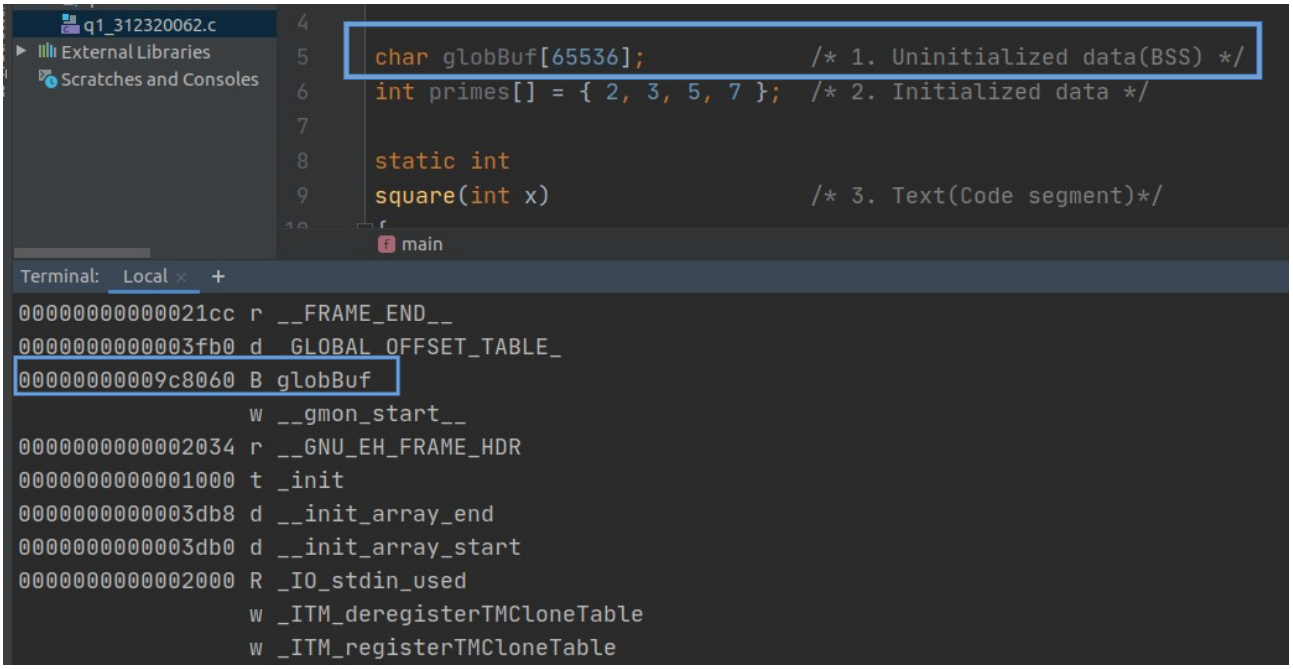
Question 1:

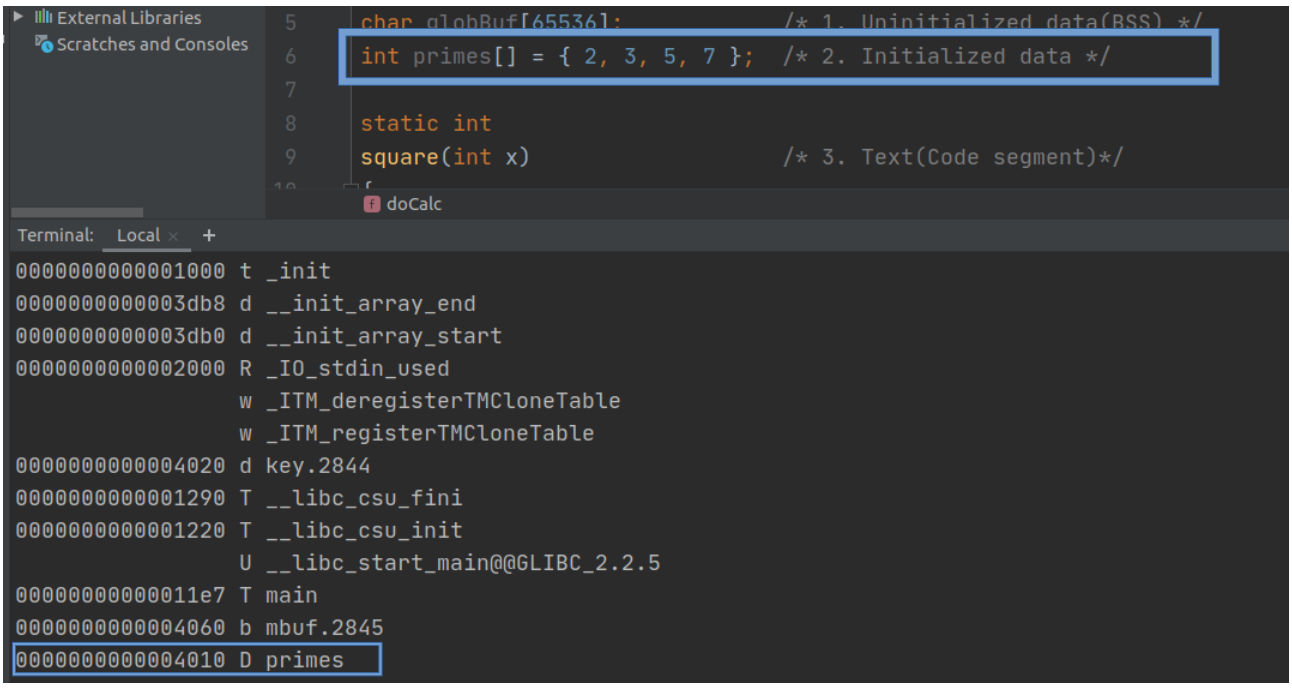
1. Where is allocated?
Answer: Uninitialized data(BSS)



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?

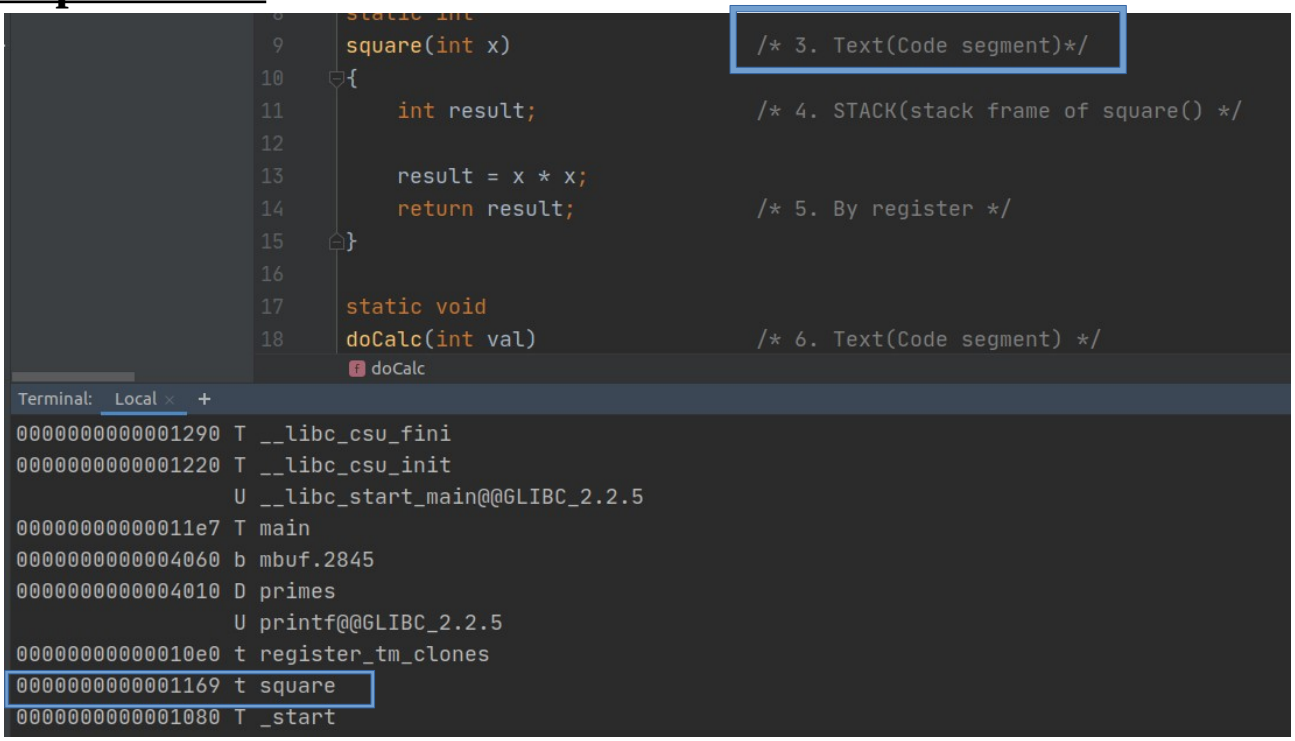


Explanation:

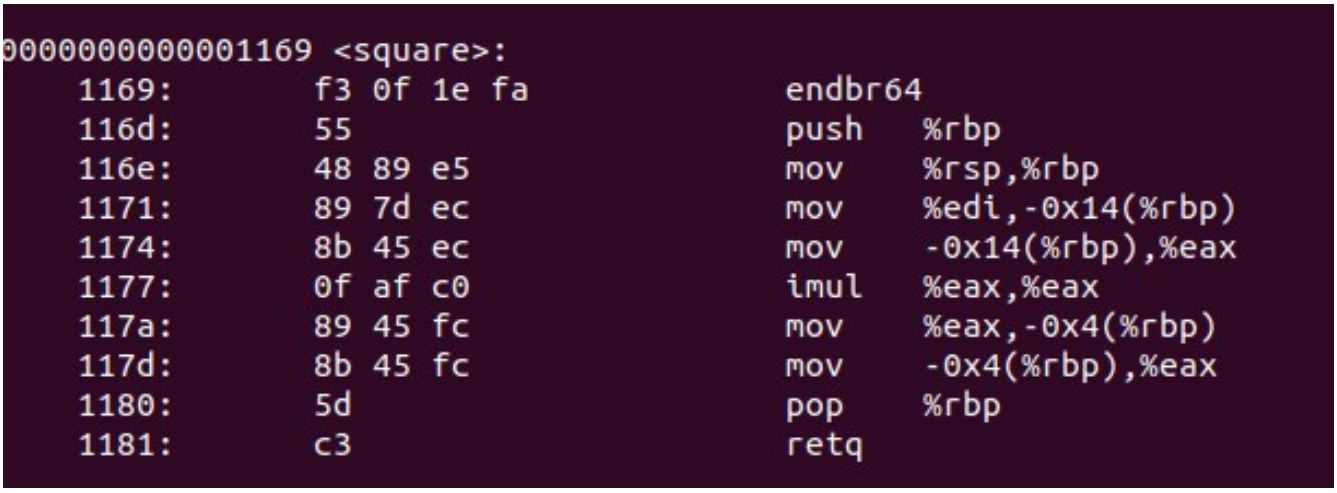
I used the nm command, as we can see primes have symbol type of D , which means that primes is in Initiliazed data.
Capital d means that primes is global variable.

3. Where is allocated?

Explanation:



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).

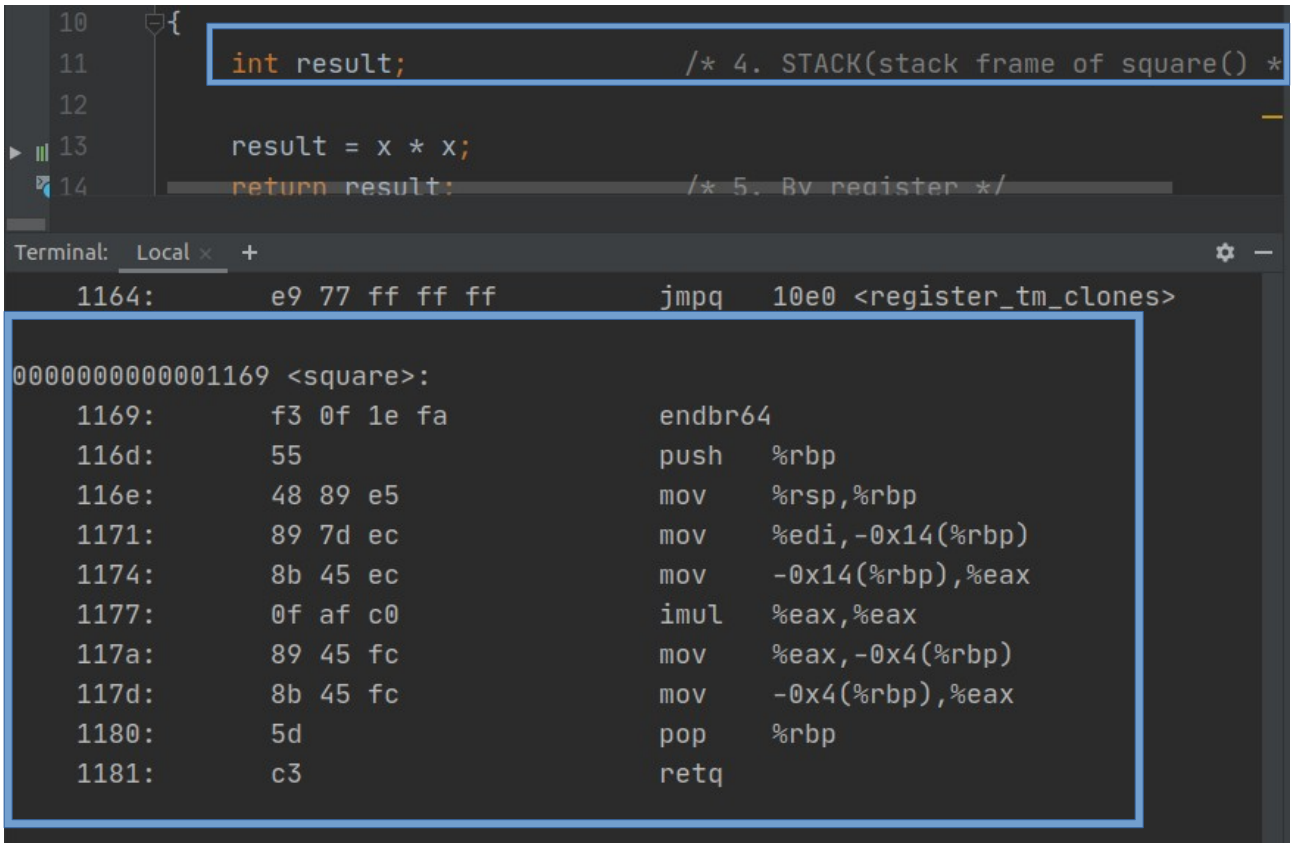


Lets look at the screen shot above and show that all the memory space of the function is in the stack, using objdump of the file.
I will explaine each line:
-push % rbp means that we push old base pointer to the stack(rbp holds the return value ,where we should return after done the function.).

-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 downards.
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?



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.

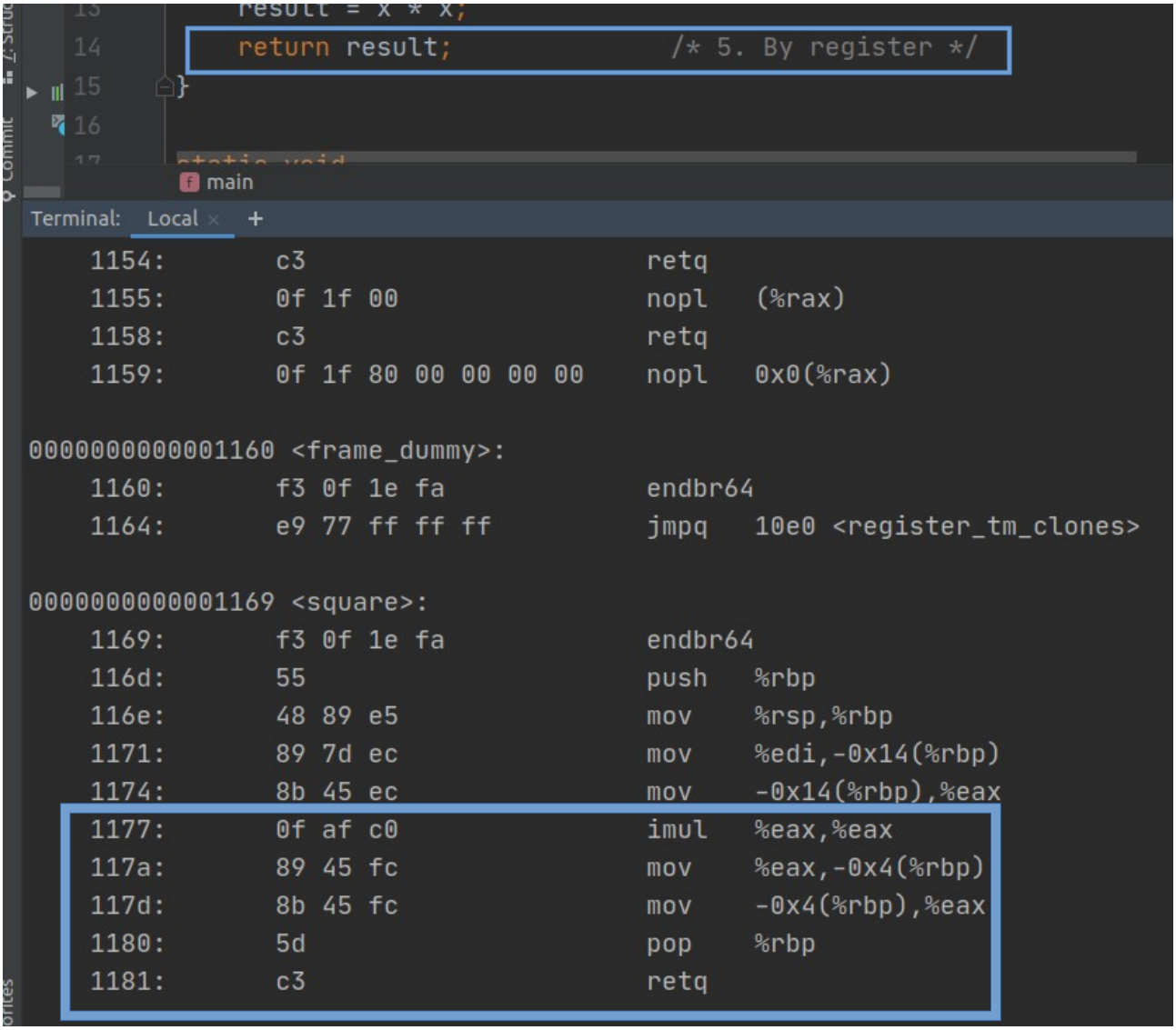
-mov %edi,-0x14(%rbp) push the parameter to function at the bottom of stack frame(at -0x14(%rbp)).

-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 multipie 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?



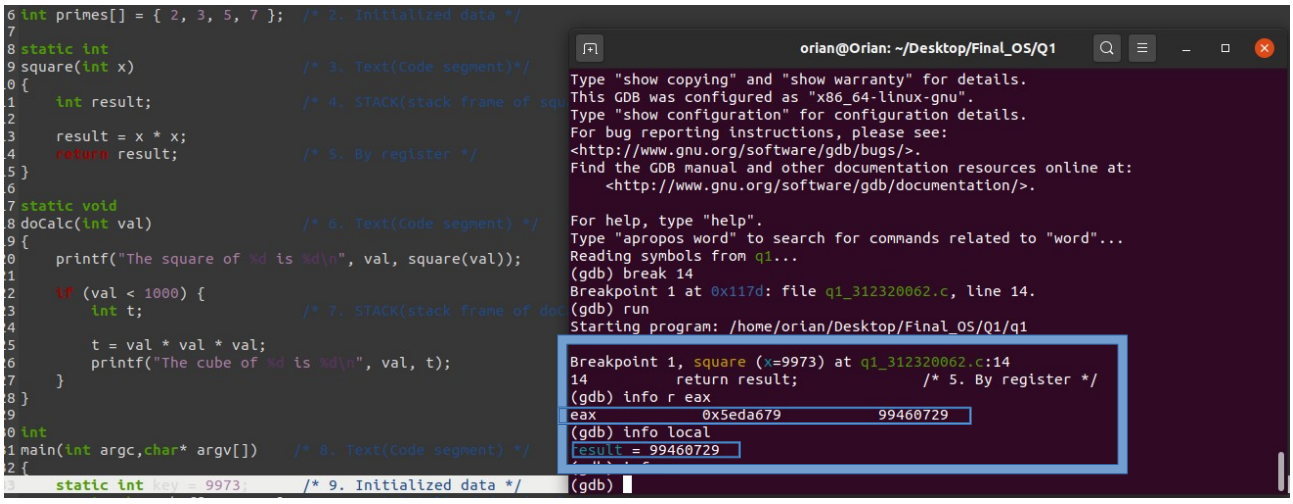
Explanation:

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).

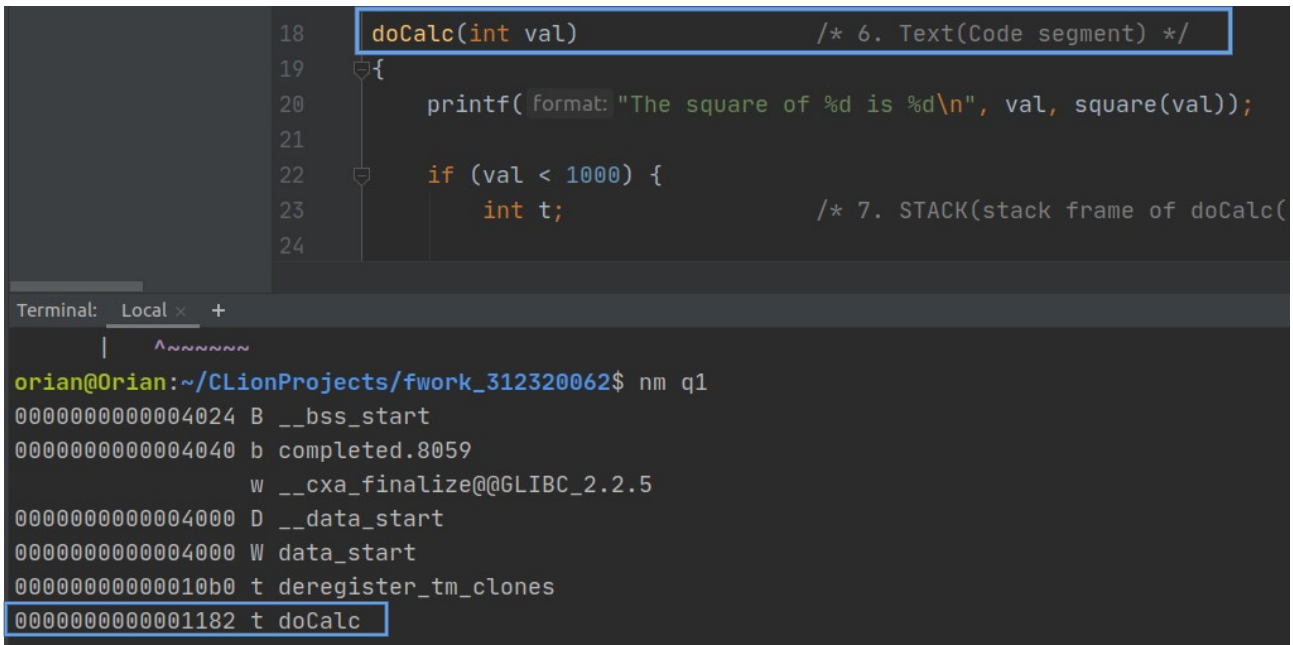
At line 117a we can see that that value from eax register saved into -0x4(%rbp) (in our program its the result =x*x, the eax register has been multiplied by itself.). Lets remember that rbp register point the start of the function on the stack, the stack is

geting bigger when the pointer go down. -0x4 its the result variable memory space.
At line 117d we can see that eax register now have the value of -0x4(%rbp)(we took the value from result into eax register,
as we mentioned above eax register is used to return value from functions.
After this we pop %rbp which means that now the base pointer will be back to where we left when we start doing the function.
Then the retq is return from the function.



As we can see at the screen shot above, we put break point at line 14 in our code and we want see look at eax register and result variable, as you can see they are equals and thats why for sure the value return via register.

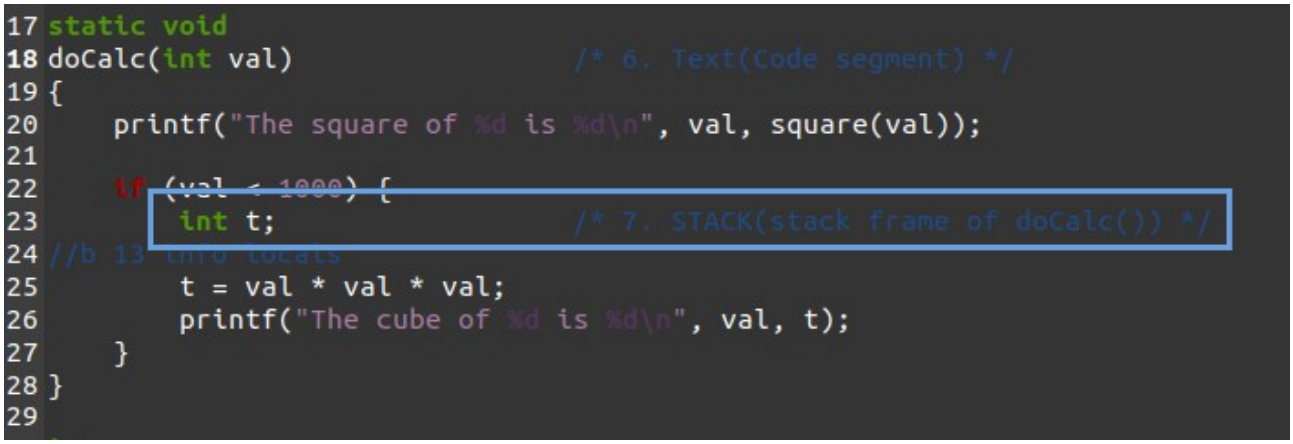
6. Where is allocated?



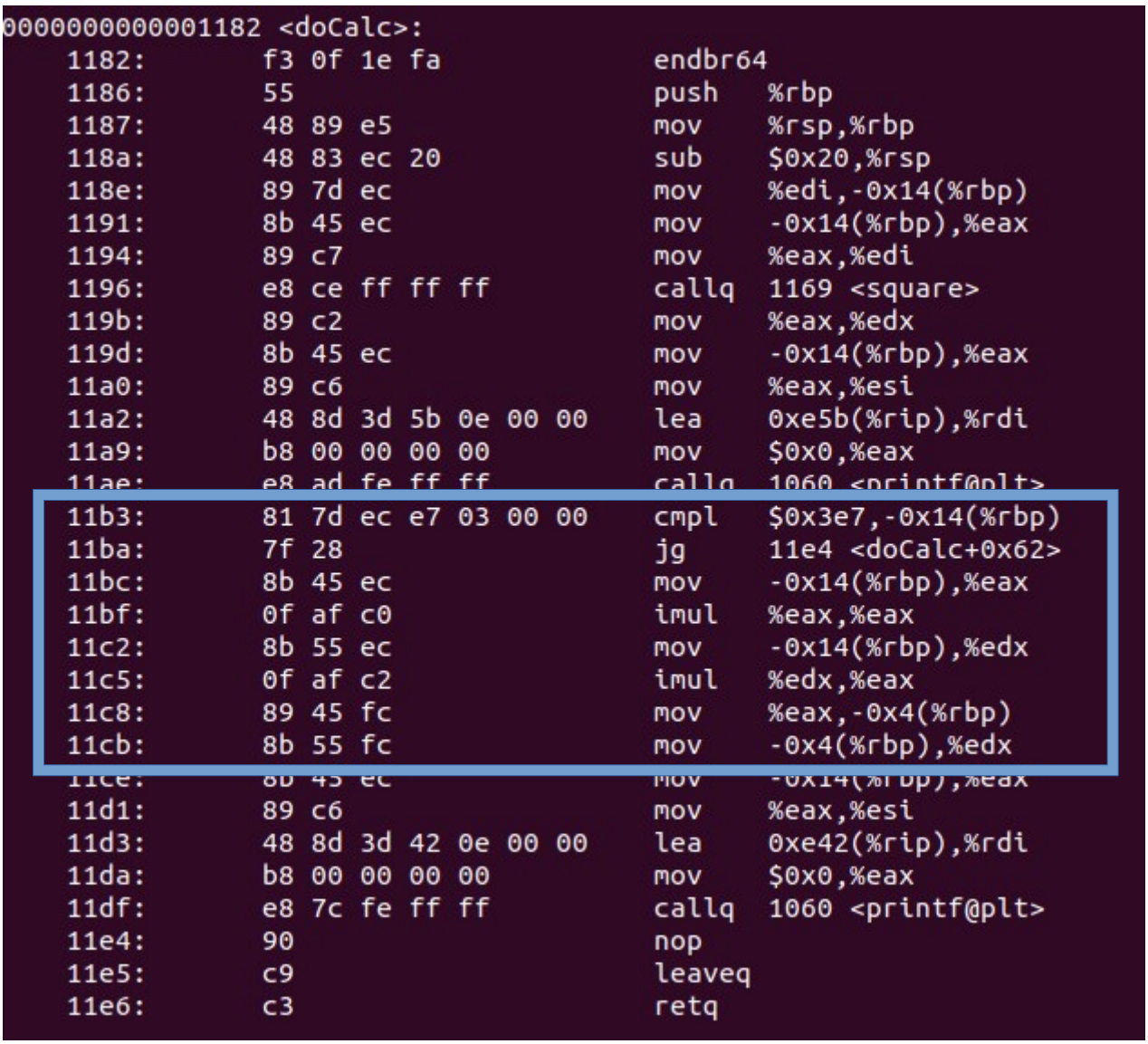
Explanation:

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 info locals
25         t = val * val * val;
26         printf("The cube of %d is %d\n", val, t);
27     }
28 }
29
```



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).

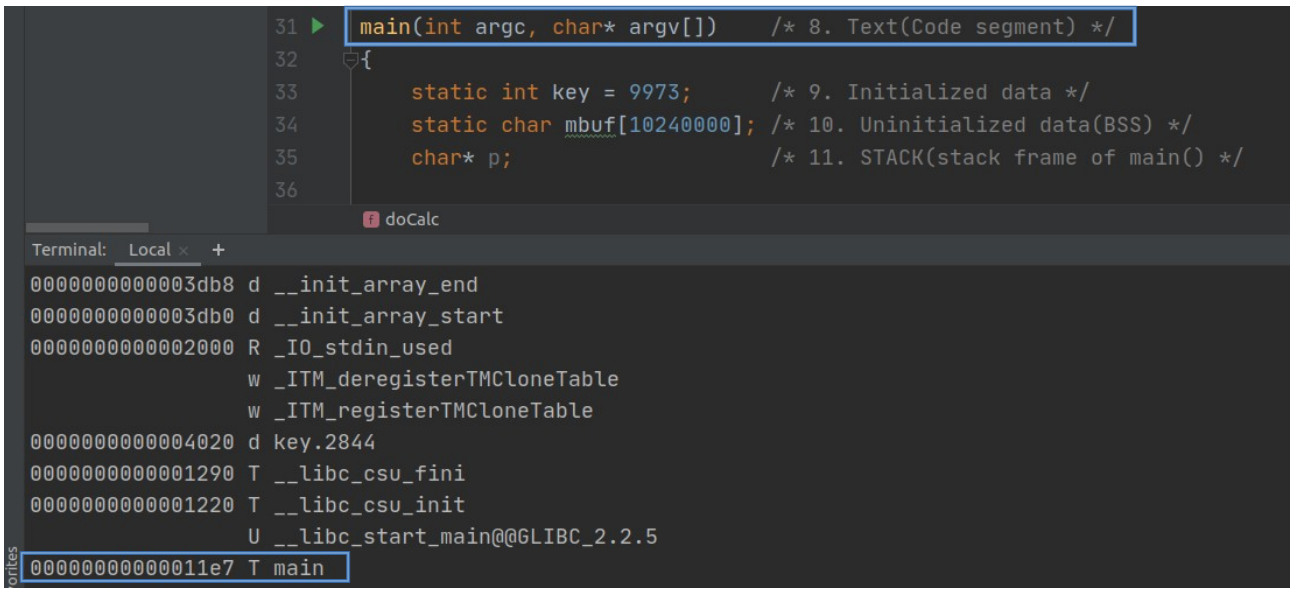
At line 11bf we will multipie the eax register by it self, after this line the eax register will hold the value of val*val.

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 multipie 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).

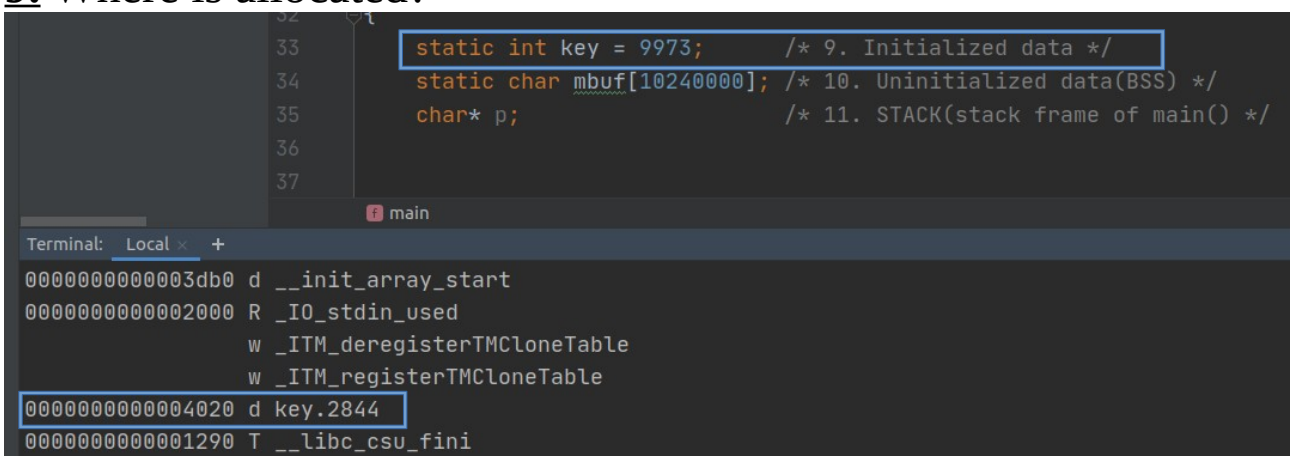
8. Where is allocated?



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?



Explanation:

I used the nm command, as we can see key have symbol type of d , which means that key is in Initiliazed 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
        f main
Terminal:  Local  +
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?

```
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. Not allocated until p get value. */
36                                  /*     if p had a value he would be allocated on stack frame
                                                of main()*/
37
```

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.