère gérer nous même ce type d'appel,et on évite d'avoir une erreur renvoyée par le compilateur.

int refinc(void *ptr):

Ici on retrouve le compteur de référence de chaque élément de la liste comme dans Mark & Sweep par exemple. Plus il y a de nœuds qui possèdent de références vers d'autres éléments, plus cette valeur est grande. Lorsque le compteur de référence tombe à 0. Le nœud pourra être marqué comme libre. Cependant, notre algorithme ne gère pas la réorganisation des éléments dans la mémoire efficacement. Alors plus ils y aura des opérations d'allocation et de désallocation mémoire (de petite taille), plus notre mémoire risque d'être fragmenté et les performances de recherches/allocations futures seront moindre, c'est le point faible principal de notre algorithme.

Une des optimisations possibles aurait été de réorganiser les blocs libres vers la dernière page de la structure globale, rendant notamment les opérations d'alignements bcp plus simples et rapides. En effet, considérons le cas qui n'est pas à notre avantage comme suit :

Imaginons que nous allouons pleins de blocs mémoire et que par chance, ils sont tous parfaitement alignés les uns à la suite des autres. Si on décide de libérer un élément au milieu des autres blocs non libres, le bloc récemment libéré sera donc laissé vide parmis les autres utilisés. Il sera également marqué comme étant l'adresse du premier bloc mémoire libre, même si les autres blocs suivants immédiats ne sont pas libres, ceux qui eux sont libres se trouvent beaucoup plus loin dans la structure.

Maintenant, si on alloue un bloc qui rentre parfaitement, pas de problème. Mais si nous devons diviser ce bloc pour l'aligner correctement avec les autres, car il est plus grand que celui que nous avions libéré précédemment, alors il sera beaucoup plus fastidieux et plus long de continuer à parcourir la liste jusqu'au

prochain bloc libre que si nous avions déjà les blocs libres regroupés entre eux en bout de chaîne. On préfère avoir un gros regroupement de cases libres dans la mémoire qui ne peut se situer en fin de structure, que si celui-ci s'y trouve déjà, que de déplacer tous les blocs vers la fin de la structure. On ne gère donc pas le "déplacement" de ce bloc vide en position peu avantageuse vers une position beaucoup plus intéressante.

Lorsque *refinc()* reçoit un pointeur qui n'a pas été retourné par *mymalloc()* la valeur de retour est **-1.** De même lorsque le pointeur a bien été retourné par *malloc()*, mais bien été désalloué car le compteur était à zéro, la valeur de retour sera **-1.**

void myfree(void *ptr):

Enfin, la fonction myfree(), qui a un comportement assez proche de free() de la librairie standard. "Est-ce que les blocs libérés sont retournés immédiatement à *free()* ? " -> Non, et c'est pour que notre algorithme garde un comportement prévisible et similaire à la gestion de la mémoire dans la librairie standard. Les blocs retournés à *myfree()* sont donc simplement marqués comme libres, mais pas retournés à *free()*. Ensuite, tant que le bloc suivant, dans la même page, est libre alors fusionner ensemble. On à estimé que ce n'était pas dans notre intérêt de immédiatement retourner les petits blocs mémoires au système d'exploitation. Mais si la première page est libre en entier, on l'élimine en utilisant la fonciton free() de librairie standart.

Affichage: Sortie telle qu'affiché par CLion (Jetbrains.com) avec c99.

```
int* testint1 = mymalloc(sizeof(int));
char* testchar = mymalloc(sizeof(char));
int* testint2 = mymalloc(sizeof(int));
int* testint3 = testint2;
int ref = refinc(testint3);
printf("Nombre reference apres refinc: %d\n",ref);
```

```
Nombre reference apres refinc: 2
```

```
*testint1 = 10;
printf("L'entier du testint1: %d\n", *testint1);
```

```
L'entier du testint1: 10
```

```
imprimer();
imprimervide();
```

La structure du memoire:
{{ TaillePage = 4096 **|** (size: 16) (compteur: 1) (Non libre) **|**
-> **|** (size: 16) (compteur: 1) (Non libre) **| ->** **|** (size: 16) (compteur: 2) (Non libre) **|**
-> **|** (size: 3952) (compteur: 0) (Libre) **| }}**
Liste de memoire libre : **|** (size: 3952) **|->**

```
myfree(testint3);
myfree(testchar);
imprimer();
imprimervide();
```

La structure du memoire:
{{ TaillePage = 4096 **|** (size: 16) (compteur: 1) (Non libre) **|->|** (size: 16) (compteur: 0) (Libre) **|**
-> **|** (size: 16) (compteur: 1) (Non libre) **| ->** **|** (size: 3952) (compteur: 0) (Libre) **| }}**
Liste de memoire libre : **|** (size: 16) **|->|** (size: 3952) **|->**

```
myfree(testchar);
```

Erreur myfree(): Pointeur deja liberee

```
int ref2 = refinc(testchar);
```

Erreur refinc(): Pointeur deja liberee

```
printf("Nombre reference apres refinc: %d\n",ref2);
```

Nombre reference apres refinc: -1

```
myfree(NULL);
```

Erreur trouverCase(): Pointeur null

```
void* erreur1 = mymalloc(-1);
```

Erreur mymalloc(): Taille null ou negative

```
void* erreur2 = mymalloc(17000000);
```

Erreur mymalloc(): Depasse taille maximum 16Mo

```
printf("Exemple de fusion: \n");
myfree(testint1);
imprimer();
imprimervide();
```

Exemple de fusion:
La structure du memoire:
{{ TaillePage = 4096 **|** (size: 64) (compteur: 0) (Libre) **|->|** (size: 16) (compteur: 1) (Non libre) **|**
-> **|** (size: 3952) (compteur: 0) (Libre) **| }}**
Liste de memoire libre : **|** (size: 64) **|->|** (size: 3952) **|->**

```
myfree(testchar);
```

Erreur trouveCase(): Memoire non allouer par mymalloc

```
printf("Demander plus de 4ko memoire: \n");
void* testgros = mymalloc(5000);
imprimer();
imprimervide();
```

Demander plus de 4ko memoire:
La structure du memoire:
{{ TaillePage = 4096 **|** (size: 64) (compteur: 0) (Libre) **|->|** (size: 16) (compteur: 1) (Non libre) **|**
-> **|** (size: 3952) (compteur: 0) (Libre) **| }}**
-> {{ TaillePage = 8192 **|** (size: 5008) (compteur: 1) (Non libre) **|**
-> **|** (size: 3152) (compteur: 0) (Libre) **| }}**
Liste de memoire libre : **|** (size: 64) **|->|** (size: 3952) **|->|** (size: 3152) **|->**

```
void* test4 = mymalloc(64);
myfree(testint2);
myfree(test4);
printf("Detruire la page libre si c'est la premiere page de la liste
                                ou la seul page restante\n");
imprimer();
imprimervide();
```

Detruire la page libre si c'est la premiere page de la liste ou la seul page restante
La structure du memoire:
{{ TaillePage = 8192 **|** (size: 5008) (compteur: 1) (Non libre) **| -> |** (size: 3152) (compteur: 0) (Libre) **| }}**
Liste de memoire libre : | (size: 3152) **|->**

Pour aller plus loin :

Voici ce que nous obtenons en sortie lorsque nous faisons appel à *Valgrind*, utilitaire très pratique pour observer ce qu'il se passe avec la mémoire, le compilateur et les allocations dynamiques de la mémoire. Comme vous pouvez le voir, avec l'exécutions des test qui ont été mis en ligne sur StudiuM, il n'y a aucune fuite mémoire !

```
~  gcc -o mymalloc -g mymalloc.c
~ valgrind --tool=memcheck --leak-check=yes ./mymalloc
==4771==
Nombre reference apres refinc: 2
L'entier du testint1: 10
La structure du memoire:
{{ TaillePage = 4096  | (size: 16) (compteur: 1) (Non libre)  | ->  | (size: 16) (compteur: 1) (Non libre)  | ->
| (size: 16) (compteur: 2) (Non libre)  | ->  | (size: 3856) (compteur: 0) (Libre) |  }}
Liste de memoire libre : | (size: 3856) |->

==4771== Conditional jump or move depends on uninitialised value(s)
==4771==    at 0x10000191B: detruirepage (mymalloc.c:274)
==4771==    by 0x100001B41: myfree (mymalloc.c:347)
==4771==    by 0x100001BF6: main (mymalloc.c:366)
==4771==
La structure du memoire:
{{ TaillePage = 4096  | (size: 16) (compteur: 1) (Non libre)  | ->  | (size: 16) (compteur: 0) (Libre) | ->  |
(size: 16) (compteur: 1) (Non libre)  | ->  | (size: 3856) (compteur: 0) (Libre) |  }}
Liste de memoire libre : | (size: 16) |-> | (size: 3856) |->

Erreur myfree(): Pointeur deja liberee
Erreur refinc(): Pointeur deja liberee
Nombre reference apres refinc: -1
Erreur trouverCase(): Pointeur null
Erreur mymalloc(): Taille null ou negative
Erreur mymalloc(): Depasse taille maximum 16Mo
Exemple de fusion:
==4771== Conditional jump or move depends on uninitialised value(s)
==4771==    at 0x1000014CB: fusionner (mymalloc.c:168)
==4771==    by 0x100001B38: myfree (mymalloc.c:345)
==4771==    by 0x100001C6C: main (mymalloc.c:376)
==4771==
```

```
La structure du memoire:
{{ TaillePage = 4096  | (size: 96) (compteur: 0) (Libre) | ->  | (size: 16) (compteur: 1) (Non libre)  | ->  | (size: 3856) (compteur: 0) (Libre) |  }}
Liste de memoire libre : | (size: 96) |-> | (size: 3856) |->

Erreur myfree(): Pointeur deja liberee
Demander plus de 4ko memoire:
La structure du memoire:
{{ TaillePage = 4096  | (size: 96) (compteur: 0) (Libre) | ->  | (size: 16) (compteur: 1) (Non libre)  | ->  | (size: 3856) (compteur: 0) (Libre) | }}
  -> {{ TaillePage = 8192  | (size: 5008) (compteur: 1) (Non libre)  | ->  | (size: 3120) (compteur: 0) (Libre) |  }}
Liste de memoire libre : | (size: 96) |-> | (size: 3856) |-> | (size: 3120) |->

==4771== Conditional jump or move depends on uninitialised value(s)
==4771==    at 0x1000014CB: fusionner (mymalloc.c:168)
==4771==    by 0x100001B38: myfree (mymalloc.c:345)
==4771==    by 0x100001CC6: main (mymalloc.c:385)
==4771==
==4771== Conditional jump or move depends on uninitialised value(s)
==4771==    at 0x1000014CB: fusionner (mymalloc.c:168)
==4771==    by 0x100001B38: myfree (mymalloc.c:345)
==4771==    by 0x100001CCF: main (mymalloc.c:386)
==4771==
==4771== Invalid write of size 8
==4771==    at 0x100001576: fusionner (mymalloc.c:180)
==4771==    by 0x100001B38: myfree (mymalloc.c:345)
==4771==    by 0x100001CCF: main (mymalloc.c:386)
==4771==  Address 0x202019a00 is not stack'd, malloc'd or (recently) free'd
==4771==
==4771==
==4771== Process terminating with default action of signal 11 (SIGSEGV)
==4771==  Access not within mapped region at address 0x202019A00
==4771==    at 0x100001576: fusionner (mymalloc.c:180)
==4771==    by 0x100001B38: myfree (mymalloc.c:345)
==4771==    by 0x100001CCF: main (mymalloc.c:386)
==4771==  If you believe this happened as a result of a stack
==4771==  overflow in your program's main thread (unlikely but
==4771==  possible), you can try to increase the size of the
==4771==  main thread stack using the --main-stacksize= flag.
==4771==  The main thread stack size used in this run was 8388608.
==4771==
==4771== HEAP SUMMARY:
==4771==     in use at exit: 35,754 bytes in 169 blocks
==4771==   total heap usage: 190 allocs, 21 frees, 44,202 bytes allocated
==4771==
==4771== 48 bytes in 2 blocks are possibly lost in loss record 21 of 47
```

==4771== at 0x1000DA6EA: calloc (in /usr/local/Cellar/valgrind/3.14.0/lib/valgrind/vgpreload_memcheck-amd64-darwin.so)
==4771== by 0x1005E6742: map_images_nolock (in /usr/lib/libobjc.A.dylib)
==4771== by 0x1005F955F: __objc_personality_v0 (in /usr/lib/libobjc.A.dylib)
==4771== by 0x10000947A: dyld::notifyBatchPartial(dyld_image_states, bool, char const* (*)(dyld_image_states, unsigned int, dyld_image_info const*), bool, bool) (in /usr/lib/dyld)
==4771== by 0x10000962D: dyld::registerObjCNotifiers(void (*)(unsigned int, char const* const*, mach_header const* const*), void (*)(char const*, mach_header const*), void (*)(char const*, mach_header const*)) (in /usr/lib/dyld)
==4771== by 0x100241A26: _dyld_objc_notify_register (in /usr/lib/system/libdyld.dylib)
==4771== by 0x1005E6233: environ_init (in /usr/lib/libobjc.A.dylib)
==4771== by 0x1001D8E35: _os_object_init (in /usr/lib/system/libdispatch.dylib)
==4771== by 0x1001E4AD1: libdispatch_init (in /usr/lib/system/libdispatch.dylib)
==4771== by 0x1000E89C4: libSystem_initializer (in /usr/lib/libSystem.B.dylib)
==4771== by 0x10001C591: ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==4771== by 0x10001C797: ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==4771==
==4771== LEAK SUMMARY:
==4771== definitely lost: 0 bytes in 0 blocks
==4771== indirectly lost: 0 bytes in 0 blocks
==4771== possibly lost: 48 bytes in 2 blocks – Du a l'installation de valgrind sur MacOS (problème connu)
==4771== still reachable: 12,648 bytes in 9 blocks
==4771== suppressed: 23,058 bytes in 158 blocks
==4771== Reachable blocks (those to which a pointer was found) are not shown.
==4771== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==4771==
==4771== For counts of detected and suppressed errors, rerun with: -v
==4771== Use --track-origins=yes to see where uninitialised values come from
Notre programme effectue 6 tests sur les 6 qui étaient proposés sur Studium.

L'affichage de test.c:

```
Test 1 OK
Test 2 OK
Test 3 OK
Test 4 OK
Test 5 OK
Test 6 OK
Test 6 632474318 OK
Done
```

Bibliographie :

14Memory-2x2.pdf
Memory management with mmap - University of Utah Engeneering
Skip-list - Geeks-for-Geeks
Valgrind - Stack Overflow - Possible Memory Leak Valgrind in OSX - Best Answer
malloc() — Reserve Storage Block - IBM® Knowledge Center
Wikibooks.org - Programmation C - Types_de_base
Tutorials Point - C Programming - C Type Casting