TP de SEPC
Révision de C

Grégory Mounié (CC-BY-SA ⓒⓘⓢ)*

2023-2024

Préambule

Ce TP a pour but de vous faire réviser le langage C. Vous devriez pouvoir réaliser chacun des TPs. N'hésitez pas à poser des questions pendant la séance si vous bloquez sur un point.

Attendus

Acceptable : Faire les exercices de la section 3 page 3 et passer avec succès leurs tests. Savoir utiliser gdb valgrind et votre compilateur en ligne de commande (section 2, page 2). L'exercice de la section 4 page 6 également si vous suivez le cours de PCSEF.

Bien : Avoir aussi terminé l'exercice de la section 5 page 6.

Très bien : Avoir terminé correctement tous les exercices.

Excellent : Vous avez déjà bien commencé votre TP 1 à rendre

Si vous êtes complètement débutant en C

Il va falloir changer cet état de fait rapidement, idéalement dans les deux semaines, pour atteindre le niveau d'un débutant avancé. En effet, de nombreux cours vont s'appuyer sur C dans votre 2A et ne vous attendront pas.

Plusieurs méthodes, probablement à utiliser en même temps avec la proportion qui vous convient :

1. Le livre *Modern C* de Jens Gustedt est organisé par niveau de "grand débutant" à "expert" (PDF disponible à `https://hal.inria.fr/hal-02383654`, et les sources à `https://gitlab.inria.fr/gustedt/modern-c`). Vous devriez au moins réécrire

*une partie vient de slides de Matthieu Moy, avec quelques contributions de François Broquedis, Frédéric Pétrot et Renaud Lachaize

du bon code C. 1000 lignes, qui compilent et s'exécute correctement, me semble le strict minimum ;

2. Lire de bons livres sur C (papier ou web) : *Introduction au langage C* de Bernard Cassagne dont la dernière version est à `http://matthieu-moy.fr/cours/poly-c/` ou la référence en anglais par les créateurs du langage, dont la première édition est dans le domaine public ;
   `https://archive.org/details/TheCProgrammingLanguageFirstEdition`.
3. Faire un des nombreux MOOC sur le sujet ;
4. Faire les très nombreux exercices de `https://exercism.io` ;

1. Introduction

Vous devez récupérer les squelettes de code et les tests avec la commande :

```
git clone https://github.com/gmounie/ensimag-rappeldec.git
```

Cela va créer un répertoire `ensimag-rappeldec`.
Le code est à écrire dans le répertoire `ensimag-rappeldec/src`.
Pour compiler, vous devrez ajouter un fichier `AUTHORS` dans la racine du projet et y ajouter vos prénoms, noms et login et utiliser les makefiles créés par cmake pour la compilation.
Un exemple de fichier `AUTHORS` :

```
Alice DOE doea
Bob SMITH smithb
```

La séquence des commandes pour créer les makefiles, compiler et lancer les tests avec l'ajout et édition de `ensimag-rappeldec/AUTHORS`) est :

```
cd ensimag-rappeldec
emacs AUTHORS # Penser à sauvegarder le fichier
cd ensimag-rappeldec/build
cmake .. # Il y a bien deux points ! Le but est de compiler dans
↪  build
make
make test
make check
```

2. 30 minutes de tuto : les bugs mémoires courants en C

Les tutoriaux sont assez détaillés (beaucoup de texte). Les exercices des sections suivantes seront volontairement moins guidés (comme la sec. 3, page

3). Ne restez pas bloqué dans cette section. Revenez-y plus tard, quand vous aurez à déboguer votre code

Nous avons déposé cette partie dans *Wikiversity* (une branche de Wikipedia). Elle est accessible, sans VPN, de manière pérenne à `https://fr.wikiversity.org/wiki/Débogage_avancé`.

(et en annexe dans ce PDF si besoin, à cf. annexe A, page 8)

Nous vous demandons de passer un peu de temps à faire les TP 1 à 7 (ou juste les premiers), pour vous approprier ou réapproprier les outils d'un débogage rapide, facile et efficace, mais pas trop longtemps non plus. Essayer d'utiliser les outils en situation, en faisant les autres exercices.

Les pointeurs permettent de manipuler arbitrairement la mémoire. Leur mauvaise programmation est la source de nombreux problèmes. Le but de cette partie est de vous faire manipuler les outils de débogage sur de petits exemples (petits en taille, pas en difficulté).

Les codes des 7 TP sont dans le répertoire `bug-training/`. Ces codes sont compilés par le Makefile généré par CMake, mais pas forcément avec les options que vous voulez avoir pour chacune des questions. Les réponses des TP vous expliquent comment les compiler avec les bonnes options dans votre terminal. **N'hésitez pas à déroulez les réponses dans Wikiveristy, même avant d'avoir commencer le TP, pour vous donner une idée de ce qui est attendu** (`[Dérouler]`, à droite de la solution).

1. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Pile_d'appels`
2. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Débordement_de_pile`
3. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Double_libération`
4. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Allocation:_0,_Libération:_1`
5. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Débogage_à_la_volée`
6. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Observation_à_la_loupe`
7. `https://fr.wikiversity.org/wiki/Débogage_avancé/Travail_pratique/Allocation_dans_la_pile`

3. Les listes chainées

Dans le répertoire `src`, modifier le fichier `listechainee.c` pour écrire un module de gestion de listes simplement chainées. Ce module contient les fonctions suivantes à implémenter :

```c
/* Affiche les éléments de la liste passée en paramètre sur la sortie
 * standard. */
void affichage_liste(struct elem *liste);

```

```c
/* Crée une liste simplement chainée à partir des nb_elems éléments du
 * tableau valeurs. */
struct elem *creation_liste(long unsigned int *valeurs, size_t nb_elems);

/* Libère toute la mémoire associée à la liste passée en paramètre. */
void destruction_liste(struct elem *liste);

/* Inverse la liste simplement chainée passée en paramètre. Le
 * paramètre liste contient l'adresse du pointeur sur la tête de liste
 * à inverser. */
void inversion_liste(struct elem **liste);
```

3.1. Utilisez votre debugger ici aussi !

Il est difficile, sur des petits exemples académiques de vous convaincre que passer 15 minutes à apprendre à vous servir d'un débogueur, vous économisera des heures d'édition/compilation plus tard et vous permet d'explorer une exécution en cours avec plus de facilité et de souplesse.

D'expérience, le saut sera franchi pour tout le monde lorsque vous n'aurez plus le choix, devant un projet technique (similaire au « Projet Système » (PCSEA)), gros, mal instrumenté ou pas facilement recompilable. C'est dommage, car après, vous utiliserez votre débogueur quasi-systématiquement vous demandant comment vous faisiez avant.

Ensuite, vous allez lire la documentation de votre débogueur, et y découvrir les fonctions avancées que vous auriez rêvées de connaître avant. Alors, autant partir de la fin, et vous faire manipuler un exemple avancé de GDB.

Un débogueur permet de lire, et changer, simplement les valeurs des variables, de la mémoire, l'instruction courante ou d'explorer la pile d'appel avec les variables locales des fonctions.

Après l'écriture de `affichage_liste`, il est facile d'ajouter une fonction qui affiche les valeurs des têtes de listes et le contenu de la liste.

Exemple :

```c
void debug_inversion(struct elem *h1, struct elem *h2) {
   printf("head1= %p, head2= %p\n", h1, h2);
   affichage_liste(h1);
   affichage_liste(h2);
}

void inversion_liste(struct elem **liste) {
  struct elem *one_list_head= *liste;
  struct elem *another_list_head;
...
```

```
  Inversion loop is here, e.g. at line 90
...
}
```

Puis lancez GDB sur le programme et définissez un breakpoint. Sur ce breakpoint ajoutez une commande qui va appeler la fonction `debug_inversion` avec les bons arguments. À chaque fois que le breakpoint est atteint, la commande s'exécute.

```
$ gdb listechainee
(gdb) break listechainee.c:90
....
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>call debug_inversion(one_list_head, another_list_head)
>end
(gdb) run
...
(gdb) cont
...
(gdb) cont 10 # passe 9 fois le breakpoint (stop à la 10 ième)
```

3.2. Rappel sur l'algorithme d'inversion de liste simplement chaînée

Contactez votre enseignant si vous avez des soucis d'ordre d'algorithmique !

Quelques explications rapides pour une inversion itérative. Le principe de base est de remplir à l'envers une liste temporaire. Pour cela, il suffit de supprimer le premier élément de la liste originale puis de l'insérer en tête dans la liste temporaire. Ces opérations de suppression du premier, puis son insertion en tête sont répétées tant que la première liste n'est pas vide.

À la fin, il suffit de mettre à jour la tête de liste originale avec la tête de la liste temporaire.

Il est possible de faire ensemble ces deux opérations, ce qui permet d'éviter de faire quelques affectations de pointeurs inutiles.

3.2.1. Pour réfléchir un peu...

Pourquoi faut-il éviter de faire une inversion récursive (fonctionnelle) en C ? La réponse est dans la section A.4, 12.

4. Les opérateurs binaires

Dans le répertoire `ensimag-rappeldec/src`, compléter le fichier `binaires.c`. La fonction `unsigned char crand48()`, à chaque appel, elle devra appliquer à la variable globale X la fonction suivante :

$$X_{n+1} = (aX_n + c) mod(m)$$

avec $m = 2^{48}$, `a = 0x5DEECE66D` et `c = 0xB`.

La fonction retourne ensuite l'octet correspondant aux bits 32 à 39 (en commençant la numérotation à 0 pour les bits de poids faibles).

Vous devrez utiliser les opérateurs arithmétiques `*`, `+` et `%` et les opérateurs bits à bit comme `<<`, `>>`, `&`, `|`, `~`, `^` (Décalage à gauche, Décalage à droite, ET binaire, OU binaire, NOT binaire, XOR binaire).

5. Allocation et ramasse-miette

Un ramasse-miette (Garbage Collector) généraliste performant est difficile à écrire. C'est toujours un sujet de recherche actif. Java et Go sont deux exemples, différents, d'implantations.

La programmation dun ramasse-miette simplifié pour un pool d'objets identiques, en nombre borné, et dont les références sont très bien encadrées, est beaucoup plus simple. Vous implantez trois fonctions qui initialisent, allouent et ramassent des éléments de listes chaînées. Ces fonctions seront utilisées par un programme de test fourni.

5.1. Allocation et ramasse-miette

Le but est d'écrire la fonction d'allocation et la fonction de ramassage d'éléments de liste chaînée :

```
1  struct elem {
2    int val;
3    struct elem *next;
4  };
```

Les éléments sont alloués au sein d'un bloc de mémoire pré-réservé. L'utilisation de chaque élément est notée dans un vecteur de booléens mis-à-jour par les fonctions. Ce vecteur de booléens permet à la fonction d'allocation de savoir si une zone du bloc correspondant à un élément est libre ou pas. Les fonctions de manipulation du vecteur de booléens sont fournies (lire la valeur d'un booléen, l'écrire, mettre tout le vecteur à `false`).

5.2. Fonctions interdites

Vous ne devez pas utiliser les fonctions qui vous permettraient de gérer des allocations dynamiques pour les `struct elem` tel que `malloc()`, `free()`, `realloc()`, etc.

Vous pouvez bien sûr ajouter vos propres structures de données et les initialiser correctement. Suivant votre façon de coder d'autres fonctions peuvent être utiles.

5.3. Le travail à réaliser

Vous devez implanter dans le fichier `src/elempool.c` les deux fonctions

```
1 struct elem *alloc_elem();
2 void gc_elems(struct elem **heads, int nbheads);
```

Ces deux fonctions manipulent des adresses qui sont dans le bloc de mémoire `static unsigned char *memo` de `src/elempool.c`
alloué par la fonction `void init_elems()`.

Le bloc de mémoire alloué a une taille de 1000 `struct elem`. La fonction d'initialisation est appelée au début de chaque fonction de tests.

Pour tracer l'utilisation du bloc, des fonctions de manipulation d'un vecteur de 1000 booléens sont fournies. Chaque booléen peut être lu ou écrit individuellement. Il est également possible de les passer tous à `false (0)`.

```
1 bool bt1k_get(long unsigned int n); // obtenir la valeur du bit n
2 void bt1k_set(long unsigned int n, bool val); // mettre le bit n à val
3 void bt1k_reset(); // tous les bits à false
```

La fonction `struct elem *alloc_elem(void)` renvoie :

0 ou NULL : si il n'y a pas de portion du bloc mémoire disponible,

une adressse : celle d'une portion du bloc qui sera utilisée comme un `struct elem`.

Pour cela, elle utilisera le vecteur de booléens pour trouver une zone libre. Dans cet exercice un simple parcours linéaire depuis le début du vecteur pour trouver la valeur que vous cherchez suffira.

La fonction `void gc_elems(struct elem **heads, int nbheads)` a
pour but de trouver les portions du bloc qui sont et ne sont pas utilisées. Pour cela elle reçoit en argument un tableau contenant les adresses de 0, une ou plusieurs têtes de liste. Ces listes contiennent les éléments utilisés. **Tout élément qui n'est pas chaîné dans une des listes passées en paramètre, au moment de l'appel de `gc_elems()`, est donc libre.**

Le but de la fonction sera donc de mettre à `true` tous les booléens correspondant à un élément utilisé et de mettre à `false` les autres.

6. Hello World !

Dans le répertoire `src`, compléter le fichier C de nom `hello.c`.
Votre programme affichera :

$$\forall\ p \in world,\ hello\ p$$

7. Les nombres flottants

Dans le répertoire `src`, compléter le fichier C de nom `flottants.c`
(cf. `https://0.30000000000000004.com/` pour vous rendre compte que dans les autres langages, sauf pour une poignée comme Raku, ce n'est pas mieux)

7.1. question facile

Afficher la valeur de l'opération $0.1+0.2-0.3$ en effectuant le calcul avec les 3 tailles de nombres flottants (Il y a bien trois tailles !). Les constantes devront aussi être initialisées avec le bon type. Pour cela, il faut parfois leur adjoindre le bon suffixe (`0.1f` ou `0.1L`).
Chacune de ces trois valeurs sera affichée sur une ligne en notation `[-]d.ddddddedd` (avec les puissances de 10). Les trois lignes seront à afficher dans l'ordre croissant de la taille en octet des 3 types flottants.
Indication : vous devriez créer des variables intermédiaires du bon type.

7.2. question moins facile

Idem, mais sans créer de variables intermédiaires.

8. Les fonctions : les pointeurs de fonctions

Dans le répertoire `ensimag-rappeldec/src`, compléter le fichier `fqsort.c` pour trier le tableau de nombres complexes en fonction de l'argument complexe des nombres.
Vous utiliserez la fonction `qsort` de la bibliothèque standard.

A. Les bugs mémoires courants en C

Cette annexe est la version d'origine des tutoriaux de Wikiversity.
Cette partie du TP a aussi été déposée dans wikiversity (une branche de wikipedia). Elle est accessible, sans VPN, de manière pérenne à `https://fr.wikiversity.org/wiki/D%C3%A9bogage_avanc%C3%A9`.
Les pointeurs permettant de manipuler arbitrairement la mémoire. Leur mauvaise programmation est la source de nombreux problèmes. Le but de cette partie est de vous faire manipuler les outils de débogage sur de petits exemples (petits en taille, pas en difficulté).

A.1. Assertions, GDB et pile d'appels de fonctions

Le code `src/bug_assert.c` détecte un prédicat booléen faux (ici, pour faire simple, toujours faux, et `false` vaut `0`).

Lancez le programme `./bug_assert` compilé dans le répertoire `build/`. Remarquez que le message d'erreur donne la ligne et le fichier du programme qui a détecté le prédicat faux.

```
you@ensipc$ ./bug_assert
bug_assert:
↪  /home/gregory/ensimag-rappeldec/bug-training/bug_assert.c:20: main:
↪  Assertion `0' failed.
Abandon
```

Lancez gdb avec le programme en argument dans un terminal. Affichez graphiquement le code du programme à déboguer. Mettre un breakpoint à main. Continuez. À l'assertion, affichez la pile d'appel et remonter dans la pile d'appel de fonction jusqu'au code source impliqué `assert(false);`.

```
you@ensipc$ gdb ./bug_assert
[...]
(gdb) layout src
(gdb) break main
(gdb) run
(gdb) cont
(gdb) where
(gdb) up
(gdb) up
(gdb) up
(gdb) up
(gdb) quit # confirm
```

A.2. Double libération : garde-fou de la librairie C et Valgrind

Il est compliqué de suivre dans l'exécution d'un programme si une zone de mémoire allouée dynamiquement dans le tas (avec malloc) est encore utilisée. Mais, il faut pourtant bien la libérer un jour pour pouvoir la recycler et ne pas exploser en utilisation de la mémoire. La libération est donc sujette à deux bugs courants : libérer trop tôt une zone encore utilisée, et la libérer deux fois dans deux endroits différents du code. Ce TP tourne autour de ce deuxième bug.

Les langages interprétés (Python, Perl, Ruby, Julia, etc.) et certains langages compilés (Java, Go, Haskell, etc.) font cette libération en exécutant un ramasse-miettes, c'est-à-dire du code dédié à la détection de ces libérations. D'autres méthodes existent comme le

compteur de référence d'Objective-C ou des règles strictes d'emprunt d'un pointeur, en Rust.

Le code `src/bug_doublefree.c` libère deux fois le tableau alloué, une première fois juste avant la fin de la fonction `fibon()` et une fois juste après.

Lancez le programme `./bug_doublefree`. C'est la bibliothèque C qui détecte le problème au moment du second free.

```
you@ensipc$ ./bug_doublefree
free(): double free detected in tcache 2
Abandon
```

Lancez le programme avec valgrind. Il détecte également le problème et donne les différentes lignes du programme pour l'allocation et les deux libérations. Valgrind donne aussi en fin un résumé de ce qu'il a observé (une allocation et deux libérations).

```
you@ensipc$ valgrind ./bug_doublefree
[...]
==31719== Invalid free() / delete / delete[] / realloc()
==31719==    at 0x484217B: free (vg_replace_malloc.c:872)
==31719==    by 0x109291: main (bug_doublefree.c:24)
==31719==  Address 0x4a8f040 is 0 bytes inside a block of size 400
↪  free'd
==31719==    at 0x484217B: free (vg_replace_malloc.c:872)
==31719==    by 0x1091EF: fibon (bug_doublefree.c:13)
==31719==    by 0x109285: main (bug_doublefree.c:22)
==31719==  Block was alloc'd at
==31719==    at 0x483F7B5: malloc (vg_replace_malloc.c:381)
==31719==    by 0x10923F: main (bug_doublefree.c:19)
==31719==
==31719== HEAP SUMMARY:
==31719==     in use at exit: 0 bytes in 0 blocks
==31719==   total heap usage: 1 allocs, 2 frees, 400 bytes allocated
==31719==
==31719== All heap blocks were freed -- no leaks are possible
==31719==
==31719== For lists of detected and suppressed errors, rerun with: -s
==31719== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
↪  0)
```

A.3. Libération abusive ! Libérer ce qui n'est pas alloué.

Il existe deux principales façons d'allouer de la mémoire dans un programme : dans la pile (les variables locales) et dans le tas (malloc/free). Un pointeur dans un langage

comme C ou C++ peut donc pointer de la mémoire valide mais qui n'a jamais été allouée avec un malloc.

La fonction `free()` sert exclusivement à libérer de la mémoire allouée dans le tas avec `malloc`, `calloc` ou `realloc`.

Le code `src/bug_stackallocthenfree.c` utilise un tableau alloué dans la pile. Ce tableau ne doit surtout pas être libéré. Cela n'a pas de sens du point de vue de la mémoire.

Lancez le programme `./bug_stackallocthenfree` sans valgrind puis avec valgrind. Valgrind vous indique que votre variable a été allouée dans la frame de la fonction main (les variables locales de main).

```
you@ensipc$ ./bug_stackallocthenfree
munmap_chunk(): invalid pointer
you@ensipc$ valgrind ./bug_stackallocthenfree
[...]
==11348== Invalid free() / delete / delete[] / realloc()
==11348==    at 0x484217B: free (vg_replace_malloc.c:872)
==11348==    by 0x1091DF: fibon (bug_stackallocthenfree.c:13)
==11348==    by 0x1092A6: main (bug_stackallocthenfree.c:22)
==11348==  Address 0x1fff0002e0 is on thread 1's stack
==11348==  in frame #2, created by main (bug_stackallocthenfree.c:16)
[...]
```

Pour confirmer exactement quelle est la variable dans la pile, il va falloir regarder les adresses des variables. Nous allons donc connecter valgrind et gdb et remonter la pile d'appel jusqu'au main afin d'afficher les adresses de ses variables. Noter que l'adresse fautive est `0x1fff0002e0`.

```
you@ensipc$ valgrind --vgdb=full --vgdb-error=0
↪  ./bug_stackallocthenfree &
[...] # copy gdb command: "target remote | vgdb --pid=..."
you@ensipc$ gdb ./bug_stackallocthenfree
[...]
(gdb) target remote | /usr/bin/vgdb --pid=14526 # paste
(gdb) cont # valgrind stop at bug free(0x1fff0002e0)
(gdb) where # call stack
(gdb) up 2 # 2 frames up to go to main
(gdb) print &p
$1 = (unsigned int (*)[100]) 0x1fff0002e0
(gdb) quit # confirm
```

C'est bien la variable p du main qui est injustement libérée !

A.4. Débordement de pile : gdb et la pile d'appel

Depuis la création et l'implantation du langage Algol dans les années 1958-1960, la plupart des langages de programmations permettent d'exécuter des fonctions récursives. L'idée de base est d'avoir une zone de mémoire dédiée à enregistrer les informations pour cela : la pile.

Un petit peu de pile est consommé par chaque appel récursif pour y stocker les variables locales, les arguments de la fonction, l'adresse de retour, tous liés à l'appel. Cet espace n'est rendu qu'à la terminaison de la fonction. Il n'est donc pas possible d'empiler à l'infini des appels récursifs en C. Certains langages fonctionnels comme OCaml, LISP (et ses descendants : Scheme, Clojure, etc.) ou Haskell, sont capables d'enlever une récursion terminale pour rendre le programme itératif et ainsi, éliminer le problème. Le langage Go, augmente la pile jusqu'à plusieurs Gio ce qui repousse le problème, mais ne l'élimine pas.

Le programme `src/bug_stackoverflow.c` utilise une liste circulaire pour appeler à l'infini une fonction récursive. Une fonction récursive utilise une pile pour ses variables locales et ses paramètres. La pile est juste un bloc de mémoire contigüe de quelques Mio par défaut. La commande `ulimit -a` vous donne la liste de vos limites.

Lancez le programme `./bug_stackoverflow`. C'est le système qui détecte l'utilisation illégale de mémoire (SEGFAULT) au-delà de la pile. Mais cela ressemble à tous les autres accès illégaux que vous pouvez faire avec un pointeur.

```
you@ensipc$ /bug_stackoverflow
Erreur de segmentation (core dumped)
```

Lancez le programme avec valgrind. Il détecte également le problème, mais surtout, il vous indique que c'est un débordement de pile et pas autre chose !

```
you@ensipc$ valgrind ./bug_stackoverflow
[...]
==8672== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==8672==
==8672== Process terminating with default action of signal 11 (SIGSEGV)
==8672==  Access not within mapped region at address 0x1FFE801FF8
==8672== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==8672==    at 0x109141: functional_list_length
↪  (bug_stackoverflow.c:22)
[...]
```

Pour déboguer vos codes (ici, comprendre que la liste est circulaire), vous avez souvent ajouté des printf pour obtenir des traces. Cela demande de savoir à l'avance ce que l'on veut observer, puis d'éditer le code, puis de recompiler, puis de réexécuter, puis de recommencer le tout si la trace ne suffit pas.

C'est un peu long, non ?

Lancez gdb avec le programme en argument dans un terminal. Démarrez le programme. Affichez le code de la fonction fautive. Demander l'affichage de `h` et de `h->next` à

chaque passage à la ligne 25. Démarrez le programme en confirmant le redémarrage depuis le début.

Voilà, vous avez votre trace ! Générer la trace en soi est long car gdb fait un arrêt à chaque passage à la ligne 25 et votre terminal n'est pas aussi rapide que vous le pensez, mais vous pouvez faire ńControl-Cż pour interrompre l'exécution.

```
you@ensipc$ gdb ./bug_stackoverflow
[...]
(gdb) run
(gdb) list
(gdb) where # then quit, otherwise, view roughtly 206000 frames
(gdb) dprintf 25,"h: %x, h->next: %x\n", h, h->next
(gdb) run # then confirm the restart from beginning
          # then Control-C to stop the execution
```

A.5. Attraper les bugs au vol et changer les valeurs des variables pendant l'exécution

Attention : le code tourne en boucle ! Il peut donc vider votre batterie ainsi que faire fondre la banquise !. Vous limiterez son temps d'exécution à 300 secondes max. Attention à bien mettre les parenthèses, sinon vous risquez de limiter aussi tous les processus que vous lancerez dans le même terminal.

Le code `src/bug_loop.c` est similaire à celui de la section A.4, page 12 mais sans déborder de la pile. Il tourne donc à l'infini. Pour gagner du temps, il indique aussi comment l'attraper au vol avec votre débogueur. Sinon vous pouvez obtenir le PID du processus avec la commande `ps`.

Lancez le programme en tâche de fond en limitant son temps d'exécution à 5 minutes (300 secondes). Demandez à votre débogueur de s'accrocher au processus en train de tourner avant qu'il soit fini. Affichez le code à proximité de l'arrêt. Avancez dans la boucle while jusqu'à la ligne 14 `l++`. Changez la valeur de h. Continuez jusqu'à la sortie de la fonction. Continuez jusqu'à la terminaison.

```
you@ensipc$ ( ulimit -t 300; ./bug_loop ) &
My PID is: 12345. Catch me with 'gdb ./bug_loop 12345' !
you@ensipc$ ps
[...]
 12345 pts/3    00:00:01 bug_loop
[...]
you@ensipc$ gdb ./bug_loop 12345
[...]
(gdb) list
(gdb) next # variable number (0-3) of next to arrive at l++
(gdb) next # 2 next in this example
```

```
(gdb) set variable h = 0 # change h
(gdb) finish # go to the end of the fonction
(gdb) next
(gdb) next
(gdb) next # end of procesus processus
(gdb) quit # no confirmation here as processus is dead
```

A.6. Observer une position mémoire précise

Dans cette partie, nous allons explorer des exemples de bugs très similaires, mais dont le comportement apparent est différent. L'algorithme de base est identique à celui de la section A.2, page 9 et la section A.3, page 10. Ici, nous allons dépasser, un peu, ou beaucoup de l'allocation initiale.

Le code `src/bug_overreadheap_tiny.c` lit une valeur juste un peu devant l'allocation. Cela provoque instantanément un SEGFAULT. Il est donc relativement aisé de trouver la source du problème avec gdb en demandant simplement les valeurs des variables. Valgrind permet aussi de trouver facilement la ligne de code (ici 27) avec l'erreur. Nous affichons (commande ńxż pour *examine*) préalablement les 20 premières entrées du tableau, en décimal (`/20d`)

```
you@ensipc$ ./bug_overreadheap_tiny
Erreur de segmentation (core dumped)
you@ensipc$ gdb ./bug_overreadheap_tiny
(gdb) layout src
(gdb) run
(gdb) x /20d p
(gdb) print p[i-2]
Cannot access memory at address 0x55595555929c
(gdb) quit # confirm
you@ensipc$ valgrind ./bug_overreadheap_tiny
[...]
==19345== Invalid read of size 4
==19345==    at 0x1091BC: fibon (bug_overreadheap_tiny.c:27)
==19345==    by 0x109276: main (bug_overreadheap_tiny.c:35)
==19345==  Address 0x404a8003c is not stack'd, malloc'd or (recently)
↪  free'd
==19345==
[...]
```

Le code `src/bug_overwriteheap_tiny.c` écrit une valeur juste un peu après l'allocation (un entier de 4 octets, 0 octet après, donc juste après). Lors d'une exécution normal, il ne se passe rien ! Il se termine parfaitement normalement. Mais le bug est bien là et valgrind est capable de le détecter

```
you@ensipc$ ./bug_overwriteheap_tiny
you@ensipc$ valgrind ./bug_overwriteheap_tiny
[...]
==19897== Invalid write of size 4
==19897==    at 0x1091D2: fibon (bug_overwriteheap_tiny.c:27)
==19897==    by 0x109276: main (bug_overwriteheap_tiny.c:36)
==19897==  Address 0x4a89c80 is 0 bytes after a block of size 40,000
↪  alloc'd
==19897==    at 0x483F7B5: malloc (vg_replace_malloc.c:381)
==19897==    by 0x109230: main (bug_overwriteheap_tiny.c:33)
==19897==
==19897==
[...]
```

Dans le code `src/bug_overwriteheap_large.c` l'allocation est 4 fois trop petite. Beaucoup d'écritures débordent. Néanmoins, la détection n'a lieu qu'au moment du free. Valgrind sait détecter le problème. Mais, si nous soupçonnons un débordement en écriture, nous pouvons demander à gdb de surveiller explicitement des modifications juste après notre allocation.

Cette technique est particulièrement utile lorsque quelque chose a modifié une de vos structures de données : vous ne savez pas quoi ou quand, mais vous savez où.

Démarrez gdb avec le programme. Mettez un breakpoint au malloc de la ligne 33. Faites le malloc. Mettez un watchpoint à la fin du malloc. Il serait possible d'exprimer le watch point avec des casts et des opérations sur les pointeurs, ce qui vous aiderait à comprendre le bug, mais vous allez utiliser directement la valeur affichée par le print, juste pour vous souvenir qu'un pointeur est un entier presque comme les autres, avec une arithmétique particulière fonction de la taille du type.

Attention, à quelques subtilités sur l'utilisation des interfaces en mode texte. Le code suivant ne fait pas de ńlayout srcż pour vous faciliter le copier-coller de la valeur du pointeur p.

```
you@ensipc$ ./bug_overwriteheap_large
double free or corruption (!prev)
you@ensipc$ gdb ./bug_overwriteheap_large
(gdb) break 33
(gdb) run
(gdb) next
(gdb) print p
$1 = (unsigned int*) 0x5555555592a0
(gdb) watch *(int *)(0x5555555592a0 + SIZE)
Hardware watchpoint 2: *(int *)(0x5555555592a0 + SIZE)
(gdb) cont
(gdb) print i # fail at 1/4 of the expected length
```

```
$2 = 2500
(gdb) print size
$3 = 10000
(gdb) quit # confirm
```

A.7. Tomber de Charybde (malloc/free) en Scylla (ne pas faire malloc/free)

Pour éviter le tourbillon de Charybde des allocations dynamiques dans le tas (heap), certains aventureux ou aventureuses sont prêts à se jeter dans la gueule du monstre Scylla des allocations dynamiques dans la pile (stack).

Le problème est que vous pouvez faire ce que vous voulez dans votre pile. Vous avez le droit d'y lire et d'y écrire partout ! Et, du coup tous les outils précédents deviennent beaucoup plus difficiles à exploiter.

Le code `src/bug_allocinstack.c` reproduit un schéma classique de la programmation en Python. Mais, pour allouer tous ses objets, Python utilise le tas (malloc) ! Même si la syntaxe est identique, Python ne fait pas du tout la même chose que ce code qui crée son objet dans la pile de la fonction.

```
you@ensipc$ ./bug_allocinstack
Erreur de segmentation (core dumped)
you@ensipc$ valgrind ./bug_allocinstack
[...] # no acurate indication
==21326== Invalid write of size 8
==21326==    at 0x1091A4: main (bug_allocinstack.c:28)
==21326==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Par soucis pédagogiques, j'ai omis deux points. D'abord, le compilateur pourrait vous prévenir dans les cas simples. C'est pour cela que le code fait deux opérations arithmétiques sur le pointeur. Cela suffit à lui masquer le problème. Il est donc crucial d'utiliser un compilateur récent et de lire les warnings ! Si votre compilateur C vous indique que vous faites un truc bizarre, il a surement raison. Ensuite, la technique du watch de la section A.6, page 14 fonctionne très bien aussi avec la pile et pourrait être exploitée ici pour savoir qui modifie la structure.

A.8. Analyse statique de votre compilateur

Pour cette partie, il faut au moins la version 10 de gcc.

Au prix d'un surcoût parfois important à la compilation, vous pouvez ajouter l'option `-fanalyzer` dans les FLAGS au début du fichier `CMakeLists.txt`.

Vous pouvez voir l'analyseur statique de GCC à l'uvre en recompilant l'exemple `bug_doublefree`. Il vous explique en détails la double libération.

```
you@ensipc$ emacs ../CMakeLists.txt # ajouter -fanalyzer dans les flags
you@ensipc$ rm -i ./bug_doublefree
you@ensipc$ make
[...] # décrypter le joli et long warning
```

A.9. WARNING(renoncement à Valgrind) : La surveillance des allocations par le code compilé

Au prix d'un surcoût à l'exécution et à **l'impossibilité d'utiliser facilement valgrind**, il est possible de demander à votre compilateur C de mieux détecter certains problèmes mémoire.

Si vous renoncez à Valgrind, ajoutez l'option `-fsanitize=address` dans les FLAGS au début du fichier `CMakeLists.txt`.