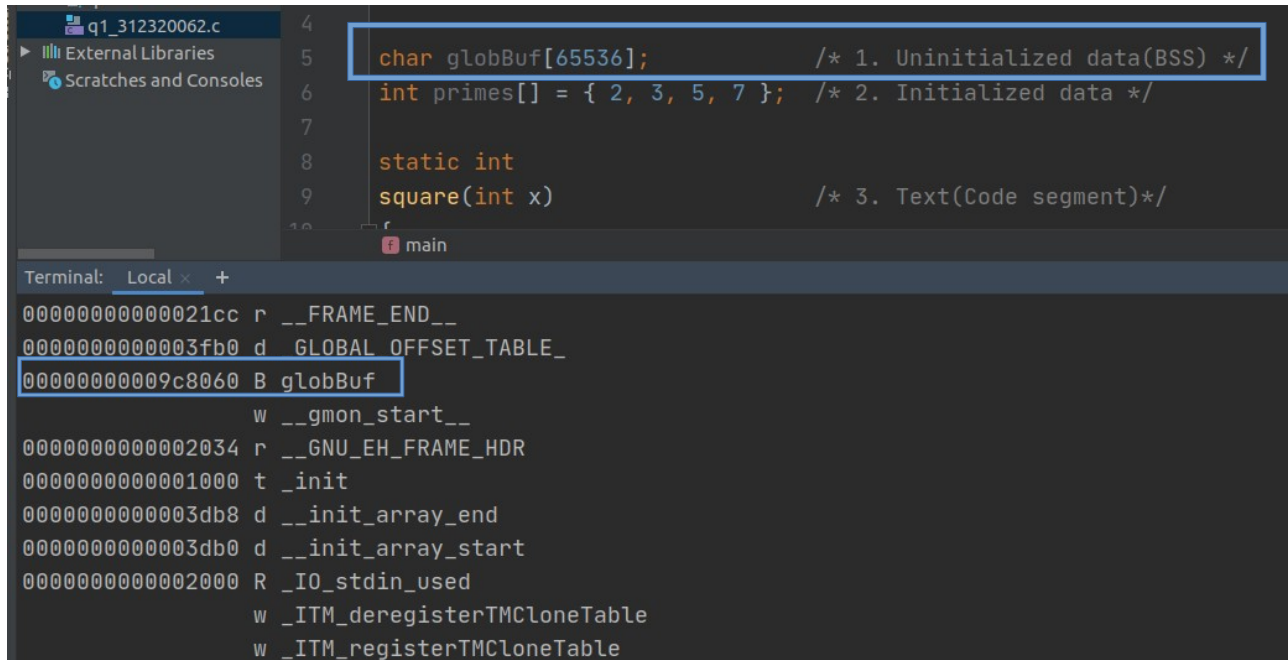


Question 1:

1. Where is allocated?

Answer: Uninitialized data(BSS)



The screenshot shows a code editor with the following C code:

```
4 char globBuf[65536]; /* 1. Uninitialized data(BSS) */
5 int primes[] = { 2, 3, 5, 7 }; /* 2. Initialized data */
6
7
8 static int
9 square(int x) /* 3. Text(Code segment)*/
10 {
11     return x * x;
12 }
13
14 int main() {
15     // ...
16 }
```

The terminal output shows the result of the `nm` command:

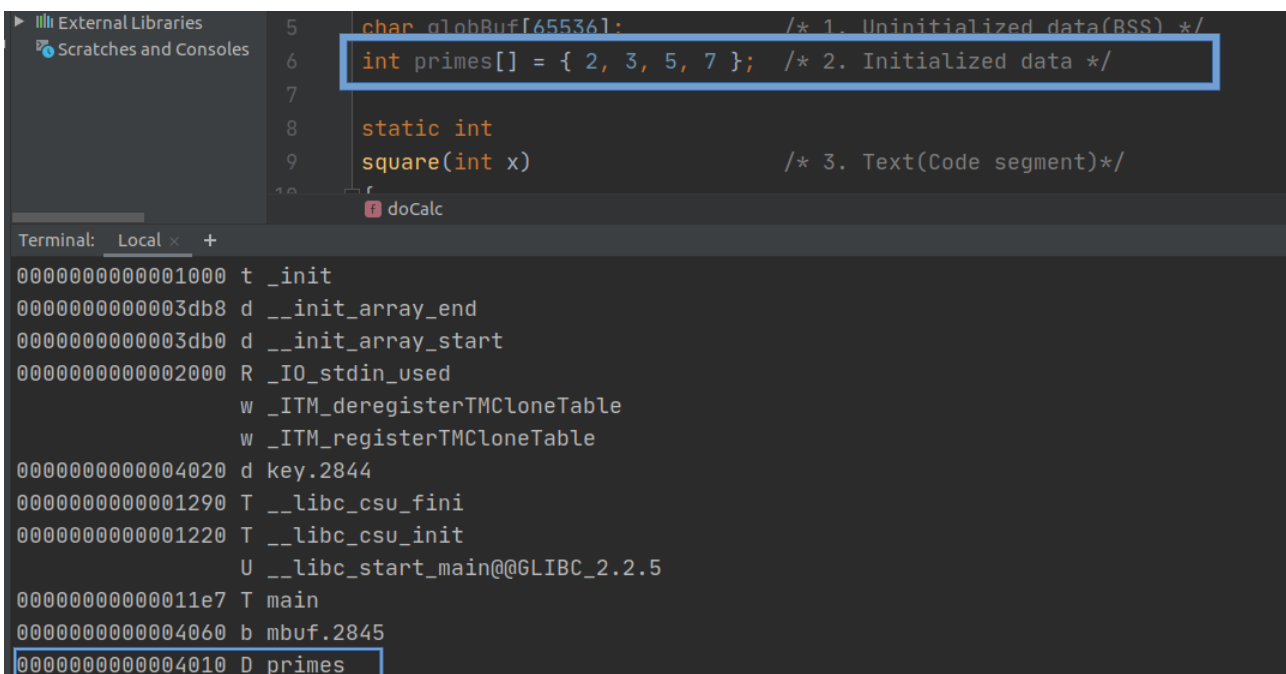
```
Terminal: Local x +
000000000000021cc r __FRAME_END__
00000000000003fb0 d GLOBAL_OFFSET_TABLE_
00000000000009c8060 B globBuf
w __gmon_start__
00000000000002034 r __GNU_EH_FRAME_HDR
00000000000001000 t _init
00000000000003db8 d __init_array_end
00000000000003db0 d __init_array_start
00000000000002000 R _IO_stdin_used
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
```

The line `00000000000009c8060 B globBuf` is highlighted, indicating that `globBuf` is located in the BSS segment.

Explanation:

I used the `nm` command, as we can see `globBuf` have a symbol type of `B` which means the `globBuf` is in Uninitialized data(BSS). Capital `b` means that `globBuf` is global variable.

2. Where is allocated?



The screenshot shows the same code editor as before, but with the following code:

```
5 char globBuf[65536]; /* 1. Uninitialized data(BSS) */
6 int primes[] = { 2, 3, 5, 7 }; /* 2. Initialized data */
7
8 static int
9 square(int x) /* 3. Text(Code segment)*/
10 {
11     return x * x;
12 }
13
14 int doCalc() {
15     // ...
16 }
```

The terminal output shows the result of the `nm` command:

```
Terminal: Local x +
00000000000001000 t _init
00000000000003db8 d __init_array_end
00000000000003db0 d __init_array_start
00000000000002000 R _IO_stdin_used
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
00000000000004020 d key.2844
00000000000001290 T __libc_csu_fini
00000000000001220 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
000000000000011e7 T main
00000000000004060 b mbuf.2845
00000000000004010 D primes
```

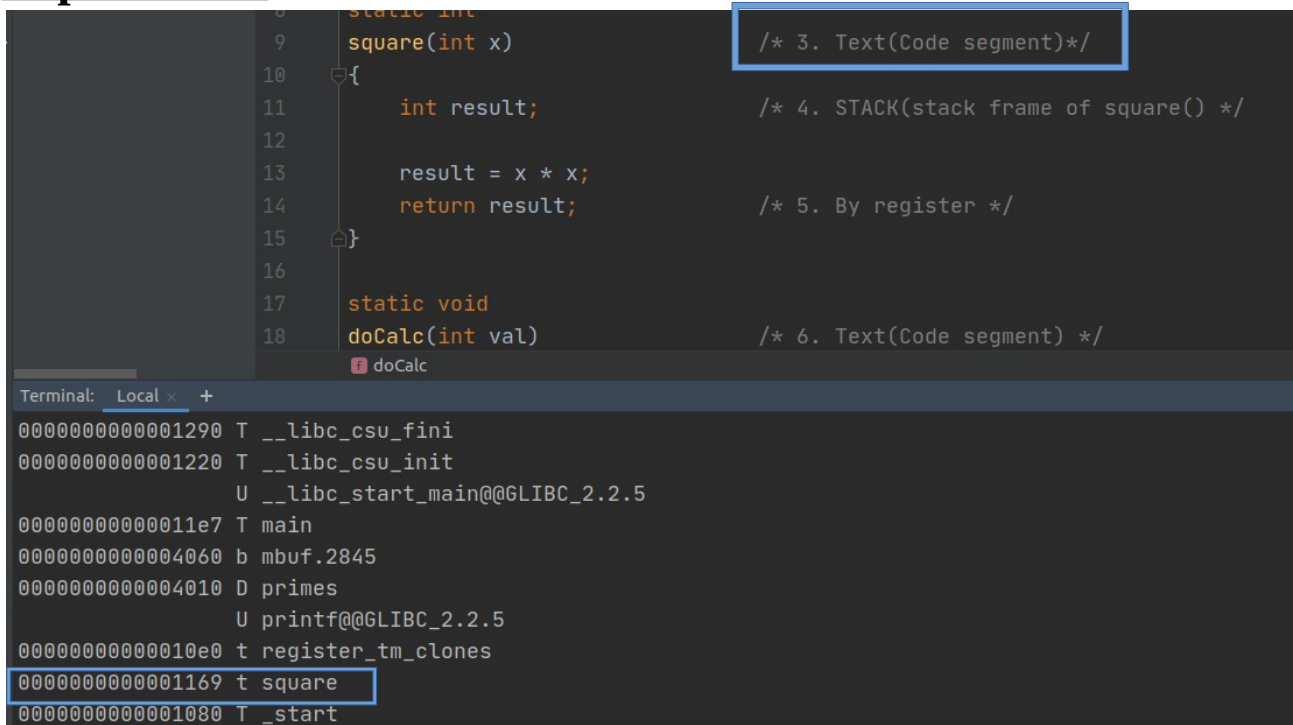
The line `00000000000004010 D primes` is highlighted, indicating that `primes` is located in the D segment.

Explanation:

I used the nm command, as we can see primes have symbol type of D , which means that primes is in Initilized data. Capital d means that primes is global variable.

3. Where is allocated?

Explanation:



The image shows a code editor with C code and a terminal window below it. The code defines a function `square` and a static function `doCalc`. Comments in the code identify segments: `/* 3. Text(Code segment)*/` for the function definition, `/* 4. STACK(stack frame of square() */` for the function body, `/* 5. By register */` for the return statement, and `/* 6. Text(Code segment) */` for the static function definition. The terminal shows the output of the `nm` command, listing symbols and their types. The symbol `square` is highlighted with a blue box, showing its type as `t` (Text/Code segment).

```
9      static int
10     square(int x)                                /* 3. Text(Code segment)*/
11     {
12         int result;                               /* 4. STACK(stack frame of square() */
13         result = x * x;
14         return result;                            /* 5. By register */
15     }
16
17     static void
18     doCalc(int val)                               /* 6. Text(Code segment) */
19     {
20         doCalc

```

```
Terminal: Local x +
0000000000001290 T __libc_csu_fini
0000000000001220 T __libc_csu_init
                U __libc_start_main@@GLIBC_2.2.5
00000000000011e7 T main
0000000000004060 b mbuf.2845
0000000000004010 D primes
                U printf@@GLIBC_2.2.5
00000000000010e0 t register_tm_clones
0000000000001169 t square
0000000000001080 T _start

```

I used the nm command, as we can see square have symbol type of t , which means that function pointer of square is in Text(Code segment).

```

000000000000001169 <square>:
 1169:      f3 0f 1e fa      endbr64
 116d:      55              push    %rbp
 116e:      48 89 e5        mov     %rsp,%rbp
 1171:      89 7d ec        mov     %edi,-0x14(%rbp)
 1174:      8b 45 ec        mov     -0x14(%rbp),%eax
 1177:      0f af c0        imul    %eax,%eax
 117a:      89 45 fc        mov     %eax,-0x4(%rbp)
 117d:      8b 45 fc        mov     -0x4(%rbp),%eax
 1180:      5d              pop     %rbp
 1181:      c3              retq

```

Lets look at the screen shot above and show that all the memory space of the function is in the stack, using objdump of the file.

I will explaine each line:

-push % rbp means that we push old base pointer to the stack(rbp holds the return value ,where we should return after done the function.).

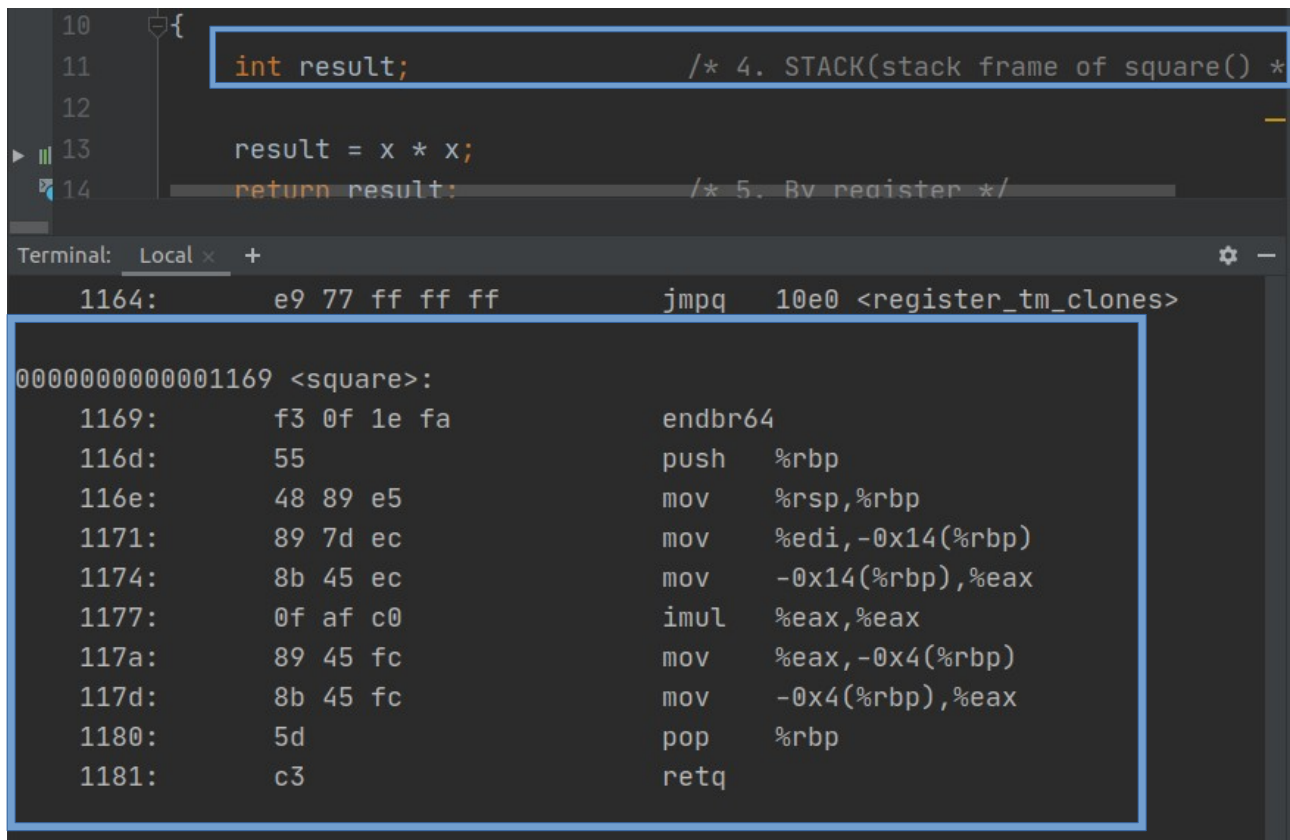
-mov %rsp,%rbp means that we move the value from %rsp(stack pointer ,the register have a value of the top of the stack frame<square>) into % rbp now rbp points to the top of the square stack frame.

-mov %edi,-0x14(%rbp) means to move the value from %edi(register that usally hold input parameters to function) into -0x14(%rbp)

, remember the stack grown downwards.

Thats way we can be sure the memory space of the funcion in the stack becuase the pointer went down(-0x14 bytes) the stack grown.

4. Where is allocated?



The screenshot shows a debugger window with two panes. The top pane displays C code for a function named `square`. The code is as follows:

```
10 {
11     int result; /* 4. STACK(stack frame of square() *)
12
13     result = x * x;
14     return result; /* 5. Rv register */
```

The bottom pane shows the assembly code for the `<square>` function. The assembly is as follows:

```
1164: e9 77 ff ff ff jmpq 10e0 <register_tm_clones>
00000000000001169 <square>:
1169: f3 0f 1e fa endbr64
116d: 55 push %rbp
116e: 48 89 e5 mov %rsp,%rbp
1171: 89 7d ec mov %edi,-0x14(%rbp)
1174: 8b 45 ec mov -0x14(%rbp),%eax
1177: 0f af c0 imul %eax,%eax
117a: 89 45 fc mov %eax,-0x4(%rbp)
117d: 8b 45 fc mov -0x4(%rbp),%eax
1180: 5d pop %rbp
1181: c3 retq
```

Explantion:

We will explaine this using objdump -d and explain line by line.

-push % rbp means that we push old base pointer to the stack(rbp holds the return value ,where we should return after done the function.).

-mov %rsp,%rbp means that we move the value from %rsp(stack pointer ,the register have a value of the top of the stack frame<square>) into % rbp now rbp points to the top of the square stack frame.

-mov %edi,-0x14(%rbp) push the parameter to function at the bottom of stack frame(at -0x14(%rbp)).

-mov -0x14(%rbp),%eax move the parameter we got from function into %eax(remember eax have 2 common use: to store the return value of a function and for certain calculations like mul and div.

-imul %eax,%eax multiplie the value by it self(in our program its $x*x$).

-mov %eax,-0x4(%rbp) move the value from eax into -0x4(%rbp) rbp points to the top of the stack frame. When we insert value into lower addres the stack acutally getting biger because it grown downwards to the lower address. Eax register had the value of paramater multipiled by it self and becuase we insert this value into -0x4(%rbp) we can say that result is in the stack frame of sqaure.

5. How the value return from function?

```
13      result = x * x;
14      return result; /* 5. By register */
15  }
16
17  static void
main
Terminal: Local x +
1154:      c3                      retq
1155:      0f 1f 00                nopl    (%rax)
1158:      c3                      retq
1159:      0f 1f 80 00 00 00 00    nopl    0x0(%rax)

00000000000001160 <frame_dummy>:
1160:      f3 0f 1e fa            endbr64
1164:      e9 77 ff ff ff        jmpq    10e0 <register_tm_clones>

00000000000001169 <square>:
1169:      f3 0f 1e fa            endbr64
116d:      55                      push    %rbp
116e:      48 89 e5                mov     %rsp,%rbp
1171:      89 7d ec                mov     %edi,-0x14(%rbp)
1174:      8b 45 ec                mov     -0x14(%rbp),%eax
1177:      0f af c0                imul    %eax,%eax
117a:      89 45 fc                mov     %eax,-0x4(%rbp)
117d:      8b 45 fc                mov     -0x4(%rbp),%eax
1180:      5d                      pop     %rbp
1181:      c3                      retq
```

Explanation:

The `eax` register have two common uses: to store the return value of a function and for certain calculations.

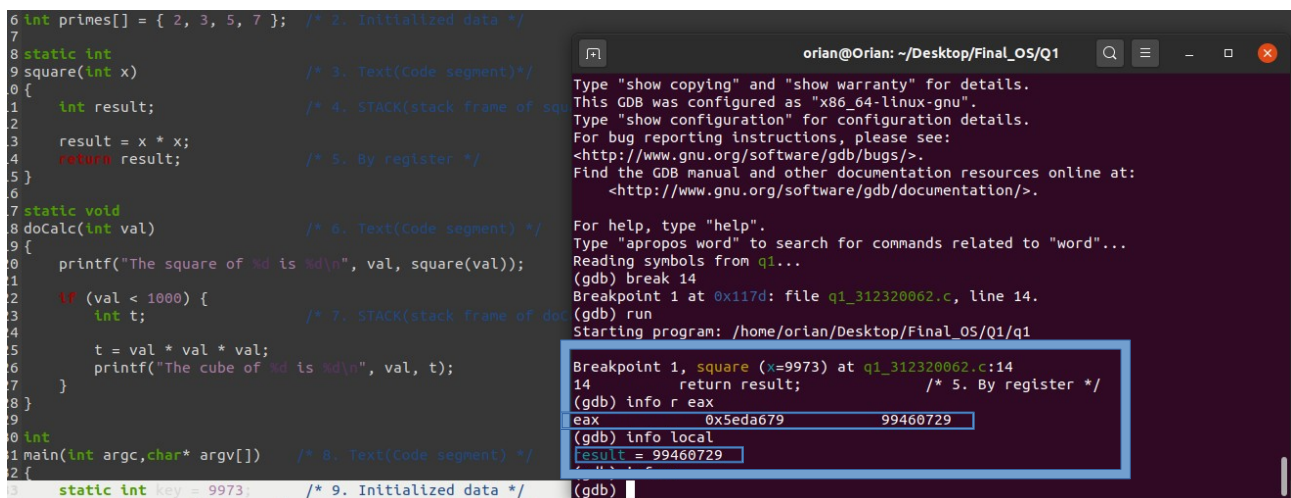
At line 1177 we can see that `eax` is multiplied by itself (in our program its the $x*x$).

At line 117a we can see that that value from `eax` register saved into `-0x4(%rbp)` (in our program its the `result = x*x`, the `eax` register has been multiplied by itself.). Lets remember that `rbp` register point the start of the function on the stack, the stack is

getting bigger when the pointer goes down. -0x4 is the result variable memory space.

At line 117d we can see that the eax register now has the value of -0x4(%rbp) (we took the value from result into the eax register, as we mentioned above the eax register is used to return the value from functions).

After this we pop %rbp which means that now the base pointer will be back to where we left when we started doing the function. Then the retq is return from the function.



```
6 int primes[] = { 2, 3, 5, 7 }; /* 2. Initialized data */
7
8 static int
9 square(int x) /* 3. Text(Code segment) */
10 {
11     int result; /* 4. STACK(stack frame of square) */
12     result = x * x;
13     return result; /* 5. By register */
14 }
15
16 static void
17 doCalc(int val) /* 6. Text(Code segment) */
18 {
19     printf("The square of %d is %d\n", val, square(val));
20
21     if (val < 1000) {
22         int t; /* 7. STACK(stack frame of doCalc) */
23         t = val * val * val;
24         printf("The cube of %d is %d\n", val, t);
25     }
26 }
27
28 int
29 main(int argc, char* argv[]) /* 8. Text(Code segment) */
30 {
31     static int key = 9973 /* 9. Initialized data */
32 }
```

orlan@Orlan: ~/Desktop/Final_OS/Q1

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/>.

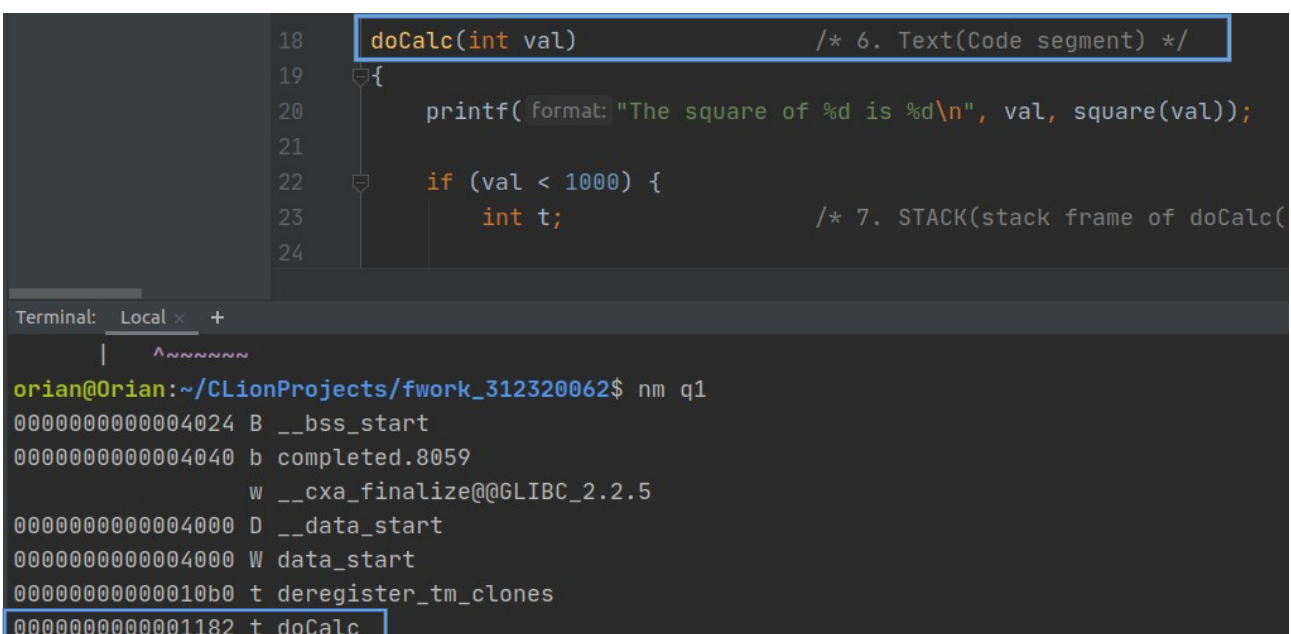
For help, type "help".
Type "apropos word" to search for commands related to "word"...

Reading symbols from q1...
(gdb) break 14
Breakpoint 1 at 0x117d: file q1_312320062.c, line 14.
(gdb) run
Starting program: /home/orlan/Desktop/Final_OS/Q1/q1

Breakpoint 1, square (x=9973) at q1_312320062.c:14
14 return result; /* 5. By register */
(gdb) info r eax
eax 0x5eda679 99460729
(gdb) info local
result = 99460729
(gdb)

As we can see at the screen shot above, we put break point at line 14 in our code and we want to see and look at the eax register and result variable, as you can see they are equal and that's why for sure the value is returned via register.

6. Where is allocated?



```
18 doCalc(int val) /* 6. Text(Code segment) */
19 {
20     printf(format: "The square of %d is %d\n", val, square(val));
21
22     if (val < 1000) {
23         int t; /* 7. STACK(stack frame of doCalc) */
24     }
```

Terminal: Local x +

orlan@Orlan:~/CLionProjects/fwork_312320062\$ nm q1

```
0000000000004024 B __bss_start
0000000000004040 b completed.8059
                w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
00000000000010b0 t deregister_tm_clones
0000000000001182 t doCalc
```

Explanation:

I used the nm command, as we can see doCalc have symbol type of t , which means that doCalc is in Text(Code segment).

7. Where is allocated?

```
17 static void
18 doCalc(int val)                /* 6. Text(Code segment) */
19 {
20     printf("The square of %d is %d\n", val, square(val));
21
22     if (val < 1000) {
23         int t;                  /* 7. STACK(stack frame of doCalc()) */
24 //b 13 00000000
25         t = val * val * val;
26         printf("The cube of %d is %d\n", val, t);
27     }
28 }
29
```



```

00000000000001182 <doCalc>:
1182:    f3 0f 1e fa    endbr64
1186:    55            push    %rbp
1187:    48 89 e5       mov     %rsp,%rbp
118a:    48 83 ec 20    sub     $0x20,%rsp
118e:    89 7d ec       mov     %edi,-0x14(%rbp)
1191:    8b 45 ec       mov     -0x14(%rbp),%eax
1194:    89 c7         mov     %eax,%edi
1196:    e8 ce ff ff ff callq   1169 <square>
119b:    89 c2         mov     %eax,%edx
119d:    8b 45 ec       mov     -0x14(%rbp),%eax
11a0:    89 c6         mov     %eax,%esi
11a2:    48 8d 3d 5b 0e 00 00 lea     0xe5b(%rip),%rdi
11a9:    b8 00 00 00 00 mov     $0x0,%eax
11ae:    e8 ad fe ff ff callq   1060 <printf@plt>
11b3:    81 7d ec e7 03 00 00 cmpl    $0x3e7,-0x14(%rbp)
11ba:    7f 28          jg      11e4 <doCalc+0x62>
11bc:    8b 45 ec       mov     -0x14(%rbp),%eax
11bf:    0f af c0       imul    %eax,%eax
11c2:    8b 55 ec       mov     -0x14(%rbp),%edx
11c5:    0f af c2       imul    %edx,%eax
11c8:    89 45 fc       mov     %eax,-0x4(%rbp)
11cb:    8b 55 fc       mov     -0x4(%rbp),%edx
11ce:    8b 45 ec       mov     -0x14(%rbp),%eax
11d1:    89 c6         mov     %eax,%esi
11d3:    48 8d 3d 42 0e 00 00 lea     0xe42(%rip),%rdi
11da:    b8 00 00 00 00 mov     $0x0,%eax
11df:    e8 7c fe ff ff callq   1060 <printf@plt>
11e4:    90            nop
11e5:    c9            leaveq
11e6:    c3            retq

```

Explanation: Lets look on the box in the screen shot above.

The lines 11b3-11ba will determine if we are going to get into the if condition in the program or jump on it.

At line 11bc :the %eax register get the value of -0x14(%rbp) which hold the value of val(the parameter we got as input to functuin).

At line 11bf we will multiplie the eax register by it self, after this line the eax register will hold the value of val*val.

At line 11c2 we will copy again the value of val(from -0x14(%rbp) to edx register, now edx will hold the value of val.

At line 11c5 we will multiple eax register with edx register and the result well saved into eax register, after this line the eax register will hold the value of val*val*val,
at line 11c8 we will copy the value of eax into -0x4(%rbp).
This is the end of the line “t=val*val*val;” in our program.

After this we can be sure that the variable t is sitting on the stack.
At -0x4(%rbp)
note: (rbp is base pointer which points to the base of the current stack frame).

8. Where is allocated?

```
31 ▶ main(int argc, char* argv[]) /* 8. Text(Code segment) */
32 {
33     static int key = 9973; /* 9. Initialized data */
34     static char mbuf[10240000]; /* 10. Uninitialized data(BSS) */
35     char* p; /* 11. STACK(stack frame of main()) */
36
doCalc

Terminal: Local x +
0000000000003db8 d __init_array_end
0000000000003db0 d __init_array_start
0000000000002000 R _IO_stdin_used
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
0000000000004020 d key.2844
0000000000001290 T __libc_csu_fini
0000000000001220 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
00000000000011e7 T main
```

Explanation:

I used the nm command, as we can see main have symbol type of T , which means that main is in Text(Code segment). Capital t means that main is global.

9. Where is allocated?

```
33 static int key = 9973; /* 9. Initialized data */
34 static char mbuf[10240000]; /* 10. Uninitialized data(BSS) */
35 char* p; /* 11. STACK(stack frame of main()) */
36
main

Terminal: Local x +
0000000000003db0 d __init_array_start
0000000000002000 R _IO_stdin_used
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
0000000000004020 d key.2844
0000000000001290 T __libc_csu_fini
```

Explanation:

I used the nm command, as we can see key have symbol type of d , which means that key is in Initilized data.

10. Where is allocated?

```

34 static char mbuf[10240000]; /* 10. Uninitialized data(BSS) */
35 char* p; /* 11. STACK(stack frame of main()) */
36
37
main
0000000000004020 d key.2844
0000000000001290 T __libc_csu_fini
0000000000001220 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
00000000000011e7 T main
0000000000004060 b mbuf.2845
0000000000004010 D primes

```

Explanation:

I used the nm command, as we can see mbuf have symbol type of b , which means that mbuf is in Uninitiliazed data(BSS).

11. Where is allocated?

```

81 main(int argc, char* argv[]) /* 8. Text(Code segment) */
82 {
83     static int key = 9973; /* 9. Initialized data */
84     static char mbuf[10240000]; /* 10. Uninitialized data(BSS) */
85     char* p; /* 11. Not allocated until p get value. */
86             /* if p had a value he would be allocated on stack frame
87                of main() */

```

Explanation:

```

00000000000011e7 <main>:
11e7: f3 0f 1e fa      endbr64
11eb: 55              push    %rbp
11ec: 48 89 e5        mov     %rsp,%rbp
11ef: 48 83 ec 10     sub     $0x10,%rsp
11f3: 89 7d fc        mov     %edi,-0x4(%rbp)
11f6: 48 89 75 f0     mov     %rsi,-0x10(%rbp)
11fa: 8b 05 20 2e 00 00 mov     0x2e20(%rip),%eax    # 4020 <key.2844>
1200: 89 c7          mov     %eax,%edi
1202: e8 7b ff ff ff  callq   1182 <doCalc>
1207: bf 00 00 00 00  mov     $0x0,%edi
120c: e8 5f fe ff ff  callq   1070 <exit@plt>
1211: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
1218: 00 00 00
121b: 0f 1f 44 00 00  nopl    0x0(%rax,%rax,1)

```

Lets look at the screen shot above.when the compiler move on the code, it see the variable p and remember that he need to consider

it. When the compiler see that there is no use in p within the function the variable optimized out and dont allocate on the stack, if we will initialize the variable p we will see that p is allocated on the stack and didnt optimized out.

Lets explaine the disassembly screen shot:

first 3 lines are to save the rbp and allocate 10 bytes on the stack.

Lines 4,5 is the save of argc,argv on the stack.

Lines 6,7,8 we can see that we take the value of key and move it to eax and then to edi and call the doCalc function.

We passed the line of the char* p in our code and we can see that p isnot allocated on the stack.

P.S: Each compiler doing optimization by it self, the compiler see there is no use in the p variable and thats why it has been optimized out and didnt allocated on the stack.(I used flags to cancel the optimization).

The compilers are smart, if we was using p the compiler would allocate it on the stack frame of main.