CM 13 - OUTILS DE DEBUGGING

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

BUG OU BOGUE

▷ En informatique, un bug ou bogue est un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement

▷ A l'été 1945, alors que le prototype dernier cri des ordinateurs appelé Mark 1 (5 tonnes, 15m L x 0.5m P x 2.50m H) qui interprétait des kilomètres de bandes préalablement perforées par les programmeurs pour additionner deux nombres de 23 chiffres en 3 dixièmes de secondes, il cessa de fonctionner...

▷ Une mite aux ailes grillées fut extraite d'un des relais électro-mécanique ; le bug informatique était né !

DEBUGGING

▷ Le processus de debugging correspond à l'analyse des bugs d'un programme

▷ Un debugger est un logiciel qui aide le développeur dans ce processus

▷ Il permet

- ▷ d'exécuter le programme pas-à-pas
- ▷ d'afficher la valeur des variables à tout moment
- ▷ de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme

DEBUGGING

▷ Le debugger a besoin d'informations spécifiques et supplémentaires pour s'exécuter (e.g., les symboles)

▷ Il faut donc le préciser à la compilation à l'aide de l'option `-g`

▷ C'est une étape nécessaire et essentielle

▷ Il existe plein de debugger

▷ Nous nous focaliserons sur l'interface de commande gdb, acronyme de GNU DeBugger et une des interfaces graphiques le gérant ddd, acronyme de Data Display Debugger

GDB

UTILISATION BASIQUE

▷ Après une compilation incluant le mode debugging

```
$>gcc -std=c99 -g bug.c -o bug
```

▷ Il suffit de lancer le debugger en lui passant notre exécutable

```
$>gdb bug
```

▷ Dans l'interface de **gdb**, on peut lancer le programme avec la commande `run` en lui fournissant les éventuels arguments nécessaires à notre programme

```
(gdb) run arg1 "arg2" ...
```

UTILISATION BASIQUE

▷ Le programme analysé peut être redémarré en utilisant la commande `kill` pour stopper l'exécution actuelle et relancé à l'aide de la commande **`run`**

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run ...
```

▷ Quitter le debugger s'effectue à l'aide de la commande **`quit`**

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$>
```

UTILISATION BASIQUE

▷ Une aide sur chaque commande du debugger peut être obtenue à l'aide de la commande `help` ou `help <command>`

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping it
user-defined -- User-defined commands
```

Contrôle de l'exécution

▷ Il est possible de stopper l'exécution du programme en utilisant la combinaison de touches **Ctrl+C**

```
(gdb) run
Starting Program: /home/gblin/a.out
^C
Program received signal SIGINT, Interrupt.
main () at loop.c:12
12   while(1){
(gdb)
```

▷ Il est également possible de relancer l'exécution à l'aide de la commande **continue**

Contrôle de l'exécution

▷ Pour visualiser où s'est arrêté le programme, on peut lister les lignes du code source autour du point d'exécution actuel à l'aide de la commande **list** (par défaut 10 lignes)

```
(gdb) list
7       printf("stupid : %d\n",cpt);
8       foo(cpt-1);
9    }
10   int main(void){
11      int i=1;
12      while(1){
13          foo(i);
14          i=i%3+1;
15      }
16      return EXIT_SUCCESS;
(gdb)
```

CONTRÔLE DE L'EXÉCUTION

▷ Deux commandes permettent d'exécuter pas à pas le programme: la commande **next** passe à la prochaine instruction de la fonction courante alors que **step** passe à la prochaine instruction du programme

```
(gdb) next
13      foo(i);
(gdb) next
stupid stuff : 2
stupid stuff : 1
14      i=i%3+1;
(gdb) next
12    while(1){
(gdb) next
13      foo(i);
(gdb) step
foo (cpt=3) at loop.c:4
4     if(cpt==0){
(gdb)
```

```
#include <stdlib.h>
#include <stdio.h>
void foo(int cpt){
  if(cpt==0){
    return;
  }
  printf("stupid : %d\n",cpt);
  foo(cpt-1);
}
int main(void){
  int i=1;
  while(1){
    foo(i);
    i=i%3+1;
  }
  return EXIT_SUCCESS;
}
```

CONTRÔLE DE L'EXÉCUTION

▷ Il est possible de consulter la valeur d'une variable à l'aide de la commande **print** suivie du nom de la variable d'intérêt

```
(gdb) step
foo (cpt=3) at loop.c:4
4       if(cpt==0){
(gdb) print cpt
$1 = 3
(gdb) print truc
No symbol "truc" in current context.
(gdb)
```

▷ Il est possible de modifier la valeur d'une variable via la commande **set var**

```
(gdb) set var cpt = 3
(gdb) print cpt
$2 = 3
(gdb)
```

CONTRÔLE DE L'EXÉCUTION

▷ Il est possible d'appeler n'importe quelle fonction utilisée dans le programme ainsi que celles de la librairie standard via la commande **call**

```
(gdb) call foo(1)
stupid stuff : 1
(gdb) call strlen("foo")
$6 = 3
(gdb)
```

▷ La commande **finish** permet d'exécuter la fin de la fonction

```
(gdb) step
foo (cpt=1) at loop.c:4
4       if(cpt==0){
(gdb) finish
Run till exit from #0  foo (cpt=1) at loop.c:4
stupid stuff : 1
main () at loop.c:13
13        foo(i);
(gdb)
```

EXAMEN DE LA PILE D'APPEL

▷ La commande **backtrace** permet de voir l'état de la pile d'appel

```
(gdb) backtrace
#0  foo (cpt=0) at loop.c:4
#1  0x00001f33 in foo (cpt=1) at loop.c:8
#2  0x00001f33 in foo (cpt=2) at loop.c:8
#3  0x00001f5f in main () at loop.c:13
(gdb)
```

▷ La commande **frame** permet, à l'aide du nombre situé à gauche, d'examiner chaque segment de pile correspondant à un appel de fonction et d'en afficher l'état des variables locales avec la commande **info locals**

```
(gdb) frame 3
#3  0x00001f5f in main () at loop.c:13
13      foo(i);
(gdb) info locals
i = 1
(gdb)
```

GIVE ME A BREAK !

▷ La commande **break** permet de spécifier au debugger de s'arrêter à chaque passage à une ligne spécifique d'un code source (plus précis que **Ctrl+C**)

▷ On peut également en ajouter pour une fonction particulière, tous les lister (**list breakpoints**) et en supprimer (**delete <num-bkpts>**)

```
(gdb) break loop.c:12
Breakpoint 1 at 0x1f7b: file loop.c, line 12.
(gdb) break foo
Breakpoint 2 at 0x1ef2: file loop.c, line 4.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00001f7b in main at loop.c:12
2       breakpoint     keep y   0x00001ef2 in foo at loop.c:4
(gdb) delete 2
```

BIG BROTHER

▷ Il est possible de surveiller une variable - via les **watchpoint** - en lecture (**rwatch**), écriture (**watch**) ou les deux (**awatch**) qui arrêtent le programme juste après l'exécution de l'une des opérations sur la variable surveillée

```
(gdb) awatch i
Hardware access (read/write) watchpoint 6: i
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 6: i
Old value = 1
New value = 2
main () at loop.c:12
12    while(1){
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 6: i
Value = 2
0x00001f57 in main () at loop.c:13
13      foo(i);
```

ALZHEIMER

▷ Finalement, il peut être intéressant de visualiser l'état de la mémoire à partir d'une adresse via la commande **x** (pour **eX-amine**)

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
```

ALZHEIMER

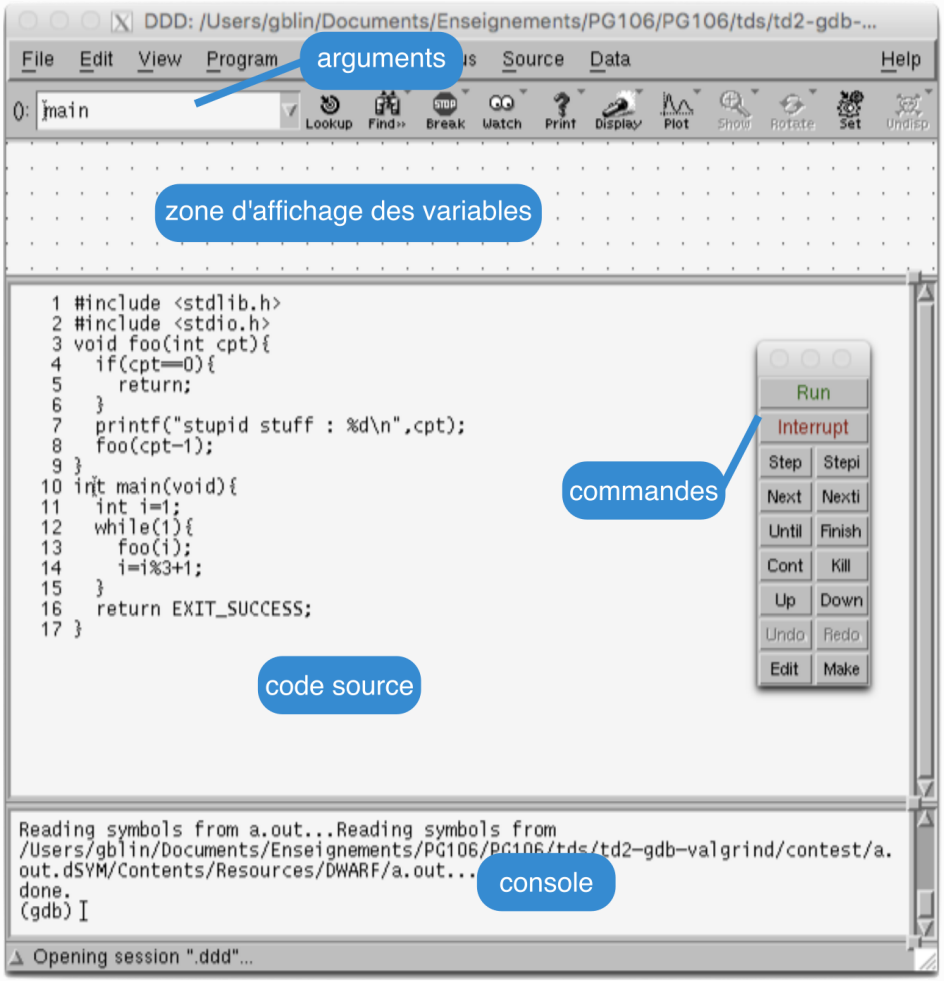
▷ Finalement, il peut être intéressant de visualiser l'état de la mémoire à partir d'une adresse via la commande **x** (pour **eX-amine**)

```
(gdb) set var i = 0xF000293A
(gdb) x/xw &i
0xbffffa60: 0xf000293a
(gdb) x/dw &i
0xbffffa60: -268424902
(gdb) x/uw &i
0xbffffa60: 4026542394
(gdb) x/tw &i
0xbffffa60: 11110000000000000010100100111010
(gdb) x/cb &i
0xbffffa60: 58 ':'
(gdb) x/sb &i
0xbffffa60: ":)"
```

DATA DISPLAY DEBUGGER

UNE INTERFACE PLUS CONVIVIALE

▷ L'outil Data Display Debugger permet de manipuler **gdb** dans un environnement plus convivial



EXEMPLES

```c
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

int main(void){
  char c;
  c = fgetc(stdin);
  while(c != EOF){
    if(isalnum(c)){
      printf("%c", c);
    }else{
      c = fgetc(stdin);
    }
  }
  return EXIT_SUCCESS;
}
```

▷ Infinite loop

EXEMPLES

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char *buf;
  buf = malloc(1<<31);
  fgets(buf, 1024, stdin);
  printf("%s\n", buf);
  return EXIT_SUCCESS;
}
```

▷ SegFault

VALGRIND

PROFILING

▷ L'outil Memcheck de la suite d'outils Valgrind permet de détecter la plupart des erreurs relatives à la gestion de la mémoire

▷ A l'instar de **gdb**, **Memcheck** a besoin d'informations de debuggage et nécessite une compilation avec l'option **-g** mais également sans optimisation de préférence (i.e., **-O0**)

▷ Lancer l'outil sur votre programme s'effectue à l'aide de la commande suivante

```
valgrind --leak-check=yes <executable> <args>
```

▷ L'option **–leak-check** demande l'exécution du détecteur de fuite mémoire

▷ ⚠ L'exécution du programme est fortement ralentie (e.g. 20 à 30 fois)

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
==1437== Memcheck, a memory error detector
==1437== Copyright (C) 2002-2015, and GNU GPLd, by Julian Seward et al.
==1437== Using Valgrind-3.11.0 and LibVEX; rerun with -h for ...
==1437== Command: ./a.out
==1437==
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

- ⚠ Beaucoup d'information !

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

- 1437 correspond au processus id

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

- `...Invalid write...` indique le type d'erreur; ici une écriture non autorisée de 4 octets

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Les lignes suivantes présentent une backtrace suivie des adresses mémoires impliquées

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Ici on voit que l'erreur se produit juste après une allocation mémoire effectuée par **malloc** à la ligne 3 de notre fichier source

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== Invalid write of size 4
==1437==    at 0x100000F4C: f (valgrind.c:4)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==  Address 0x100a712d8 is 0 bytes after a block of size 40 alloc'd
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Il faut fixer les erreurs dans l'ordre d'apparition – ici l'accès à `x[10]`

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== 40 bytes in 1 blocks are definitely lost in loss record 19 of 61
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
==1437==
==1437== LEAK SUMMARY:
==1437==    definitely lost: 40 bytes in 1 blocks
==1437==    indirectly lost: 0 bytes in 0 blocks
==1437==      possibly lost: 0 bytes in 0 blocks
==1437==    still reachable: 0 bytes in 0 blocks
==1437==         suppressed: 22,140 bytes in 186 blocks
...
==1437== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 16 from 16)
```

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== 40 bytes in 1 blocks are definitely lost in loss record 19 of 61
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Une backtrace fournit la localisation de l'allocation de la mémoire concernée

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== 40 bytes in 1 blocks are definitely lost in loss record 19 of 61
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Ici on voit que l'allocation a été effectuée par **malloc** à la ligne 3 de notre fichier source

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes ./a.out
...
==1437== 40 bytes in 1 blocks are definitely lost in loss record 19 of 61
==1437==    at 0x100008EBB: malloc (in vgpreload_memcheck.so)
==1437==    by 0x100000F43: f (valgrind.c:3)
==1437==    by 0x100000F73: main (valgrind.c:7)
...
```

▷ Il y a plusieurs types de fuites mémoires dont les plus importants sont ”definitely” et ”possibly lost” qu’il faut résoudre

▷ Ici c’est le tableau de 40 octets qui n’est pas libéré

LIRE ET COMPRENDRE MEMCHECK

```c
#include <stdlib.h>
void f(void){
   int i;
   int* x = malloc(10 * sizeof(int));
   x[i] = 0;
   free(x);
}
```

```c
int main(void){
   f();
   return EXIT_SUCCESS;
}
```

```
$> valgrind --leak-check=yes --track-origins=yes ./a.out
...
==1792== Use of uninitialised value of size 8
==1792==    at 0x100000F20: f (valgrind.c:5)
==1792==    by 0x100000F63: main (valgrind.c:10)
==1792==  Uninitialised value was created by a stack allocation
==1792==    at 0x100000F00: f (valgrind.c:2)
...
```

▷ Avec l'option **–track-origins=yes** Memcheck peut également détecter l'utilisation de variables non initialisées et rechercher leur localisation dans le fichier source

DOGGY BAG

TO TAKE AWAY ...

▷ La gestion de la mémoire par le programmeur fournit un grand pouvoir

▷ Mais, comme le dit oncle Ben, ”Un grand pouvoir implique de grandes responsabilités”

▷ Les programmes de debugging sont performant et très utiles pour des erreurs non facilement détectables depuis le fichier source

▷ Préparez-vous au GDB-contest à venir !

Questions?