

Гайд о том, как пользоваться gdb – самый минимум для отладки 1-2-3 лабы и выполнения домашнего задания.

Версия от 03.11.2022

Автор: Штанов Евгений Юрьевич

Есть вопросы? Для связи со мной:

https://t.me/Klavishnik0_o

https://vk.com/klavishnik0_o

shtanov.klavishnik@yandex.ru

1. Что это такое?

GDB – это отладчик. Самая полезная для вас (в данный момент) его функция – способность «заморозить» состояние программы и выполнять её пошагово.

2. Как начать пользоваться

Первое действие – необходимо собрать вашу программу с отладочными символами. Это нужно для того, чтобы «под отладкой» вы видели исходный код вашей программки, а не ассемблерные команды.

Пусть файл с исходным кодом у нас называется **1.c**, а скомпилированный бинарный файл будет иметь название **bin**. Флагом **-g** добавим отладочные символы к бинарному файлу. Для этого выполним следующие команды:

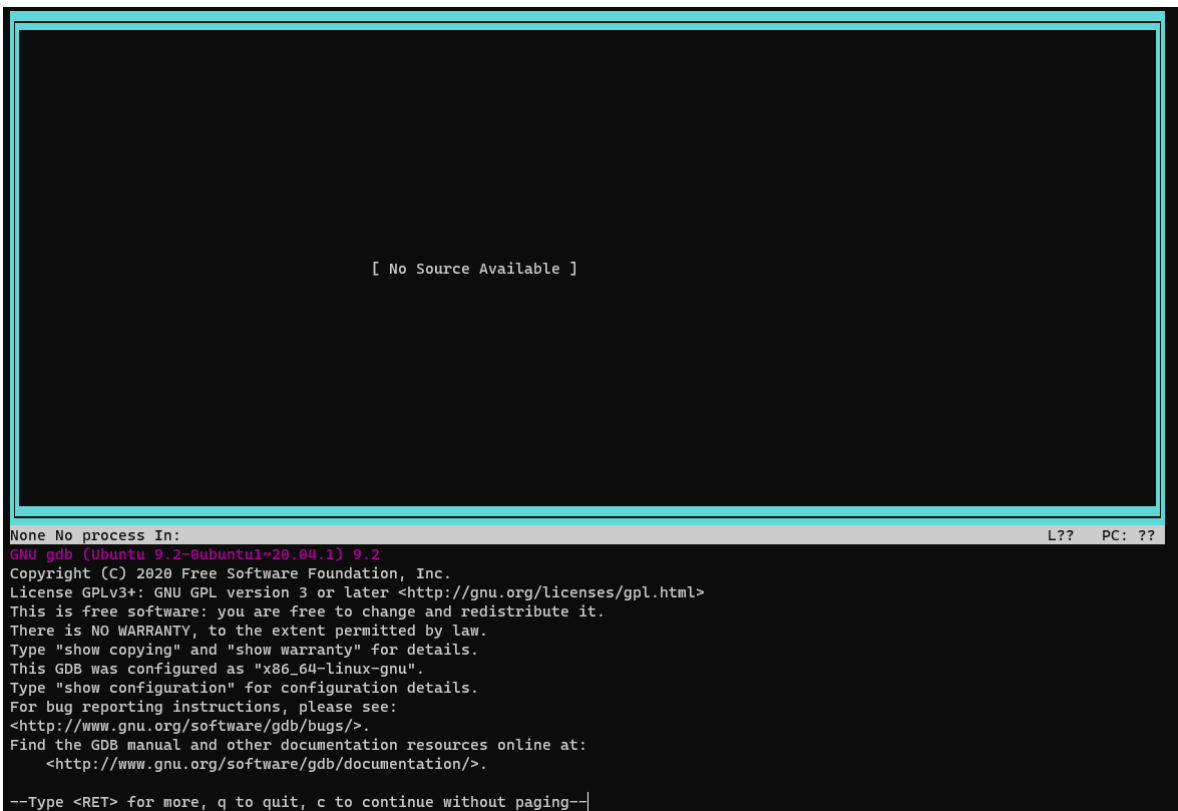
```
gcc 1.c -o bin -g
```

А теперь запустим отладчик. Запускается он командой **gdb** и на вход принимает название бинарного файла **bin**. Флаг **-tui** нужен для запуска в интерактивном режиме. Команда:

```
gdb -tui ./bin
```

3. Стадия отрицания

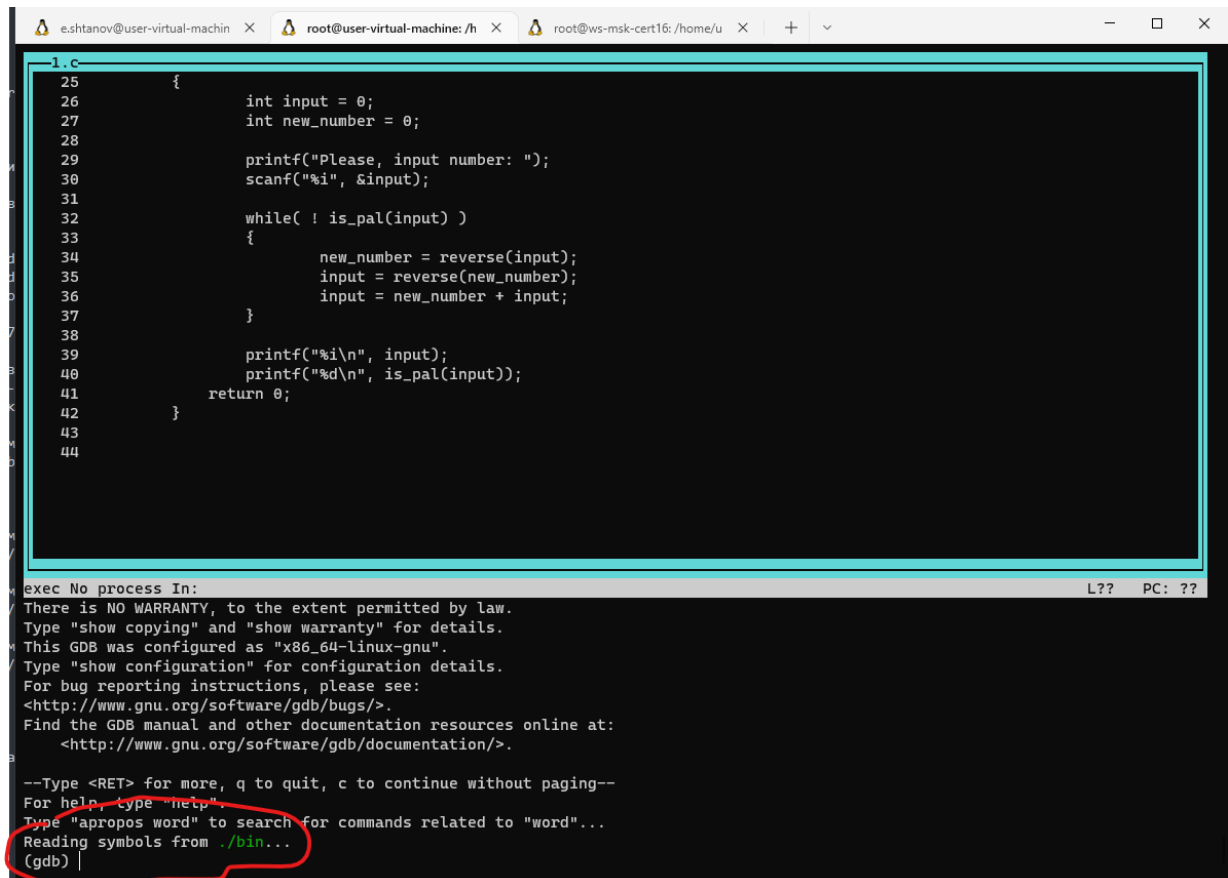
После запуска вы увидите такое окно



```
[ No Source Available ]

None No process in:                                     L??  PC: ??
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.
--Type <RET> for more, q to quit, c to continue without paging--
```

Нажимаем клавишу **Enter**



```
1.c
25 {
26     int input = 0;
27     int new_number = 0;
28
29     printf("Please, input number: ");
30     scanf("%i", &input);
31
32     while( ! is_pal(input) )
33     {
34         new_number = reverse(input);
35         input = reverse(new_number);
36         input = new_number + input;
37     }
38
39     printf("%i\n", input);
40     printf("%d\n", is_pal(input));
41     return 0;
42 }
43
44

exec No process in:
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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

--Type <RET> for more, q to quit, c to continue without paging--
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bin...
(gdb) |
```

Видно, что появился доступ работе с окружением gdb и отладчик подгрузил отладочные символы для bin, поэтому в окне выше показан исходный код.

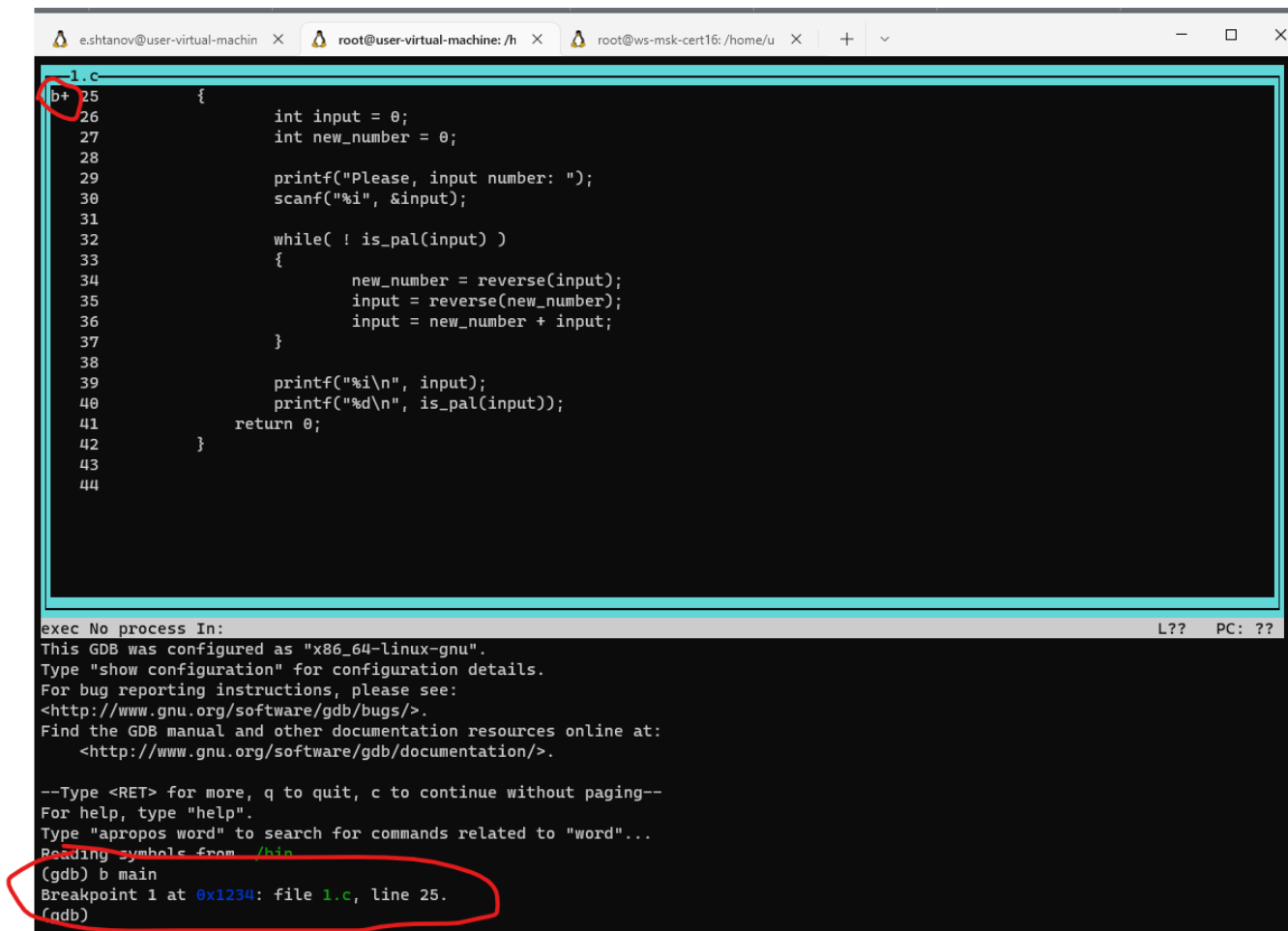
4. Стадия гнева

Для того, чтобы начать пользоваться отладчиком, нужно запустить программу. НО, если мы ее запустим сейчас, то она выполнится вся. Нужно остановить выполнение программы. Для этого существует механизм «точек останова» или **breakpoint**. Если коротко – это своеобразная «метка», которая ставится на строку, при достижении которой отладчик остановит выполнение вашей программы, зафиксировав её состояние.

Чтобы поставить точку останова необходимо знать название функции или номер строки. Поскольку точка входа в нашу программу является функция **main** (т.е. для нас это начало нашей программы), то на неё и поставим брейкпоинт командой:

```
b main
```

Видно, что появился указатель с точкой останова **b+**, а в выводе дана информация на какой строке стоит точка останова



```
1.c
b+
25     {
26         int input = 0;
27         int new_number = 0;
28
29         printf("Please, input number: ");
30         scanf("%i", &input);
31
32         while( ! is_pal(input) )
33         {
34             new_number = reverse(input);
35             input = reverse(new_number);
36             input = new_number + input;
37         }
38
39         printf("%i\n", input);
40         printf("%d\n", is_pal(input));
41         return 0;
42     }
43
44

exec No process in:
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

--Type <RET> for more, q to quit, c to continue without paging--
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /bin/...
(gdb) b main
Breakpoint 1 at 0x1234: file 1.c, line 25.
(gdb)
```

Чтобы посмотреть все точки останова нужно выполнить команду

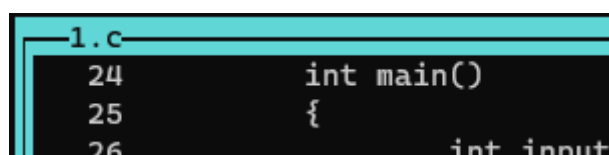
`info b`

Чтобы удалить точку останова нужно выполнить команду

`d 1`, где 1 – это номер точки останова

```
(gdb) info b
Num      Type      Disp Enb Address          What
1        breakpoint keep y   0x0000000000001234 in main at 1.c:25
(gdb) d 1
(gdb) info b
No breakpoints or watchpoints.
(gdb)
```

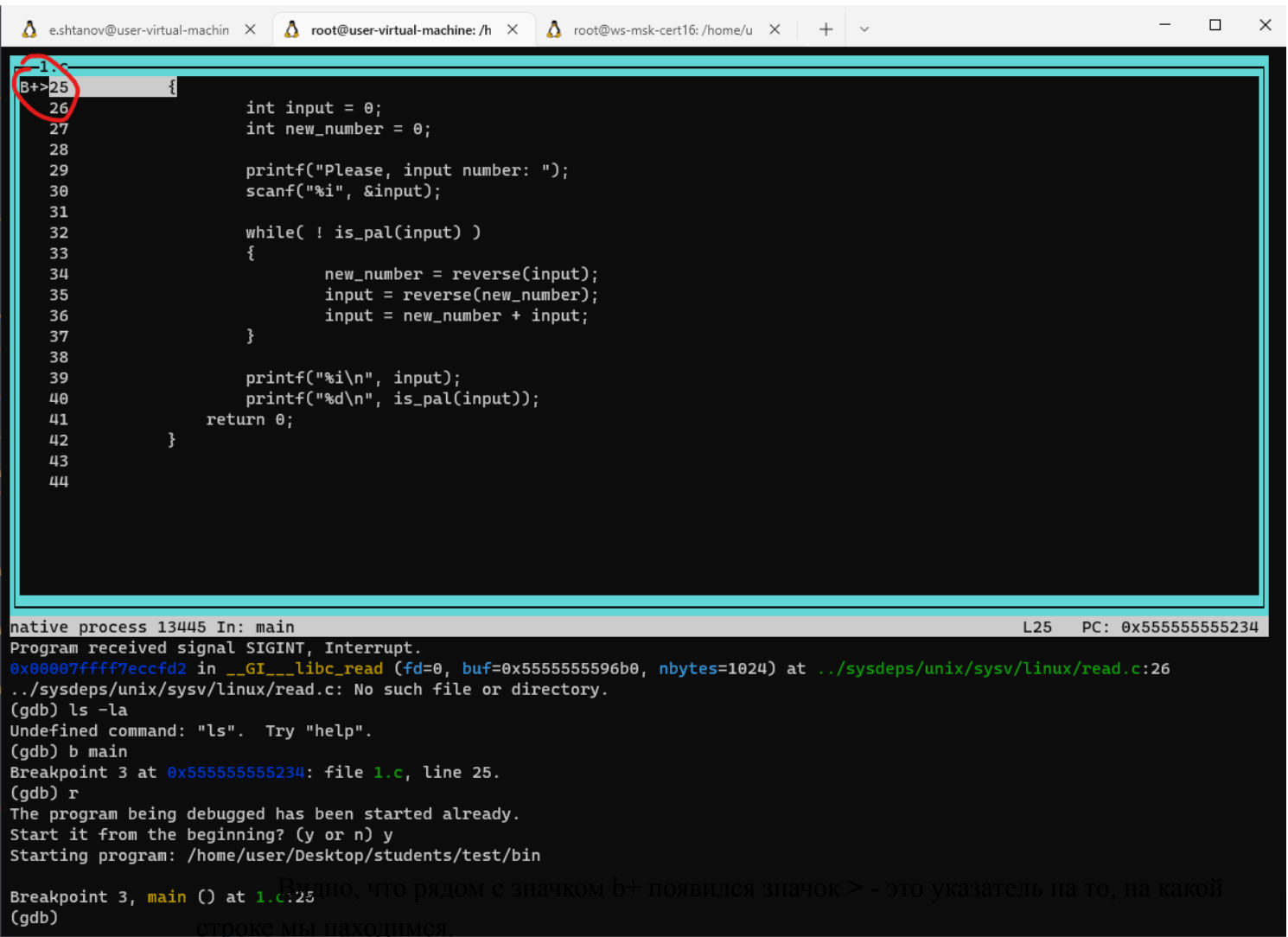
Указатель на точку останова в области исходного кода тоже пропал



```
1.c
24     int main()
25     {
26         int input
```

5. Стадия торга

Теперь начнем выполнять нашу программу. Установив точку останова, наберем **r** (от run – запуск, хотя можно и **g**un написать)



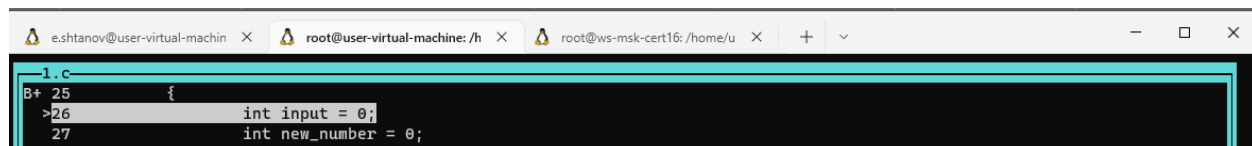
```
1
25 {
26     int input = 0;
27     int new_number = 0;
28
29     printf("Please, input number: ");
30     scanf("%i", &input);
31
32     while( ! is_pal(input) )
33     {
34         new_number = reverse(input);
35         input = reverse(new_number);
36         input = new_number + input;
37     }
38
39     printf("%i\n", input);
40     printf("%d\n", is_pal(input));
41     return 0;
42 }
43
44
```

native process 13445 In: main L25 PC: 0x55555555234
Program received signal SIGINT, Interrupt.
0x00007ffff7eccfd2 in __GI___libc_read (fd=0, buf=0x5555555596b0, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:26
../sysdeps/unix/sysv/linux/read.c: No such file or directory.
(gdb) ls -la
Undefined command: "ls". Try "help".
(gdb) b main
Breakpoint 3 at 0x55555555234: file 1.c, line 25.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Desktop/students/test/bin

Видно, что рядом с значком b+ появился значок > - это указатель на то, на какой строке мы находимся.

В отладчике можно перемещаться только вперед – к предыдущему состоянию вернуться нельзя!

Теперь попробуем пойти дальше. Для этого введем **n** (от next)



```
1.c
B+ 25 {
>26     int input = 0;
27     int new_number = 0;
28
```

Видно, что перешли на новую строку, после выполнения которой будет инициализирована и объявлена переменная `input`.

Для того, чтобы посмотреть состояние переменных, используем команду **p** **название_переменной** (от **print**)

p input

```
(gdb) n
(gdb) p input
$1 = -7008
```

Переменная имеет такое значение, поскольку еще не проинициализирована новым значением и хранит в себе данные, оставшиеся от прошлых использований.

Сделаем **n** и **print input** еще раз

```
1.c
B+ 25      {
26          int input = 0;
>27      int new_number = 0;
28
29          printf("Please, input number: ");
30          scanf("%i", &input);
31
32          while( ! is_pal(input) )
33          {
34              new_number = reverse(input);
35              input = reverse(new_number);
36              input = new_number + input;
37          }
38
39          printf("%i\n", input);
40          printf("%d\n", is_pal(input));
41          return 0;
42      }
43
44

native process 13445 In: main
Breakpoint 3 at 0x55555555234: file 1.c, line 25.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Desktop/students/test/bin

Breakpoint 3, main () at 1.c:25
(gdb) n
(gdb) p input
$1 = -7008
(gdb) n
(gdb) p input
$2 = 0
(gdb) |
```

Наглядно видно, что теперь переменная имеет нулевое значение.

Чтобы каждый раз не писать **print input** можно использовать конструкцию **display** **название_переменной** - тогда содержимое переменное будет **выводиться на каждом шаге**.

display input

```
1: input = 0
(gdb) n
1: input = 0
(gdb) n
1: input = 0
(gdb)
```

6. Стадия депрессии

Конструкциями **next** нельзя зайти внутрь функции – шаги будут идти строго по порядку в **main**. Для захода внутрь функции необходимо дойти до ее вызова и **оказавшись на ней** выполнить **s**.

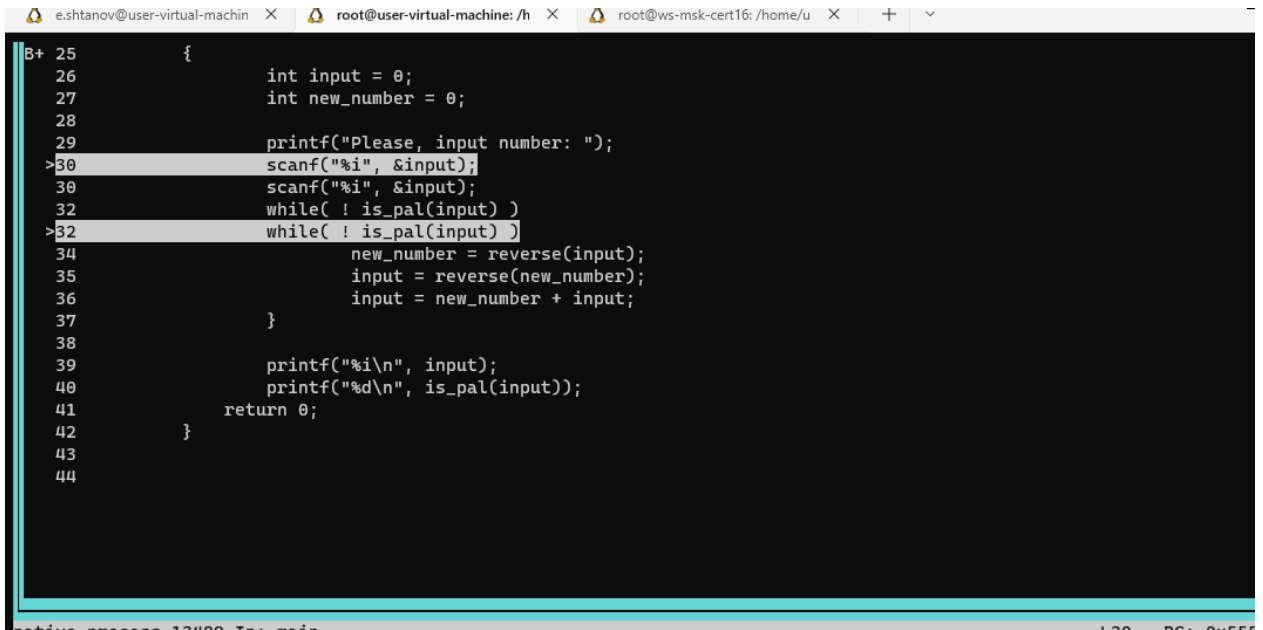
```
16      bool is_pal(int input)
>17      {
18          int reverse_num = reverse(input);
19          if (input == reverse_num)
20              return true;
21          return false;
22      }
23
24      int main()
B+ 25      {
26          int input = 0;
27          int new_number = 0;
28
29          printf("Please, input number: ");
30          scanf("%i", &input);
```

native process 13445 In: is_pal

```
(gdb) n
(gdb) p input
$2 = 0
(gdb) display input
1: input = 0
(gdb) n
1: input = 0
(gdb) n
1: input = 0
(gdb) n
1: input = 213123213
(gdb) s
is_pal (input=21845) at 1.c:17
(gdb) |
```

7. Стадия принятия

Если у вас «поехал» экран, как тут:

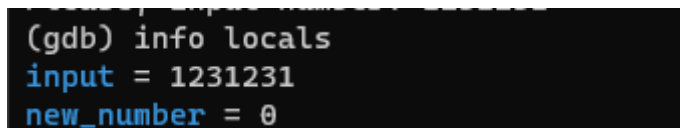


```
e.shtanov@user-virtual-machin X root@user-virtual-machine: /h X root@ws-msk-cert16: /home/u X + v
B+ 25      {
    26          int input = 0;
    27          int new_number = 0;
    28
    29          printf("Please, input number: ");
>80      scanf("%i", &input);
    30      scanf("%i", &input);
    31      while( ! is_pal(input) )
>82      while( ! is_pal(input) )
    34          new_number = reverse(input);
    35          input = reverse(new_number);
    36          input = new_number + input;
    37      }
    38
    39      printf("%i\n", input);
    40      printf("%d\n", is_pal(input));
    41      return 0;
    42  }
    43
    44
```

То нажмите комбинацию **ctrl+l**

Чтобы выйти из gdb нажмите **q**

Чтобы посмотреть все переменные в области видимости выполните **info locals**



```
(gdb) info locals
input = 1231231
new_number = 0
```


8. Массивы

А что делать с массивами?

Далее идут сакральные знания.

- 1) Для начала нужно вспомнить, переменная в памяти - это одна ячейка. А массив данных в памяти - это набор таких ячеек, которые расположены друг за другом. Напоминаю, что в Си работа с массивами организована через указатель - переменную, **значение** которой - ЭТО АДРЕС ПЕРВОГО ЭЛЕМЕНТА МАССИВА.
- 2) GDB по синтаксису и возможностям очень похож на Си.

А теперь перейдем к экспериментам:

```
int main()
{
    int size = 0;

    printf("Enter size of array: ");
    scanf("%d", &size);

    int * array = (int *) malloc( sizeof(int) * size);

    for(int i = 0; i < size; i++)
        array[i] = i;

    array_print(array, size);

    return 0;
}
```

Есть вот такой код. Видно, что тут вводится с клавиатуры количество элементов массива, дальше под массив выделяется динамически память, а потом каждому элементу массива приравнивается значение его индекса.

С этим разобрались.

Если раньше переменную мы смотрели командой `p (print)`, то и массив можно посмотреть этой командой. Логично? Логично. Смотрим.

```
14
15 int main()
16 {
17     int size = 0;
18
19     printf("Enter size of array: ");
20     scanf("%d", &size);
21
22     int * array = (int *) malloc( sizeof(int) * size);
23
24     for(int i = 0; i < size; i++)
25         array[i] = i;
26
27     array_print(array, size);
28
29     return 0;
30
multi-thre Thread 0x7ffff7dc27 In: main
(gdb) p array
$2 = (int *) 0x0
```

Почему ноль? Потому что в данный момент массив не инициализирован! ПАМЯТЬ ЕЩЕ НЕ ВЫДЕЛЕНА. УКАЗАТЕЛЬ УКАЗЫВАЕТ В ПУСТОТУ.

Делаем **n** и **p array** смотрим, что получилось

```
14
15 int main()
16 {
17     int size = 0;
18
19     printf("Enter size of array: ");
20     scanf("%d", &size);
21
22     int * array = (int *) malloc( sizeof(int) * size);
23
24     for(int i = 0; i < size; i++)
25         array[i] = i;
26
27     array_print(array, size);
28
29     return 0;
30
multi-thre Thread 0x7ffff7dc27 In: main
$7 = (int *) 0x0
(gdb) p array
$8 = (int *) 0x0
(gdb) p array
$9 = (int *) 0x0
(gdb) p ^array
A syntax error in expression, near '^array'.
(gdb) p &array
$10 = (int **) 0x7ffffffffffd410
(gdb) n
(gdb) p array
$11 = (int *) 0x405ac0
(gdb) |
```

Видим, что теперь в указателе **array** хранится какой-то адрес.

А этот адрес - это место, где будут расположены данные нашего массива.

Указатель содержит адрес, по которому хранятся данные (повторяю 100500 раз потому что вы не понимаете).

Если мы сейчас залезем в этот участок памяти, то мы там ничего нормального не увидим - элементы массива не проинициализированы, поэтому там будет валяться “мусор” - данные от операционной системы.

“Перескочим” участок кода, где происходит инициализация массива.

Кидаем точку останова на строку 27 и продолжаем выполнение программы:

b 27

c

```
14
15 int main()
16 {
17     int size = 0;
18
19     printf("Enter size of array: ");
20     scanf("%d", &size);
21
22     int * array = (int *) malloc( sizeof(int) * size);
23
24     for(int i = 0; i < size; i++)
25         array[i] = i;
26
27     array_print(array, size);
28
29     return 0;
30
B+
B+>

multi-thre Thread 0x7ffff7dc27 In: main L27 PC
A syntax error in expression, near '^array'.
(gdb) p &array
$10 = (int **) 0x7fffffd410
(gdb) n
(gdb) p array
$11 = (int *) 0x405ac0
(gdb) b 27
Breakpoint 2 at 0x401229: file 1.c, line 27.
(gdb) c
Continuing.

Breakpoint 2, main () at 1.c:27
(gdb) |
```

Смотрим, что у нас хранится в **array**

p array

```
26
B+> 27 array_print(array, size);
28
29 return 0;
30

multi-thre Thread 0x7ffff7dc27 In: main
$10 = (int **) 0x7fffffd410
(gdb) n
(gdb) p array
$11 = (int *) 0x405ac0
(gdb) b 27
Breakpoint 2 at 0x401229: file 1.c, line 27.
(gdb) c
Continuing.

Breakpoint 2, main () at 1.c:27
(gdb) p array
$12 = (int *) 0x405ac0
(gdb) |
```

Хм, ничего не изменилось, как же так.

А все правильно! Мы смотрим то, чему равно значение указателя! А там адрес! И он не менялся! А изменилось значение по этому адресу!

А как посмотреть?

Да как и в Си, с помощью операции разыменования

`p *array`

```
26
B+> 27 array_print(array, size);
28
29 return 0;
30

multi-thre Thread 0x7ffff7dc27 In: main
(gdb) p array
$11 = (int *) 0x405ac0
(gdb) b 27
Breakpoint 2 at 0x401229: file 1.c, line 27.
(gdb) c
Continuing.

Breakpoint 2, main () at 1.c:27
(gdb) p array
$12 = (int *) 0x405ac0
(gdb) p *array
$13 = 0
(gdb) |
```

Ноль! Правильно, это ведь значение нулевого элемента массива.

А давайте посмотрим чему равен следующий элемент?

`p *(array+1)`

```
14
15 int main()
16 {
17     int size = 0;
18
19     printf("Enter size of array: ");
20     scanf("%d", &size);
21
22     int * array = (int *) malloc( sizeof(int) * size);
23
24     for(int i = 0; i < size; i++)
25         array[i] = i;
26
B+> 27     array_print(array, size);
28
29     return 0;
30
31 }
```

```
(gdb) p *array
$2 = 0
(gdb) p *(array+1)
$3 = 1
```

Единица. Все правильно.

Таким способом можно хоть весь массив посмотреть. Например третий элемент

`p *(array+3)`

```
(gdb) p *(array+3)
$4 = 3
```

Вроде все верно.

Т.е. конструкция имеет вид

`p *(name_array + i)`, где **i** - номер индекса массива

А как чекнуть то весь массив?

Да как и в Сишке! Проблема фундаментальная - нужно знать какой длины у тебя массив.

Конструкция выглядит так:

`p *name_array@size_to_display`

Причем здесь **size_to_display** - это сколько элементов массива вывести

Т.е. у массив длиной 10 элементов можно вывести на экран как 2 элемента, так и 14, только во втором случае вы залезете в “чужую” память на 4 элемента.

И так, пишем команду

`p *array@size`

(size - моя переменная у меня в коде объявленная, это размер массива)

```
26
27 array_print(array, size);
28
29 return 0;
30

multi-thre Thread 0x7ffff7dc27 In: main L27 PC: 0
$14 = 1
(gdb) p i
No symbol "i" in current context.
(gdb) p size
$15 = 10
(gdb) p *array+3
$16 = 3
(gdb) p array@size
$17 = {0x405ac0, 0xaafffd839, 0x7fffffd538, 0x7ffff7dee237 <__libc_start_call_main+103>, 0x0, 0x4011b2 <main>, 0x100000002,
0x7fffffd538, 0x7fffffd548, 0x47670c2e523a72dc}
(gdb) p *array@size
$18 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb) |
```

(На картинке наглядно видно, что будет, если забыть поставить звездочку)

Ну собственно вывелся весь массив.

А если хочу посмотреть первые пять элементов?

Пишем

`p *array@5`

```
(gdb) p *array@5
$19 = {0, 1, 2, 3, 4}
(gdb) |
```

Работает.

9. Доступ к памяти

Ну в целом понятно. А как именно в ПАМЯТИ то данные посмотреть?

Есть такая конструкция

x/NUM**спецификатор_системы_счисления****размер_данных** **адрес**
(ПРОБЕЛ ЗДЕСЬ ОДИН!)

Расшифровываю

x/- - это команда вызова hex редактора gdb;

спецификатор_системы_счисления - в каком виде будет вывод на экран. Если написать **x** то шестнадцатеричный, если **d** то десятичный, **c** - символ;

ДЛЯ ОТДЕЛЬНОГО ВЫВОДА СТРОКИ ИСПОЛЬЗУЙ СПЕЦИФИКАТОР S

```
gef> x/2s 0x602000000010
0x602000000010: "n"
0x602000000012: ""
```

размер_данных - **b** - байт, **h** - половина слова (2 байта) **w** - слово (4 байта) **g** - двойное слово (8 байт)

NUM - сколько элементов выбранной размерности показать

адрес - говорит само за себя. Тут в качестве аргумента передаем адрес (хоть ручками набранный 0x777ff..., хоть переменную-указатель)

Например, хочу вывести на экран все 10 элементов моего массива (10 элементов типа данных `int` (4 байта) в десятичном виде):

x/10dw array

```
(gdb) x/10dw array
0x405ac0:      0      1      2      3
0x405ad0:      4      5      6      7
0x405ae0:      8      9
```

Слева видно адреса. Первый адрес - это адрес, который хранится в указателе (ему соответствует значение 0). Единица будет храниться по адресу `0x105ac0 + 4` (размер `int`) или (как мы пишем в Си) `*(array+1)`.

Между строками разница в 0x10 или в 16 элементов. Т.е. на одной строке у нас вмещается 4 элемента по 4 байта.

А теперь хочу вывести на экран все элементы моего массива по байтово в десятичном виде (40 элементов типа данных byte (1 байт * 10 элементов * 4 байта)):

x/40db array

```
(gdb) x/40db array
0x405ac0:  0      0      0      0      1      0      0      0
0x405ac8:  2      0      0      0      3      0      0      0
0x405ad0:  4      0      0      0      5      0      0      0
0x405ad8:  6      0      0      0      7      0      0      0
0x405ae0:  8      0      0      0      9      0      0      0
```

Вывод побайтовый. Именно так хранится информация у вас в кампуктере.

Помните! Под разные типы данных выделяется разное количество памяти. Так тип данных `char` или `byte` занимает в памяти 1 байт, а тип данных `int` - 4 байта. (Конечно все еще зависит от архитектуры машины бла бла ..., но у нас именно так).

Поэтому мы можем смотреть разные данные по-разному - `int` как байт, как было сделано выше.

А если я хочу посмотреть какой-то определенный адрес?

Да пожалуйста.

Сначала получим адрес нашего массива, а потом глянем его в памяти ручками

p array

x/10dw адрес

```
(gdb) p array
$21 = (int *) 0x405ac0
(gdb) x/10dw 0x405ac0
0x405ac0:  0      1      2      3
0x405ad0:  4      5      6      7
0x405ae0:  8      9
(gdb) |
```

Все то же самое, что и через `array`.

10. Представление чисел в кампуктере

Тут должна быть отдельная глава о том, как представляются числа в кампуктере.

Честно - мне лень делать чужую работу, поэтому пишу коротко, остальное есть в гугле.

Есть три типа представления числа - прямой, обратный и дополнительные коды.

Положительные числа не представляют проблем, поэтому хранятся в прямом коде (это их чисто бинарный вид, можете в кУлькУляторе винды перевести число из десятичного в двоичное и получите его в прямом коде).

Дополнительный и обратный используются для представления отрицательных чисел.

А как в кампуктере то хранится это все дело? Как посмотреть?

Все очень просто.

Пишем код, запускаем отладчик.

И делаем такую команду

p /t название переменной

Смотрим

Прямой, дополнительный и обратный код

Число:

-123

Число двоичных разрядов:

32

CALCULATE

Диапазон:

[-2147483648,2147483647]

Представление положительного числа:

000000000000000000000000000000001111011

Обратный код:

1111111111111111111111110000100

Дополнительный код:

111111111111111111111110000101



От автора:

Пока я гуглю за вас инфу, моя ценность на рабочем рынке поднимается все выше и выше.

Учитесь делать это самостоятельно!

Не нашли ответ в ру сегменте сети? Переведите ваш запрос на английский, зайдите в браузер в режиме инкогнито и выполните поисковый запрос в гугле. Большая часть ответов будет на англоязычном сегменте сайта StackOverflow. Информацию там можно перевести с помощью встроенного в браузер переводчика.

Примеры работы санитайзеров ASAN и MSAN на типовых ошибках, возникающих при работе массивами. (<---- Это ссылка на репозиторий, нажми на неё)

Как собрать проект, состоящий из нескольких файлов? Как скомпилировать статическую или динамическую библиотеку? Смотри [здесь](#).