Programmation Avancée

Cours 5 : `gdb` et `valgrind`

Simon Forest

25 février 2021

Problème

On a vu comment localiser une erreur d'exécution à coup de `fprintf(stderr,...)` .

```
f();
fprintf(stderr,"f OK\n");
g();
fprintf(stderr,"g OK\n");
h();
fprintf(stderr,"h OK\n");
// ...
```

Cela fonctionne mais peut être fastidieux.

Débogueur

On préfère habituellement utiliser un **débogueur** pour localiser les `Erreurs de segmentation` ainsi que d'autres erreurs.

Un **débogueur** est un programme qui permet d'inspecter l'exécution d'un programme. Il permet de voir en particulier la cause d'une `Erreur de segmentation`.

Pour les programmes en `C`, le débogueur standard est `gdb`.

Exemple

Un code en C qui produit une erreur :

```c
void f(int x)
{
  int r = rand();
  int* ptr = NULL;
  *ptr = 42;
}
void g()
{
  f(21);
}
int main()
{
  g();
}
```

Exemple

On le compile et on le lance avec `gdb` pour voir ce qu'il se passe :

```
gcc -g prog.c -o prog
gdb prog
```

Noter l'option `-g` à mettre quand on utilise `gdb`.

On obtient une nouvelle invite :

```
(gdb)
```

qui permet d'interagir avec `gdb`.

Exemple

On lance alors le programme avec la commande `run` :

```
(gdb) run
```

Et on obtient alors :

```
Starting program: /dossier/prog

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400545 in f (x=21) at prog.c:9
9         *ptr = 42;
```

On voit ainsi la position dans le code qui a déclenché l'erreur : `at prog.c:9` .

Exemple

Si l'on souhaitait lancer le programme avec des arguments, comme on l'aurait fait en terminal

```
./prog 3 MonFichier.txt Simon
```

on les aurait passés à `run` de la même façon :

```
(gdb) run 3 MonFichier.txt Simon
```

Exemple

On peut afficher la valeur des variables locales avec `print` (ou juste `p`) :

```
(gdb) p x
$1 = 21
(gdb) p r
$2 = 1804289383
(gdb) p ptr
$3 = (int *) 0x0
```

Exemple

On peut quitter le débogueur avec `quit` (ou juste `q`) :

```
(gdb) q
```

Points d'arrêt

Par défaut, `gdb` va s'arrêter uniquement s'il y a une erreur d'exécution.

Mais on peut ajouter des **points d'arrêt** (*break points*) pour s'arrêter à des points précis dans le code.

On utilise pour cela la commande `breakpoint` (ou simplement `b`).

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) breakpoint code.c:9
Breakpoint 1 at 0x400543: file code.c, line 9.
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) b code.c:9
Breakpoint 1 at 0x400543: file code.c, line 9.
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) b 5
Breakpoint 2 at 0x40052e: file code.c, line 5.
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) b foo
Breakpoint 3 at 0x40052e: file code.c, line 4.
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000400543 in main at code.c:9
2       breakpoint     keep y   0x000000400536 in foo at code.c:5
3       breakpoint     keep y   0x00000040052e in foo at code.c:4
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000400543 in main at code.c:9
2       breakpoint     keep y   0x000000400536 in foo at code.c:5
3       breakpoint     keep y   0x00000040052e in foo at code.c:4
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) run
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Points d'arrêt

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Reprendre l'exécution

Une fois l'exécution arrêtée par un point d'arrêt, on peut la reprendre de différente façons :

- `continue` (ou juste `c`) pour reprendre l'exécution jusqu'à un nouveau point d'arrêt ;
- `step` (ou juste `s`) pour exécuter une ligne de code source supplémentaire ;
- `next` (ou juste `n`) pour exécuter une ligne de code source supplémentaire **mais sans rentrer dans les fonctions**.

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
gdb ./prog
```

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) b 9
Breakpoint 1 at 0x400543: file code.c, line 9.
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;
6  }
7  int main()
8  {
9    foo();          ←
10   return 0;
11 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

Si on `continue` :

```
(gdb) c
Continuing.
[Inferior 1 (process 31239) exited normally]
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;
6  }
7  int main()
8  {
9    foo();          ←
10   return 0;
11 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();  ←
5    return r*r;
6  }
7  int main()
8  {
9    foo();
10   return 0;
11 }
```

Si on `step` :

```
(gdb) s
foo () at code.c:4
4           int r = rand();
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;
6  }
7  int main()
8  {
9    foo();          ←
10   return 0;
11 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;
6  }
7  int main()
8  {
9    foo();
10   return 0;        ←
11 }
```

Si on `next` :

```
(gdb) n
10          return 0;
```

`print`

Une fois le programme arrêté, on peut utiliser `print` pour afficher des variables présentes localement.

On peut plus généralement afficher une expression que l'on aurait pu écrire dans le code source :

```
(gdb) print x
(gdb) print (x + 3) % 7
(gdb) print foo(x,4)
...
```

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
gdb ./prog
```

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) b 9
Breakpoint 1 at 0x400543: file code.c, line 9.
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;
6  }
7  int main()
8  {
9    foo();          ←
10   return 0;
11 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:9
9           foo();
```

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();  ←
  return r*r;
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) s
foo () at code.c:4
4           int r = rand();
```

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;      ←
6  }
7  int main()
8  {
9    foo();
10   return 0;
11 }
```

```
(gdb) n
5           return r*r;
```

Attention à bien faire un `next` et pas un `step` ici, sinon on rentre dans le code de `rand` !

Exemple

```c
#include <stdlib.h>
int foo()
{
  int r = rand();
  return r*r;      ←
}
int main()
{
  foo();
  return 0;
}
```

```
(gdb) p r
$1 = 1804289383
```

On voit la valeur de `r` avec un `print`.

Exemple

```
1  #include <stdlib.h>
2  int foo()
3  {
4    int r = rand();
5    return r*r;      ←
6  }
7  int main()
8  {
9    foo();
10   return 0;
11 }
```

```
(gdb) p (r % 7) + 3
$2 = 4
```

On peut aussi demander de faire un calcul sur `r` avec `print`.

Autre exemple

```c
#include <stdlib.h>
#include <string.h>
typedef struct { int x,y; } point;
int main()
{
  point *ptr = malloc(sizeof(point));
  memset(ptr, 0x42, sizeof(point));
  return 0;
}
```

```
gdb ./prog
```

Autre exemple

```c
#include <stdlib.h>
#include <string.h>
typedef struct { int x,y; } point;
int main()
{
  point *ptr = malloc(sizeof(point));
  memset(ptr, 0x42, sizeof(point));
  return 0;
}
```

```
(gdb) b 6
Breakpoint 1 at 0x40056e: file code.c, line 6.
```

Autre exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  typedef struct { int x,y; } point;
4  int main()
5  {
6    point *ptr = malloc(sizeof(point)); ←
7    memset(ptr, 0x42, sizeof(point));
8    return 0;
9  }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:6
6           point *ptr = malloc(sizeof(point));
```

Autre exemple

```c
#include <stdlib.h>
#include <string.h>
typedef struct { int x,y; } point;
int main()
{
  point *ptr = malloc(sizeof(point)); ←
  memset(ptr, 0x42, sizeof(point));
  return 0;
}
```

```
(gdb) p ptr
$1 = (point *) 0x0
(gdb) p ptr->x
Cannot access memory at address 0x0
```

Autre exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  typedef struct { int x,y; } point;
4  int main()
5  {
6    point *ptr = malloc(sizeof(point));
7    memset(ptr, 0x42, sizeof(point));   ←
8    return 0;
9  }
```

```
(gdb) n
7           memset(ptr, 0x42, sizeof(point));
(gdb) p ptr
$2 = (point *) 0x602010
```

Autre exemple

```c
#include <stdlib.h>
#include <string.h>
typedef struct { int x,y; } point;
int main()
{
  point *ptr = malloc(sizeof(point));
  memset(ptr, 0x42, sizeof(point));   ←
  return 0;
}
```

```
(gdb) p ptr->x
$3 = 0
(gdb) p ptr->y
$4 = 0
```

Autre exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  typedef struct { int x,y; } point;
4  int main()
5  {
6    point *ptr = malloc(sizeof(point));
7    memset(ptr, 0x42, sizeof(point));
8    return 0;                          ←
9  }
```

```
(gdb) n
8           return 0;
(gdb) p ptr->x
$5 = 1111638594
(gdb) p ptr->y
$6 = 1111638594
```

Autre exemple

```c
#include <stdlib.h>
#include <string.h>
typedef struct { int x,y; } point;
int main()
{
  point *ptr = malloc(sizeof(point));
  memset(ptr, 0x42, sizeof(point));
  return 0;                              ←
}
```

```
(gdb) p/x ptr->x
$7 = 0x42424242
(gdb) p/x ptr->y
$8 = 0x42424242
```

`p/x` : version de `print` qui affiche en hexadécimal.

`watchpoint`s

Il est possible de surveiller une variable et de s'arrêter à chaque changement de celle-ci grâce à la commande `watchpoint`.

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++)
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
gdb ./prog
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++)
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
(gdb) b main
Breakpoint 1 at 0x40052e: file code.c, line 4.
```

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;                ←
5    int r;
6    for(int i = 0; i < 10; i++)
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, main () at code.c:4
4           int pile = 0;
```

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;                ←
5    int r;
6    for(int i = 0; i < 10; i++)
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

```
(gdb) wa pile
Hardware watchpoint 2: pile
```

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;
5    int r;                          ←
6    for(int i = 0; i < 10; i++)
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

```
(gdb) c
Hardware watchpoint 2: pile
Old value = 32767
New value = 0
main () at code.c:5
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++) ←
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
(gdb) c
Hardware watchpoint 2: pile
Old value = 0
New value = 1
main () at code.c:6
```

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;
5    int r;
6    for(int i = 0; i < 10; i++) ←
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

L'incrémentation se fait bien ligne 10, mais, comme on est à la fin de la boucle, le watchpoint nous arrête à la prochaine instuction qui est ligne 6.

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;
5    int r;
6    for(int i = 0; i < 10; i++) ←
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

```
(gdb) p i
$1 = 1
(gdb) p r
$2 = 1894289383
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++) ←
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
(gdb) c
Hardware watchpoint 2: pile
Old value = 1
New value = 2
main () at code.c:6
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++) ←
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
(gdb) p i
$1 = 4
(gdb) p r
$2 = 1681692777
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++) ←
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

```
(gdb) c
Hardware watchpoint 2: pile
Old value = 2
New value = 3
main () at code.c:6
```

Exemple

```
1  #include <stdlib.h>
2  int main()
3  {
4    int pile = 0;
5    int r;
6    for(int i = 0; i < 10; i++) ←
7    {
8      r = rand();
9      if(r % 2)
10       pile++;
11   }
12   return 0;
13 }
```

```
(gdb) p i
$1 = 5
(gdb) p r
$2 = 1714636915
```

Exemple

```c
#include <stdlib.h>
int main()
{
  int pile = 0;
  int r;
  for(int i = 0; i < 10; i++) ←
  {
    r = rand();
    if(r % 2)
      pile++;
  }
  return 0;
}
```

etc.

`backtrace`

On peut voir la pile des fonctions appelées avec `backtrace` (ou juste `bt`).

Utile pour voir comment a été appelée une fonction, par qui et avec quels arguments.

Utile aussi pour déboguer les fonctions récursives.

Exemple

```c
int foo(int x)
{
  if(x == 0)
    return 0;
  return 1 + foo(x/2);
}
int main()
{
  foo(15);
  return 0;
}
```

```
gdb ./prog
```

Exemple

```c
int foo(int x)
{
  if(x == 0)
    return 0;
  return 1 + foo(x/2);
}
int main()
{
  foo(15);
  return 0;
}
```

```
(gdb) b main
breakpoint 1 at 0x4004e7: file code.c, line 4.
```

Exemple

```
1  int foo(int x)
2  {
3    if(x == 0)
4      return 0;          ←
5    return 1 + foo(x/2);
6  }
7  int main()
8  {
9    foo(15);
10   return 0;
11 }
```

```
(gdb) r
starting program: /dossier_code/prog

breakpoint 1, foo (x=0) at code.c:4
4             return 0;
```

Exemple

```
1  int foo(int x)
2  {
3    if(x == 0)
4      return 0;          ←
5    return 1 + foo(x/2);
6  }
7  int main()
8  {
9    foo(15);
10   return 0;
11 }
```

```
(gdb) bt
#0  foo (x=0) at code.c:4
#1  0x0000000000400501 in foo (x=1) at code.c:5
#2  0x0000000000400501 in foo (x=3) at code.c:5
#3  0x0000000000400501 in foo (x=7) at code.c:5
#4  0x0000000000400501 in foo (x=15) at code.c:5
#5  0x0000000000400514 in main () at code.c:9
```

`bt full`

On peut avoir une vision encore plus complète de la pile avec `bt full`.

Cette commande affiche **tout** le contenu de la pile à l'endroit courant.

Exemple

```c
#include <stdlib.h>
int foo(int x)
{
  int r = rand();
  if(x == 0)
    return 0;
  return r + foo(x/2);
}
int main()
{
  foo(15);
  return 0;
}
```

```
gcc -g code.c -o prog
gdb ./prog
```

Exemple

```c
#include <stdlib.h>
int foo(int x)
{
  int r = rand();
  if(x == 0)
    return 0;
  return r + foo(x/2);
}
int main()
{
  foo(15);
  return 0;
}
```

```
(gdb) b 6
Breakpoint 1 at 0x40053f:
file code.c, line 6.
```

Exemple

```
1  #include <stdlib.h>
2  int foo(int x)
3  {
4    int r = rand();
5    if(x == 0)
6      return 0;              ←
7    return r + foo(x/2);
8  }
9  int main()
10 {
11   foo(15);
12   return 0;
13 }
```

```
(gdb) r
Starting program: /dossier_code/prog

Breakpoint 1, foo (x=0) at code.c:6
6             return 0;
```

Exemple

```
1  #include <stdlib.h>
2  int foo(int x)
3  {
4    int r = rand();
5    if(x == 0)
6      return 0;              ←
7    return r + foo(x/2);
8  }
9  int main()
10 {
11   foo(15);
12   return 0;
13 }
```

```
(gdb) bt full
#0  foo (x=0) at code.c:6
        r = 1957747793
#1  0x0000000000400559 in foo (x=1) at code.c:7
        r = 1714636915
#2  0x0000000000400559 in foo (x=3) at code.c:7
        r = 1681692777
#3  0x0000000000400559 in foo (x=7) at code.c:7
        r = 846930886
#4  0x0000000000400559 in foo (x=15) at code.c:7
        r = 1804289383
#5  0x0000000000400570 in main () at code.c:11
```

D'autres commandes

- `break 6 if x >= 3` : ajoute un point d'arrêt conditionnel (l'arrêt se fait si `x >= 3`) ;
- `display x` : affiche la valeur de la variable `x` à chaque arrêt ;
- `finish` (ou `fin`) : continue l'exécution jusqu'à la fin de la fonction courante ;
- `info breakpoints` (ou juste `i b`) : affiche la liste des points d'arrêt actifs ;
- `disable 2` (ou juste `d 2`) : désactive le point d'arrêt numéro 2 ;
- `info args` : affiche la valeur des arguments de la fonction courante ;
- `info locals` : affiche la valeur des variables locales.

D'autres commandes

Aussi, comment avoir de l'aide sur `gdb` :

- `help` : affiche le sommaire de l'aide de `gdb` ;
- `help display` : affiche l'aide pour la commande `display` ;
- `help info locals` : affiche l'aide pour la commande `info locals` .

Ne pas oublier que...

... pour profiter de toutes les fonctionnalités de `gdb`, il faut compiler votre code avec l'option `-g` :

```
gcc -g code.c -o prog
```

Problème

`gdb` est utile pour trouver la cause de certaines erreurs, notamment les erreurs de segmentation.

Cependant, il ne permet pas de détecter efficacement les erreurs de corruption mémoire.

En effet, ces erreurs arrêtent le programme là où un problème a été détecté, mais pas là où il est apparu.

Exemple

```c
void f(){...}
int main()
{
  int *ptr = malloc(sizeof(int));
  f();
  free(ptr);
  return 0;
}
```

```
gcc code.c -o prog
./prog
*** Error in `./prog': munmap_chunk(): invalid pointer ... ***
```

Ouch... est-ce que `gdb` peut nous aider ?

Exemple

```c
void f(){...}
int main()
{
  int *ptr = malloc(sizeof(int));
  f();
  free(ptr);
  return 0;
}
```

```
gcc -g code.c -o prog
gdb ./prog
(gdb) r
Program received signal SIGABRT, Aborted.
0x00007ffff7a42438 in __GI_raise (sig=sig@entry=6)
```

Ouch... est-ce que `gdb` peut nous aider ?

Exemple

```
1  void f(){...}
2  int main()
3  {
4    int *ptr = malloc(sizeof(int));
5    f();
6    free(ptr);                    ←
7    return 0;
8  }
```

On utilise `bt` pour savoir où on est exactement :

```
(gdb) bt
#0  0x00007ffff7a42438 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
...
#6  0x00000000004005c7 in main () at code.c:6
```

Le problème est ligne 6. Mais on fait juste un `free` à cet endroit !

Exemple

```
1  void f(){...}
2  int main()
3  {
4    int *ptr = malloc(sizeof(int));
5    f();
6    free(ptr);
7    return 0;
8  }
```

En fait, le problème est initié dans `f`.

`gdb` peut juste nous dire où le programme a été arrêté à cause de l'erreur (ici, ligne 6). Cependant, ce n'est pas le code à cette position qu'il faut changer, mais bien celui de `f`.

Mais comment savoir où dans `f` ?

Rappel du tas

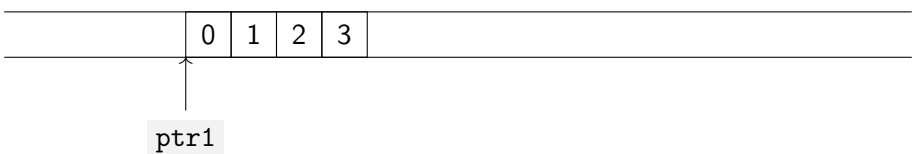
```c
char *ptr1 = malloc(4); // sizeof(char) == 1, on l'omet ici
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

Tas :

Rappel du tas

```c
char *ptr1 = malloc(4); // sizeof(char) == 1, on l'omet ici
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

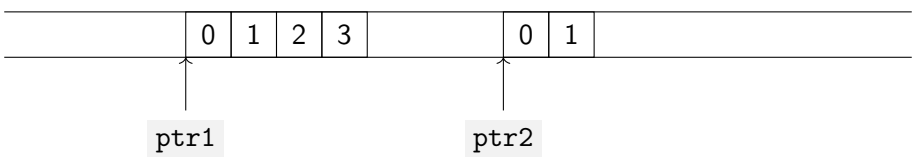
Tas :



Rappel du tas

```c
char *ptr1 = malloc(4); // sizeof(char) == 1, on l'omet ici
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

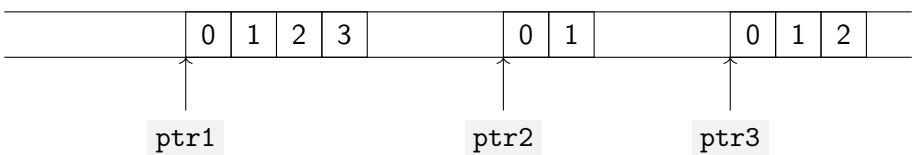
Tas :



Rappel du tas

```c
char *ptr1 = malloc(4); // sizeof(char) == 1, on l'omet ici
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

Tas :



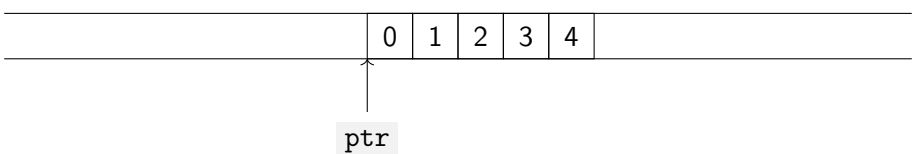
Détails supplémentaires

Les blocs de mémoire alloués par `malloc` sont libérés par `free`.

Comment `free` connait la taille de la zone à libérer ?

```
char *ptr = malloc(5);
// ...
free(ptr);
```

Tas :



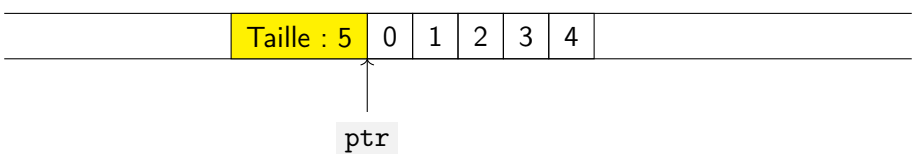
Détails supplémentaires

Les blocs de mémoire alloués par `malloc` sont libérés par `free`.

Comment `free` connait la taille de la zone à libérer ?

```
char *ptr = malloc(5);
// ...
free(ptr);
```

Tas :



Elle est en fait stockée juste avant le pointeur renvoyé.

Corruption du tas

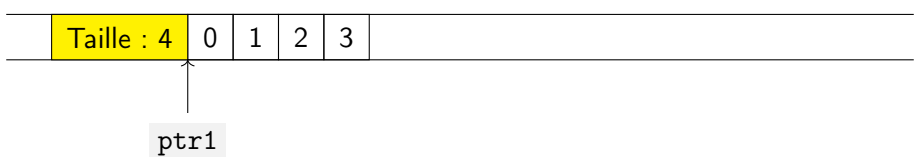
```
char *ptr1 = malloc(4);
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

Tas :

Corruption du tas

```
char *ptr1 = malloc(4);
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

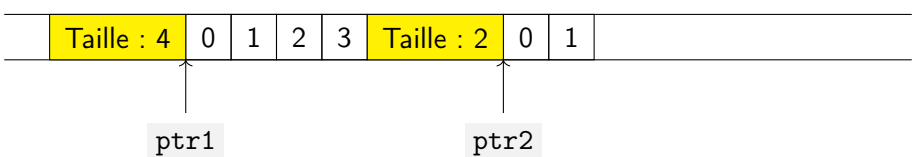
Tas :



Corruption du tas

```
char *ptr1 = malloc(4);
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

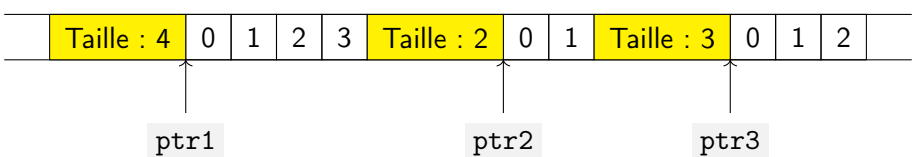
Tas :



Corruption du tas

```
char *ptr1 = malloc(4);
char *ptr2 = malloc(2);
char *ptr3 = malloc(3);
```

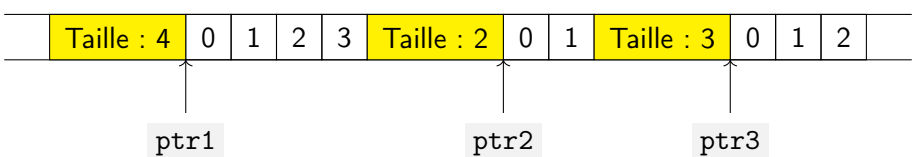
Tas :



Corruption du tas

```
for(int i = 0; i < 8; i++) // Oups...
{
  ptr1[i] = 0xff;
}
```

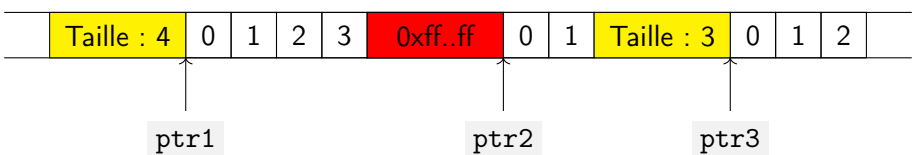
Tas :



Corruption du tas

```
for(int i = 0; i < 8; i++) // Oups...
{
  ptr1[i] = 0xff;
}
```

Tas :

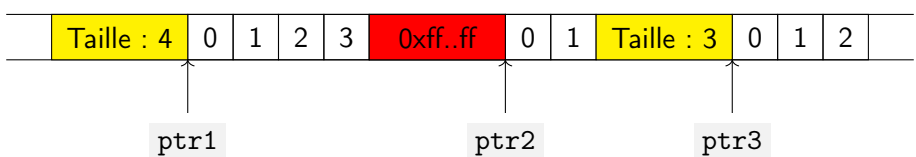


Corruption du tas

```
// ...
free(ptr2);
```

```
*** Error in `./prog': munmap_chunk(): invalid pointer ... ***
```

Tas :



Bilan

Ceci était une version simplifiée des informations stockées par `malloc`.

En vrai, `free` stocke aussi de l'information.

Il existe d'autres erreurs que celle présentée, mais elles ont presque toutes la même origine : **écrire à un endroit non autorisé**.

Dans tous les cas, il faut donc **absolument** éviter les accès en dehors des zones autorisées.

`valgrind`

On peut trouver les erreurs de corruption mémoire avec `valgrind`.

Son utilisation est assez simple :

```
gcc -g code.c -o prog
valgrind ./prog
```

Sont ensuite listées tous les accès mémoire invalides repérés par `valgrind`.

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[5] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
gcc -g code.c -o prog
valgrind ./prog
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[5] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
Invalid write of size 1
   at 0x400544: f (code.c:5)
   by 0x400557: main (code.c:9)
 Address 0x5204045 is 0 bytes after a block of size 5 alloc'd
   at 0x4C2DB8F: malloc (in ...)
   by 0x400537: f (code.c:4)
   by 0x400557: main (code.c:9)
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);          ←
  ptr[5] = 3;                     ←
}
int main()
{
  f();
  return 0;
}
```

```
Invalid write of size 1
   at 0x400544: f (code.c:5)      ←
   by 0x400557: main (code.c:9)
 Address 0x5204045 is 0 bytes after a block of size 5 alloc'd
   at 0x4C2DB8F: malloc (in ...)
   by 0x400537: f (code.c:4)      ←
   by 0x400557: main (code.c:9)
```

Autres fonctionnalités

`valgrind` permet de détecter d'autres problèmes, comme :

- l'utilisation de variables non initialisées ;
- les fuites de mémoire.

Exemple

```c
#include <stdio.h>
int main()
{
  int x;
  printf("%d",x);
  return 0;
}
```

```
gcc -g code.c -o prog
valgrind ./prog
```

Exemple

```c
#include <stdio.h>
int main()
{
  int x;
  printf("%d",x);
  return 0;
}
```

```
Conditional jump or move depends on uninitialised value(s)
   at 0x4E87B93: vfprintf (vfprintf.c:1631)
   by 0x4E8F8A8: printf (printf.c:33)
   by 0x400541: main (code.c:5)
```

Exemple

```c
#include <stdio.h>
int main()
{
  int x;
  printf("%d",x);                    ←
  return 0;
}
```

```
Conditional jump or move depends on uninitialised value(s)
   at 0x4E87B93: vfprintf (vfprintf.c:1631)
   by 0x4E8F8A8: printf (printf.c:33)
   by 0x400541: main (code.c:5)      ←
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
gcc -g code.c -o prog
valgrind ./prog
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
HEAP SUMMARY:
    in use at exit: 5 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 5 bytes allocated
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
HEAP SUMMARY:
    in use at exit: 5 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 5 bytes allocated
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
LEAK SUMMARY:
   definitely lost: 5 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
...
Rerun with --leak-check=full to see details of leaked memory
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
valgrind --leak-check=full ./prog
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  ptr[2] = 3;
}
int main()
{
  f();
  return 0;
}
```

```
5 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C2DB8F: malloc (in ...)
   by 0x400537: f (code.c:4)
   by 0x400557: main (code.c:9)
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  free(ptr);              // on corrige ici
}
int main()
{
  f();
  return 0;
}
```

```
gcc -g code.c -o prog
valgrind ./prog
```

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  free(ptr);              // on corrige ici
}
int main()
{
  f();
  return 0;
}
```

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 1 allocs, 1 frees, 5 bytes allocated

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

Exemple

```c
#include <stdlib.h>
void f()
{
  char *ptr = malloc(5);
  free(ptr);              // on corrige ici
}
int main()
{
  f();
  return 0;
}
```

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 1 allocs, 1 frees, 5 bytes allocated

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

Rappel : appel par valeur

En `C`, les fonctions sont appelées **par valeur** : les arguments qu'elles prennent sont **copiés** sur la pile.

Ainsi, si une fonction modifie ses arguments, cela n'a pas de conséquences sur les variables de l'appelant.

Si l'on veut qu'une fonction puisse modifier des données de l'appelant, il faut passer l'**adresse** de ces données.

Exemples

```c
void f(int x)
{
  x = 1;
}
int main()
{
  int x = 0;
  f(x);
  printf("x = %d",x);
}
```

Qu'affiche `./prog` ?

Exemples

```c
void f(int x)
{
  x = 1;
}
int main()
{
  int x = 0;
  f(x);
  printf("x = %d",x);
}
```

Qu'affiche `./prog` ?

```
x = 0
```

Exemples

```
void f(int* ptr)
{
  ptr = malloc(sizeof(int));
}
int main()
{
  int *ptr = NULL;
  f(ptr);
  printf("ptr = %p",ptr);
}
```

Qu'affiche `./prog` ?

Exemples

```
void f(int* ptr)
{
  ptr = malloc(sizeof(int));
}
int main()
{
  int *ptr = NULL;
  f(ptr);
  printf("ptr = %p",ptr);
}
```

Qu'affiche `./prog` ?

```
ptr = NULL
```

Exemples

```
void f(int* ptr)
{
  *ptr = 1;
}
int main()
{
  int x = 0;
  f(&x);
  printf("x = %d",x);
}
```

Qu'affiche `./prog` ?

Exemples

```
void f(int* ptr)
{
  *ptr = 1;
}
int main()
{
  int x = 0;
  f(&x);
  printf("x = %d",x);
}
```

Qu'affiche `./prog` ?

```
x = 1
```

Exemples

```c
typedef struct { char *nom; int age; } personne;
void f(personne p)
{
  strcpy(p.nom,"Simon");
  p.age = 28;
}
int main()
{
  personne p = {NULL,0};
  p.nom = malloc(100 * sizeof(char));
  f(p);
  printf("%s : %d\n",p.nom,p.age);
}
```

Qu'affiche `./prog` ?

Exemples

```c
typedef struct { char *nom; int age; } personne;
void f(personne p)
{
  strcpy(p.nom,"Simon");
  p.age = 28;
}
int main()
{
  personne p = {NULL,0};
  p.nom = malloc(100 * sizeof(char));
  f(p);
  printf("%s : %d\n",p.nom,p.age);
}
```

Qu'affiche `./prog` ?

```
Simon : 0
```

array_init

En particulier, voici comment **ne pas** écrire `array_init` (TP n°4) :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  ap = malloc(sizeof(array));
  ap->ptr = malloc(n * sizeof(int));
  ap->size = n;
}
```

array_init

En particulier, voici comment **ne pas** écrire `array_init` (TP n°4) :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  ap = malloc(sizeof(array));
  ap->ptr = malloc(n * sizeof(int));
  ap->size = n;
}
```

Non, car un appel `array_init(&t,5)` sur `array t` ne change ni `t.ptr` ni `t.size`.

array_init

En particulier, voici comment **ne pas** écrire `array_init` (TP n°4) :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  array tab;
  tab.size = n;
  tab.ptr =  malloc(n * sizeof(int));
  ap = &tab;
}
```

array_init

En particulier, voici comment **ne pas** écrire `array_init` (TP n°4) :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  array tab;
  tab.size = n;
  tab.ptr =  malloc(n * sizeof(int));
  ap = &tab;
}
```

Non, car un appel `array_init(&t,5)` sur `array t` ne change ni `t.ptr` ni `t.size`.

array_init

Solution :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  ap->ptr = malloc(n * sizeof(int));
  ap->size = n;
}
```

array_init

Solution :

```
typedef struct array{
  int* ptr;
  int size;
} array;

void array_init(array *ap, int n)
{
  ap->ptr = malloc(n * sizeof(int));
  ap->size = n;
}
```

Oui, car ici on modifie les données **pointées par** `ap` .

Un appel à `array_init(&t,5)` sur `array t` modifie bien `t.ptr` et `t.size` .