DEBUGGING C GDB

За да използва дебъгер, програмите трябва да се компилират с опция **-g** и да се изключат всички оптимизации на компилатора.

За тестови цели ще създадем 2 примера:

```
// buggy.c
#include <stdio.h>

#define XMAX 5
#define YMAX 5

int eval(int x, int y);
float density(int x, int y);

int main(void) {
  int x, y;
  float data[XMAX][YMAX];

  for (y=0; y < YMAX; y++) {
    for (x=0; x < XMAX; x++) {
      data[x][y]=density(x,y);
    }
  }

  for (y=0; y <= YMAX; y++) {
    for (x=0; x <= XMAX; x++) {
      printf("data[%i][%i] should be %f, is
%f\n",x,y,eval(x,y)/5.0,data[x][y]);
    }
  }

  return 0;
}

int eval(int x, int y) {
  int result;

  result=((XMAX*XMAX)-(x*x))/((YMAX*YMAX)-(y*y));

  return result;
}

float density(int x, int y) {
  int result;

  result=eval(x,y);
  result=result/5;

  return (float)result;
}
// buggy2.c
#include <stdio.h>

#define XMAX 5
```

```
#define YMAX 5

int eval(int x, int y);
float density(int x, int y);

int main(void) {
  int x, y;
  float data[XMAX][YMAX];

  for (y=0; y < YMAX; y++) {
    for (x=0; x < XMAX; x++) {
      data[x][y]=density(x,y);
    }
  }

  for (y=0; y < YMAX; y++) {
    for (x=0; x < XMAX; x++) {
      printf("data[%i][%i] should be %f, is
%f\n",x,y,eval(x,y)/5.0,data[x][y]);
    }
  }

  return 0;
}

int eval(int x, int y) {
  int result;

  result=((XMAX*XMAX)-(x*x))*((YMAX*YMAX)-(y*y));

  return result;
}

float density(int x, int y) {
  int result;

  result=eval(x,y);
  result=result/5;

  return (float)result;
}
```

АНАЛИЗ НА CORE DUMPS

Понякога програмата спира изпълнението си с грешка и генерира "core dump" файл. На Linux системи той може да бъде намерен в директорията, където е стартирана програмата и се нарича `core.NNNNN`, където `NNNNN` е номера на процеса, който има програмата. Ако такъв файл не е създаден, трябва да се зададе максимален размер на core dump file равен на нула, посредством:
`ulimit -c unlimited`

Ето и първия пример:

Компилиране и стартиране на програмата

```
[students@FKSU1207-2 Example]$ gcc -g -o buggy buggy.c
[students@FKSU1207-2 Example]$ ./buggy
data[0][0] should be 0.200000, is 0.000000
data[1][0] should be 0.000000, is 0.000000
data[2][0] should be 0.000000, is 0.000000
.
.
.
data[3][4] should be 0.200000, is 0.000000
data[4][4] should be 0.200000, is 0.000000
data[5][4] should be 0.000000, is 0.000000
Floating point exception (core dumped)
```

Да намерим името на core файла:

```
[students@FKSU1207-2 Example]$ ls core*
core.871
```

Стартираме **gdb** с името на програмата и името на core file като аргументи:

```
[students@FKSU1207-2 Example]$ gdb buggy core.871
GNU gdb Red Hat Linux (7.2-60.e16)
Copyright 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/students/SP/Demos/Example/buggy...done.
[New Thread 871]
Missing separate debuginfo for
Try: yum --disablerepo='*' --enablerepo='*-debug*' install
/usr/lib/debug/.build-id/a6/993d9af0d108bfc4a2bbfdb176ea3288f6fd5c
Reading symbols from /lib64/libc.so.6...(no debugging symbols
found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging
symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2

Core was generated by `./buggy'.
Program terminated with signal 8, Arithmetic exception.
```

gdb показва реда, който е предизвикал грешката заедно с актуалната стойност на променливите:

```
#0  0x00000000004005f8 in eval (x=0, y=5) at buggy.c:31
31        result=((XMAX*XMAX)-(x*x))/((YMAX*YMAX)-(y*y));
```

Ако работим със сложна програма, то използваме `backtrace` команда за да намерим къде се намира функцията в кода.

```
(gdb) backtrace
#0  0x00000000004005f8 in eval (x=0, y=5) at buggy.c:31
```

```
#1  0x0000000000400575 in main () at buggy.c:21
```

Тъй като **YMAX** е **#define 5,** то се получава делене на нула, когато се извика `eval()` с аргументи(`x=0, y=5`).

Изход от `gdb`:

```
(gdb) quit
```

GENERAL DEBUGGING

Една често срещана ситуация се получава, когато програмата не изпълнява, това което се очаква от нея.

В следващия пример тя работи ОК, но е налице неправилен резултат при `x=1, y=1`:

```
[students@FKSU1207-2 Example]$ gcc -g -o buggy2 buggy2.c
[students@FKSU1207-2 Example]$ ./buggy2
data[0][0] should be 125.000000, is 125.000000
data[1][0] should be 120.000000, is 120.000000
data[2][0] should be 105.000000, is 105.000000
data[3][0] should be 80.000000, is 80.000000
data[4][0] should be 45.000000, is 45.000000
data[0][1] should be 120.000000, is 120.000000
data[1][1] should be 115.200000, is 115.000000
data[2][1] should be 100.800000, is 100.000000
.
.
.
data[4][4] should be 16.200000, is 16.000000
```

Стартираме **gdb** само с името на файла:

```
[students@FKSU1207-2 Example]$ gdb buggy2
GNU gdb Red Hat Linux (7.2-60.e16)
Copyright 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/students/SP/Demos/Example/buggy...done.
```

Използваме команда **list** за да прегледаме кода и номерата на редовете:

```
(gdb) list
4       #define YMAX 5
5
6       int eval(int x, int y);
7       float density(int x, int y);
8
```

```
9       int main(void) {
10        int x, y;
11        float data[XMAX][YMAX];
12
13        for (y=0; y < YMAX; y++) {
(gdb) list
14          for (x=0; x < XMAX; x++) {
15            data[x][y]=density(x,y);
16          }
17        }
18
19        for (y=0; y < YMAX; y++) {
20          for (x=0; x < XMAX; x++) {
21            printf("data[%i][%i] should be %f, is
%f\n",x,y,eval(x,y)/5.0,data[x][y]);
22          }
23        }
```

Функцията **density**() е добро място за точка на прекъсване:

```
(gdb) break density
Breakpoint 1 at 0x400614: file buggy2.c, line 39.
```

Стартираме програмата:

```
(gdb) run
Starting program: /home/students/SP/Demos/Example/buggy2

Breakpoint 1, density (x=0, y=0) at buggy2.c:39
39        result=eval(x,y);
```

След достигане на точката на прекъсване, може да се провери съдържанието на променливата **result** с командата **print** :

```
(gdb) print result
$1 = 1
```

Използваме командата **step** за да изпълним един оператор:

```
(gdb) step
eval (x=0, y=0) at buggy2.c:31
31        result=((XMAX*XMAX)-(x*x))*((YMAX*YMAX)-(y*y));
```

В този случай бе извикана функцията **eval**(), което можем да видим по-добре с **backtrace**:

```
(gdb) backtrace
#0  eval (x=0, y=0) at buggy2.c:31
#1  0x0000000000400623 in density (x=0, y=0) at buggy2.c:39
#2  0x00000000004004f7 in main () at buggy2.c:15
```

За да довършим изпълнението на **eval**() в една стъпка, използваме команда **finish:**

```
(gdb) finish
Run till exit from #0  eval (x=0, y=0) at buggy2.c:31
0x0000000000400623 in density (x=0, y=0) at buggy2.c:39
39        result=eval(x,y);
Value returned is $2 = 625
```

След още една стъпа, получаваме стойността на **result**:

```
(gdb) step
40        result=result/5;
(gdb) print result
$3 = 625
(gdb) step
42        return (float)result;
(gdb) print result
$4 = 125
```

Това изглежда ОК. Тъй като не желаем да спираме на всяка итерация на цикъла, а само да спрем, когато има проблем, ще забраним тази точка на прекъсване:

```
(gdb) disable 1
```

Задаваме нова точка на прекъсване на ред 40, точно преди програмата да намери резултата, посредством делене на 5:

```
(gdb) break 40
Breakpoint 2 at 0x400626: file buggy2.c, line 40.
```

Да проверим какви точки на прекъсване сме дефинирали:

```
(gdb) info break
Num Type           Disp Enb Address            What
1   breakpoint     keep n   0x0000000000400614 in density at
buggy2.c:39
        breakpoint already hit 1 time
2   breakpoint     keep y   0x0000000000400626 in density at
buggy2.c:40
```

Сега ще зададем условие на новата точка на прекъсване, за да не спира всеки път, а само когато услоеивто стане истина:

```
(gdb) condition 2 (x==1 && y==1)
```

Продължаваме изпълнението на програмата:

```
(gdb) continue
Continuing.

Breakpoint 2, density (x=1, y=1) at buggy2.c:40
40        result=result/5;
```

Сега отново ще използваме **step** и **print** за да проверим къде е грешката:

```
(gdb) print result
$5 = 576
(gdb) step
42        return (float)result;
(gdb) print result
$6 = 115
```

Както се вижда, грешката се причинява от целочисленото делене.

Изход от **gdb**:

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1397] will be killed.
Quit anyway? (y or n) y
```

ОПИТАЙТЕ

Ето две интересни програми за упражнение:

// ll_equal.c

```
#include <stdio.h>

typedef struct node {
      int val;
      struct node* next;
} node;

/* FIXME: this function is buggy. */
int ll_equal(const node* a, const node* b) {
      while (a != NULL) {
            if (a->val != b->val)
                  return 0;
            a = a->next;
            b = b->next;
      }
      /* lists are equal if a and b are both null */
      return a == b;
}

int main(int argc, char** argv) {
      int i;
      node nodes[10];

      for (i=0; i<10; i++) {
            nodes[i].val = 0;
            nodes[i].next = NULL;
      }

      nodes[0].next = &nodes[1];
      nodes[1].next = &nodes[2];
      nodes[2].next = &nodes[3];

      printf("equal test 1 result = %d\n", ll_equal(&nodes[0],
&nodes[0]));
      printf("equal test 2 result = %d\n", ll_equal(&nodes[0],
&nodes[2]));

      return 0;
}
```

// ll_cycle.c

```
#include <stdio.h>

typedef struct node {
      int value;
      struct node *next;
} node;

int ll_has_cycle(node *head) {
      /* your code here */
}
```

```
void test_ll_has_cycle(void) {
     int i;
     node nodes[25]; //enough to run our tests
     for(i=0; i < sizeof(nodes)/sizeof(node); i++) {
          nodes[i].next = 0;
          nodes[i].value = 0;
     }
     nodes[0].next = &nodes[1];
     nodes[1].next = &nodes[2];
     nodes[2].next = &nodes[3];
     printf("Checking first list for cycles. There should be none,
ll_has_cycle says it has %s cycle\n",
ll_has_cycle(&nodes[0])?"a":"no");

     nodes[4].next = &nodes[5];
     nodes[5].next = &nodes[6];
     nodes[6].next = &nodes[7];
     nodes[7].next = &nodes[8];
     nodes[8].next = &nodes[9];
     nodes[9].next = &nodes[10];
     nodes[10].next = &nodes[4];
     printf("Checking second list for cycles. There should be a
cycle, ll_has_cycle says it has %s cycle\n",
ll_has_cycle(&nodes[4])?"a":"no");

     nodes[11].next = &nodes[12];
     nodes[12].next = &nodes[13];
     nodes[13].next = &nodes[14];
     nodes[14].next = &nodes[15];
     nodes[15].next = &nodes[16];
     nodes[16].next = &nodes[17];
     nodes[17].next = &nodes[14];
     printf("Checking third list for cycles. There should be a
cycle, ll_has_cycle says it has %s cycle\n",
ll_has_cycle(&nodes[11])?"a":"no");

     nodes[18].next = &nodes[18];
     printf("Checking fourth list for cycles. There should be a
cycle, ll_has_cycle says it has %s cycle\n",
ll_has_cycle(&nodes[18])?"a":"no");

     nodes[19].next = &nodes[20];
     nodes[20].next = &nodes[21];
     nodes[21].next = &nodes[22];
     nodes[22].next = &nodes[23];
     printf("Checking fifth list for cycles. There should be none,
ll_has_cycle says it has %s cycle\n",
ll_has_cycle(&nodes[19])?"a":"no");

     printf("Checking length-zero list for cycles. There should be
none, ll_has_cycle says it has %s cycle\n",
ll_has_cycle(NULL)?"a":"no");
}

int main(void) {
```

```
  test_ll_has_cycle();
  return 0;
}
```

Компилирайте и стартирайте програмата ll_equal.c. Ще получите:

```
[students@FKSU1207-2 Example]$ gcc -g -o ll_equal ll_equal.c
[students@FKSU1207-2 Example]$ ./ll_equal
equal test 1 result = 1
Segmentation fault (core dumped)
```

Стартирайте дебъгера и открийте грешката.

Втората задача ще ви подпомогне в интервютата за работа. В **ll_cycle.c**, довършете функцията **ll_has_cycle()** като реализирате следния алгоритъм за проверка дали едносвързан списък има цикъл.

1) Започнете с два указателя към главата на списъка. Ще наречем първия **tortoise** (костенурка), а втория **hare** (заек).
2) Преместете **hare** с два елемента. Ако не е възможно поради указател **null**, сме достигнали края на списъка и следователно списъка е ацикличен.
3) Преместете **tortoise** с един елемент. Не е необходимо да проверявате за **null**. Защо?
4) Ако **tortoise** и **hare** сочат към един и същи елемент, то списъка е цикличен. В противен случай се повтаря стъпка 2.

След правилна реализация на **ll_has_cycle()**, проверете дали програмата работи коректно.

ПОЛЕЗНИ GDB КОМАНДИ

run
: Стартира програмата за изпълнение (или я рестартира, ако тя вече е стартирана). Ако програмата очаква аргументи от команден ред, те могат да бъдат подадени при **run**.

backtrace
: Показва йерархията от извикани функции в обратен ред.

break
: Задава точка на прекъсване, където ще спре изпълнението на програмата. Може да има един параметър, който може да бъде име на функция или номер на ред. Ако няма аргумент, точката на прекъсване се задава на текущия ред.

condition
: Задава условие на точката на прекъсване. Има два аргумента – номерът на точката, за която се отнася условието и израз, който трябва да дава резултат истина или лъжа.

continue
: Продължава изпълнението на програмата от текущия ред.

delete
: Изтрива точки на прекъсване. АКо се зададе цял аргумент, изтрива точка с посочения номер, ако няма аргумент – изтрива всички точки.

Ако се подаде списък от цели аргументи, се изтриват точките с номера от списъка.

`disable`

Както **delete**, но само сменя флага на точките на прекъсване в неактивен.

`enable`

Разрешава забранени точки на прекъсване. Има аргумент: номер на точка или списък номера на точки на прекъсване.

`finish`

Изпълнява текущата функция до края на една стъпка и връща управлението.

`help`

Стартиране на вградения help.

`next`

Премества изпълнението с един ред без да „влиза“ в подпрограми. Има опционен аргумент, цяла данна, която задава колко реда да се изпълнят наведнъж. **Изпълнението може да спре по-рано от зададеното, ако има активни точки на прекъсване**.

`print`

Разпечатва данни на екрана. Има един аргумент: име на променлива или израз, чиято стойност се изчислява.

`quit`

Изход от **gdb**.

`step`

Премества изпълнението с един ред като „влиза“ в подпрограми. Има опционен аргумент, цяла данна, която задава колко реда да се изпълнят наведнъж. **Изпълнението може да спре по-рано от зададеното, ако има активни точки на прекъсване**.

За запознаване с останалите команди може да ползвате gdb5-refcard.pdf.