



Algorithmique et structures de données 2

Chapitre 1

Outils de débogage en C

Algorithmique et structures de données - acquis

- ▶ Chapitre 1 - Qu'est-ce qu'un algorithme
- ▶ Chapitre 2 - Terminaison, Correction, Complexité
- ▶ Chapitre 3 - Suites et Domination
- ▶ Chapitre 4 - Non déterminisme, parallélisme, pire des cas vs cas moyen
- ▶ Chapitre 5 - Liste chaînée
- ▶ Chapitre 6 - Arbres (binaires)
- ▶ Chapitre 7 - Organisation de la mémoire
- ▶ Chapitre 8 - Conteneur. Pile, File, Ensemble, Sac, Map
- ▶ Chapitre 9 - Arbre AVL
- ▶ Chapitre 10 - Graphe (parcours en profondeur, recherche de circuit, recherche du plus court chemin)

1. Organisation du cours
2. Quizz de rentrée
3. Le débogage
4. Comprendre le comportement du programme
5. Le *memory check*

Organisation du cours

- ▶ 1h30 de cours TD **sans machine**
- ▶ 3h sur les machines

en cas d'absence, prévenir le plus tôt possible **par mail**.

Concernant les mails

- ▶ champ objet précis ([ASD2-L2X] votre_objet),
- ▶ définissez une **signature** (nom complet, groupe),
- ▶ rien de confidentiel (pensez que c'est une carte postale),
- ▶ ne pas utiliser les majuscule mais *ceci* pour mettre en valeur un mot ou une phrase,
- ▶ pas de langage SMS,
- ▶ soyez bref(ve) et courtois(e),
- ▶ évitez l'humour et l'ironie (même avec un smiley),
- ▶ soyez modéré(e)s dans vos propos : vos messages restent,
- ▶ limitez, si possible, à un sujet le contenu de vos messages,
- ▶ ne transmettez pas un courrier reçu à d'autres personnes sans l'autorisation de l'émetteur,
- ▶ limitez la taille et le nombre de vos pièces jointes (privilégier un lien).

Algorithmique et structures de données 2

Planning prévisionnel

- ▶ Chapitre 1 - Outils de débogage en C
- ▶ Chapitre 2 - Révisions S1
- ▶ Chapitre 3 - Révisions S1
- ▶ Chapitre 4 - Automates
- ▶ Chapitre 5 - Algorithmique textuelle, distances de levenshtein
- ▶ Chapitre 6 - Tables de hachage
- ▶ Chapitre 7 - Tables de hachage
- ▶ Chapitre 8 - Plus longue sous-séquence commune
- ▶ Chapitre 9 - TBD
- ▶ Chapitre 10 - TBD

Évaluation

- ▶ TPs à rendre (pas plus d'un TP non rendu)
- ▶ 1 / 2 quizz en classe
- ▶ devoir sur table
- ▶ projet à rendre

1. Organisation du cours
2. Quizz de rentrée
3. Le débogage
4. Comprendre le comportement du programme
5. Le *memory check*

Sondage / quizz de rentrée (anonyme)

Avec votre voisin, depuis un téléphone



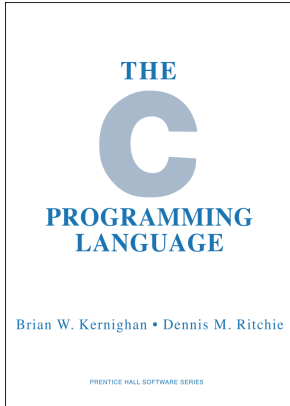
ou [https:](https://moodle.univ-paris8.fr/mod/evoting/client_poll.php?id=32&lang=fr)

[//moodle.univ-paris8.fr/mod/evoting/client_poll.php?id=32&lang=fr](https://moodle.univ-paris8.fr/mod/evoting/client_poll.php?id=32&lang=fr)

1. Organisation du cours
2. Quizz de rentrée
3. Le débogage
4. Comprendre le comportement du programme
5. Le *memory check*

À propos - Brian W. Kernighan

Premier livre sur le langage C (1978)



"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Traduction : Le débogage est deux fois plus difficile que l'écriture du code en premier lieu. Par conséquent, si vous écrivez le code aussi intelligemment que possible, vous n'êtes, par définition, pas assez intelligent pour le déboguer.

Le débogage, quand et comment ?

Le bogue : comportement inattendu du programme

▶ le compilateur râle et refuse de compiler 🙅

1. **lire** les messages d'erreurs
2. **aller voir l'emplacement signalé par le compilateur**
3. éventuellement faire une recherche (copier/coller l'erreur dans un moteur de recherche) pour comprendre sa signification

⚠ Le compilateur signale où il **détecte** l'erreur ⚠

(l'erreur peut-venir de la ligne au-dessus, ou d'une variable déclarée dans une autre fonction par exemple)

Le débogage, quand et comment ?

Le bogue : comportement inattendu du programme

- ▶ le compilateur râle et refuse de compiler 🙅
- ▶ *expectations* \neq *reality*

Le débogage, quand et comment ?

Le bogue : comportement inattendu du programme

- ▶ le compilateur râle et refuse de compiler 🙅
- ▶ *expectations* \neq *reality*
 - le programme ne s'arrête pas ♾
ex : une condition n'est pas validée alors qu'elle le devrait
 - erreur silencieuse : le résultat n'est pas celui attendu 🤔
ex : `somme(2,3)` renvoie 4
 - le programme crashe 💣
- ▶ bogue invisible 👁
ex : interaction inattendue avec le système, occupation mémoire démesurée

1. Organisation du cours
2. Quizz de rentrée
3. Le débogage
4. Comprendre le comportement du programme
5. Le *memory check*

Outils de débogage : gdb (1986)

Suivre l'exécution du programme pas à pas pour valider son comportement

1. placer des **points d'arrêt** ● pour mettre en pause le programme à un endroit donné :
 - numéro de ligne :
 - dans le main.c : `(gdb) b(reak) 10`
 - dans un module.c : `(gdb) b(reak) module.c:10`
 - appel de fonction : `(gdb) b(reak) fact_bug`
2. **afficher les valeurs des variables** :
 - `(gdb) p(rint) i`
 - `display i` : affichage de la variable à chaque arrêt du programme
 - `watch i` : interrompt l'exécution si `i` est modifiée et affiche l'ancienne et la nouvelle valeur
3. exécuter le programme pas à pas :
 - `(gdb) r(un)` : lance l'exécution
 - `(gdb) n(ext)` : exécute la ligne suivante
 - `(gdb) c(ontinue)` : continue jusqu'au point d'arrêt suivant
 - `(gdb) s(tep)` : comme `(gdb) n` sauf si la ligne contient un appel de fonction
⇒ on *rentre* dans l'appel
 - `(gdb) finish` : continue jusqu'à la fin de la fonction en cours

Afficher des tableaux dans gdb

- ▶ Situation 1 : `int T[5] = {8,1,4,3,1};`

```
(gdb) p T
$1 = {8,1,4,3,1}
```

- ▶ Situation 2 : on est `dans un appel de fonction` où T a été passé en paramètre (par exemple dans la fonction `tri_a_bulle(T, 5);`), alors T est traité comme un pointeur :

```
(gdb) p T
$2 = (int *) 0x7fffffffdf70
(gdb) p T[0]
$3 = 8
```

Afficher des tableaux dans gdb

- Situation 1 : `int T[5] = {8,1,4,3,1};`

```
(gdb) p T  
$1 = {8,1,4,3,1}
```

- Situation 2 : on est `dans un appel de fonction` où T a été passé en paramètre (par exemple dans la fonction `tri_a_bulle(T, 5);`), alors T est traité comme un pointeur :

```
(gdb) p T  
$2 = (int *) 0x7fffffffdf70  
(gdb) p T[0]  
$3 = 8
```

Solution ☒ : utiliser l'opérateur `@` : `p *array@len`

```
(gdb) p *T@5  
$4 = {8,1,4,3,1}
```

<https://sourceware.org/gdb/current/onlinedocs/gdb/Arrays.html>

Afficher des pointeurs dans gdb

```
1 typedef struct Element Element;
2 struct Element {
3     int nombre;
4     Element *suivant;
5 };
6 typedef struct Liste Liste;
7 struct Liste{
8     Element *premier;
9 };
10 int main(){
11     Liste *maListe = initialisation();
12     insertion(maListe, 4);
13     insertion(maListe, 8);
14     insertion(maListe, 15);
15     afficherListe(maListe);
16 }
```

<https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>

Afficher des pointeurs dans gdb

```
1 typedef struct Element Element;
2 struct Element {
3     int nombre;
4     Element *suivant;
5 };
6 typedef struct Liste Liste;
7 struct Liste{
8     Element *premier;
9 };
10 int main(){
11     Liste *maListe = initialisation();
12     insertion(maListe, 4);
13     insertion(maListe, 8);
14     insertion(maListe, 15);
15     afficherListe(maListe); // 8 -> 4 -> 15 -> NULL
16 }
```

<https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>

Afficher des pointeurs dans gdb

```
(gdb) p maListe
$1 = (Liste *) 0x5555555592a0
(gdb) p *maListe
$2 = {premier = 0x555555559200}
(gdb) p maListe->premier
$3 = (Element *) 0x555555559200
(gdb) p *maListe->premier
$4 = {nombre = 8, suivant = 0x5555555592e0}
(gdb) p *maListe->premier.nombre
Cannot access memory at address 0x8
(gdb) p maListe->premier.nombre
$5 = 8
(gdb) p *maListe->premier->suivant
$6 = {nombre = 4, suivant = 0x5555555592c0}
(gdb) p maListe->premier->suivant->suivant.nombre
$7 = 0
```

<https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>

Afficher des pointeurs dans gdb

Que fait l'instruction suivante ?

```
1 int * point = 3;
```

Avantages de l'utilisation du débbugger *vs logging*

- ▶ le débbugger s'arrête sur la ligne qui a fait crasher le programme
 - arrêt *dans le programme* : vérifier son code
 - arrêt *dans une librairie* : **afficher la pile d'appel** pour connaître la dernière fonction appelée
- ▶ pas besoin de modifier/recompiler/exécuter pour changer ce qu'on observe
- ▶ fonctions d'affichage en tampon (peuvent ne jamais s'afficher)

Segmentation fault ✨

Il s'agit d'un bug consistant pour un programme à tenter d'accéder à une partie de la mémoire qui ne lui appartient pas.



Utilisation gdb 🕵️ : segmentation fault

```
1 typedef struct MaStruct
2 {
3     int i;
4     int j;
5 }MaStruct;
6
7 int main()
8 {
9     MaStruct* pStruct1;
10    MaStruct* pStruct2 = NULL;
11
12    pStruct1->i = 10;
13    pStruct2->j = 5;
14
15    return 0;
16 }
```

Utilisation gdb 🕵️ : segmentation fault

```
$ gcc -g main.c
$ ./a.out
Erreur de segmentation (core dumped)
$ gdb ./a.out
Program received signal SIGSEGV, Segmentation fault.
0x000055555555523d in struct_bug () at main.c :16
```

Mieux compiler pour anticiper

```
$ gcc -g -Wall -Wextra main.c
main.c:16:17: warning: 'pStruct1' is used uninitialized [-Wuninitialized]
```

Solutions ✓

- ▶ utiliser malloc (puis free !) pour allouer le bon espace mémoire
- ▶ vérifier qu'un pointeur n'est pas NULL avant d'y accéder

Utilisation gdb 🐛 : comportement inattendu

un menu qui s'affiche deux fois au lieu d'une...

```
fonction int menu_bug()
```

Utilisation gdb 🕵️ : comportement inattendu

un menu qui s'affiche deux fois au lieu d'une...

```
fonction int menu_bug()
```

- ▶ l'instruction `fgets(saisie, 100, stdin);` s'exécute sans saisie de l'utilisateur!
- ▶ que contient `saisie`?
- ▶ que contient `*stdin._IO_read_ptr`?

solution ✅

vider le tampon d'entrée

1. Organisation du cours
2. Quizz de rentrée
3. Le débogage
4. Comprendre le comportement du programme
5. Le *memory check*

Qu'est-ce que le *garbage collection* (ou ramasse-miettes) ? 🇨🇦

« sous-système informatique de gestion automatique de la mémoire ...

1. détermine quels objets **ne peuvent plus** être utilisés par le programme
2. récupère l'espace utilisé par ces objets

... faisant généralement partie de l'environnement d'exécution **associé à un langage de programmation** particulier. »

...mais pas tous, notamment C et C++ ! Pourquoi à votre avis ?

source : [https://fr.wikipedia.org/wiki/Ramasse-miettes_\(informatique\)](https://fr.wikipedia.org/wiki/Ramasse-miettes_(informatique))

Fuite mémoire / *memory leak* 🗑️

Quel est le problème ?

- ▶ pas d'erreur ni de warning à la compilation
- ▶ pas de bug à l'exécution
- ▶ gdb ne signale pas la fuite

Risques

- ▶ ralentissement du système ⌚
- ▶ saturation de la mémoire 🚫

Fuite mémoire

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int fuite() {
6     static int* pIntNull;
7     static int* pInt;
8
9     pIntNull = malloc(sizeof(int));
10    pIntNull = NULL;
11
12    pInt = malloc(sizeof(int));
13    return 0;
14 }
15
16 int main()
17 {
18     fuite();
19     return 0;
20 }
```

Fuite mémoire 🧐 : l'outil valgrind (2002)

```
$ valgrind -leak-check=full ./a.out
==153317== HEAP SUMMARY:
==153317==      in use at exit: 8 bytes in 2 blocks
==153317==    total heap usage: 2 allocs, 0 frees, 8 bytes allocated
==153317==
==153317== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==153317== at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==153317== by 0x10915A: fuite (main.c :11) *
==153317== by 0x1091A1: main (main.c:19)
==153317==
==153317== LEAK SUMMARY:
==153317==    definitely lost: 4 bytes in 1 blocks
==153317==    indirectly lost: 0 bytes in 0 blocks
==153317==    possibly lost: 0 bytes in 0 blocks
==153317==    still reachable: 4 bytes in 1 blocks
==153317==    suppressed: 0 bytes in 0 blocks
```

* le numéro de ligne est accessible parce que la compilation a été réalisée avec l'option -g.

Fuite mémoire

`valgrind` : <https://doc.ubuntu-fr.org/valgrind>

```
$ valgrind [--options=valeur] ./a.out
```

Toutes les entrées / sorties mémoires sont analysées et vérifiées minutieusement en interceptant tous les appels à `malloc` et `free`

- ▶ exécute le programme
- ▶ dresse un diagnostic

Fuite mémoire

```
$ valgrind -leak-check=full -show-reachable=yes ./a.out
==154396== HEAP SUMMARY:
==154396==    in use at exit: 8 bytes in 2 blocks
==154396==    total heap usage: 2 allocs, 0 frees, 8 bytes allocated
==154396==
==154396== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==154396==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==154396==    by 0x10916B: fuite (main.c :12)
==154396==    by 0x1091A1: main (main.c:19)
==154396==
==154396== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==154396==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==154396==    by 0x10915A: fuite (main.c :9)
==154396==    by 0x1091A1: main (main.c:19)
==154396==
==154396== LEAK SUMMARY:
==154396==    definitely lost: 4 bytes in 1 blocks
==154396==    indirectly lost: 0 bytes in 0 blocks
==154396==    possibly lost: 0 bytes in 0 blocks
==154396==    still reachable: 4 bytes in 1 blocks
==154396==    suppressed: 0 bytes in 0 blocks
```

Fuite mémoire

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int fuite() {
6     static int* pIntNull;
7     static int* pInt;
8
9     pIntNull = malloc(sizeof(int));
10    pIntNull = NULL; // definitely lost
11
12    pInt = malloc(sizeof(int)); // still reachable
13    return 0;
14 }
15
16 int main()
17 {
18     fuite();
19     return 0;
20 }
```

Fuite mémoire

Solution

```
$ gcc -g -Wall main.c  
$ valgrind --leak-check=full --show-reachable=yes ./a.out
```

... et corriger le code.

Fuite mémoire

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int fuite() {
6     static int* pIntNull;
7     static int* pInt;
8
9     pIntNull = malloc(sizeof(int));
10    pIntNull = NULL;
11
12    pInt = malloc(sizeof(int));
13
14    free(pIntNull);
15    free(pInt);
16    return 0;
17 }
18
19 int main()
20 {
21     fuite();
22     return 0;
23 }
```

Fuite mémoire

```
$ valgrind -leak-check=full -show-reachable=yes ./a.out
==155231== HEAP SUMMARY:
==155231==      in use at exit: 4 bytes in 1 blocks
==155231==    total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==155231==
==155231== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==155231==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==155231==    by 0x10917A: fuite (main.c:9)
==155231==    by 0x1091DF: main (main.c:21)
==155231==
==155231== LEAK SUMMARY:
==155231==    definitely lost: 4 bytes in 1 blocks
==155231==    indirectly lost: 0 bytes in 0 blocks
==155231==    possibly lost: 0 bytes in 0 blocks
==155231==    still reachable: 0 bytes in 0 blocks
==155231==    suppressed: 0 bytes in 0 blocks
```

The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs.

(source C standard, 7.20.3.2/2 from ISO-IEC 9899 : lien)

Pointeur NULL

Dans quel cas un pointeur peut-il valoir NULL ?

Pointeur NULL

Dans quel cas un pointeur peut-il valoir NULL ?

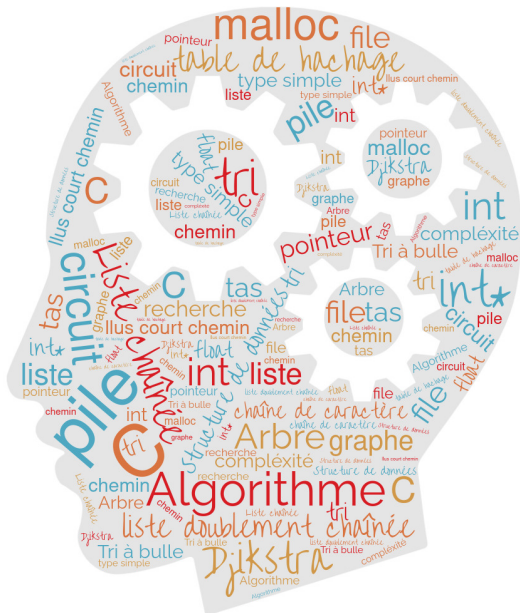
- ▶ initialisation à NULL : une variable non initialisée peut avoir un contenu *indéterminé* → il **faut** initialiser ses pointeurs ;
- ▶ échec d'allocation ;



Take away 🥤


1. quel est l'**outil de débuggage** le mieux adapté en fonction du type de bug ?
 - gcc : suivre pas à pas l'exécution du programme
 - valgrind : diagnostiquer les fuites mémoires
2. **pointeurs et bonnes pratiques**
 - initialiser à NULL si valeur inconnue
 - test valeur avant utilisation
 - appel à malloc pour l'initialisation
 - appel à free pour libérer l'espace mémoires

ASD 1 et 2



Organiser la connaissance

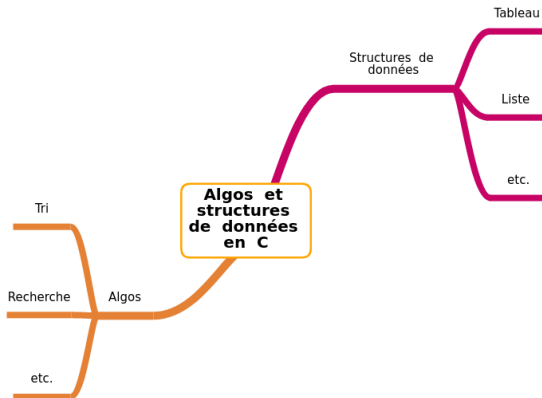
Travail à faire au cours du semestre **par groupes de 2**

Travail à rendre 

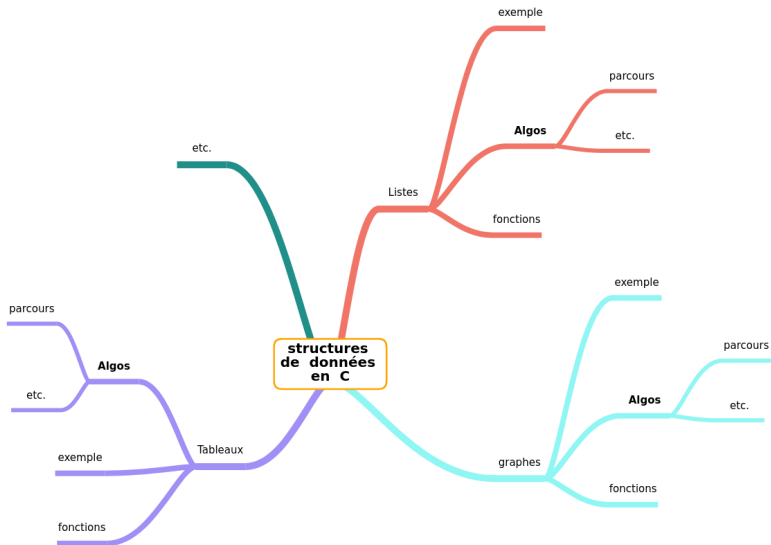
Création d'une (ou deux) *mindmap(s)* des structures de données et algorithmes en C

- ▶ création binômes (<https://lite.framacalc.org/1iq563nxhz-9yqc>)
- ▶ réfléchir à la conception sur papier (organisation générale, niveaux de profondeur max)
- ▶ choisir un outil de construction de mindmap, par ex :
 - outil en ligne de type <https://www.mindmaps.app/>
 - logiciel offline
<https://docs.freeplane.org/getting-started/getting-started.html>
 - markmap (Attention 6 niveaux de profondeur max!)
- ▶ création de la mindmap
- ▶ implémentation de la mindmap en C

Exemple conception 1



Exemple conception 2



Les règles de ce cours

- ▶ « *Peut-on utiliser youtube pour apprendre à programmer ?* » » 📺
 - Tri à bulle et danse hongroise
 - Tri rapide et danse hongroise

Sources et crédits

- ▶ Cours A. Canteaut (commandes de gdb)
 - ▶ Formats d'affichage dans gdb
 - ▶ Visu data struct et tris
 - ▶ Théorie des graphes
-
- ▶ Tri à bulle et danse hongroise
 - ▶ Tri rapide et danse hongroise
 - ▶ <https://geektionnerd.net/segfault/>
 - ▶ Liste emojis