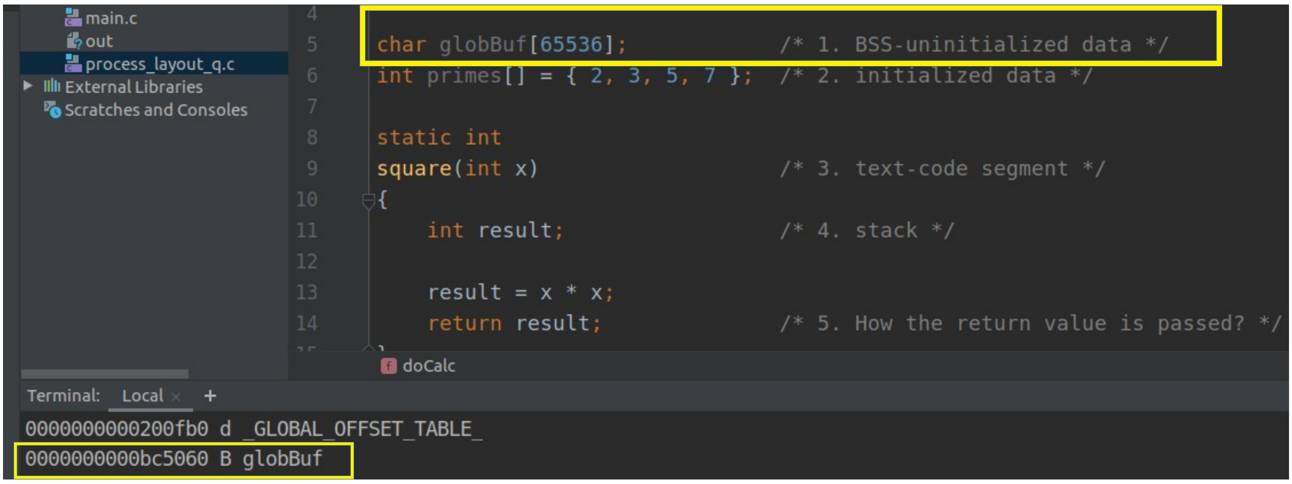
# <u>Question 1:</u>

**1.** **Question:** Where is allocated?  char globBuf[65536];
**Answer:** BSS-uninitialized data
**proof:** we can see that after execute the “nm” command we got the type of “globBuf”  to be B, that means according to “man” that “globBuf” is a global symbol and it’s found in BSS:



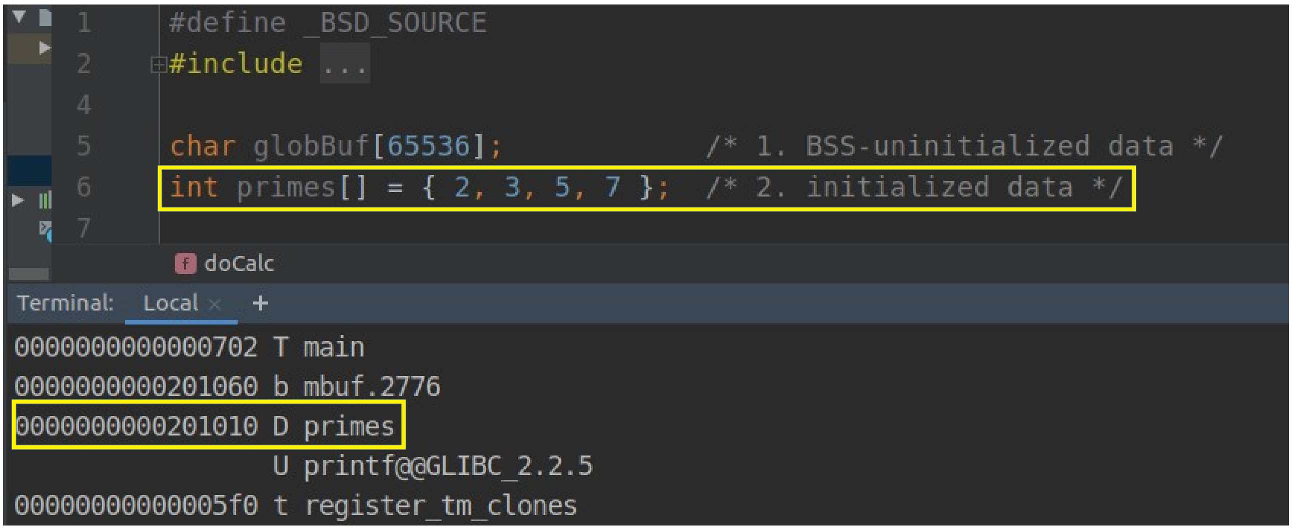
```
char globBuf[65536];            /* 1. BSS-uninitialized data */
int primes[] = { 2, 3, 5, 7 };  /* 2. initialized data */

static int
square(int x)                   /* 3. text-code segment */
{
    int result;                 /* 4. stack */

    result = x * x;
    return result;              /* 5. How the return value is passed? */
```

```
0000000000200fb0 d _GLOBAL_OFFSET_TABLE_
0000000000bc5060 B globBuf
```

**2.** **Question:** Where is allocated? int primes[] = { 2, 3, 5, 7 };
**Answer:** initialized data
**proof:** we can see that after execute the “nm” command we got the type of  “primes”  to be D, that means according to “man” that “primes” is a global  symbol and it’s found in initialized data:

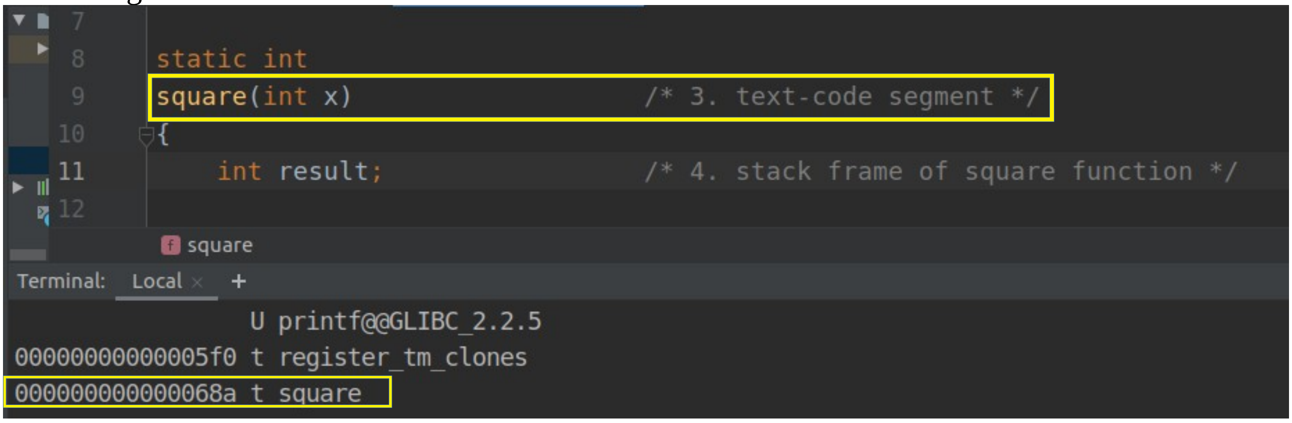


```
#define _BSD_SOURCE
#include ...

char globBuf[65536];            /* 1. BSS-uninitialized data */
int primes[] = { 2, 3, 5, 7 };  /* 2. initialized data */
```

```
0000000000000702 T main
0000000000201060 b mbuf.2776
0000000000201010 D primes
                 U printf@@GLIBC_2.2.5
00000000000005f0 t register_tm_clones
```

**3.** **Question:** Where is allocated? square(int x)
**Answer:** text-code segment (The pointer of function).
**proof:** we can see that after execute the “nm” command we got the type of “square” to be t, that means according to “man” that “square” is a local symbol and it’s found in text-code segment :

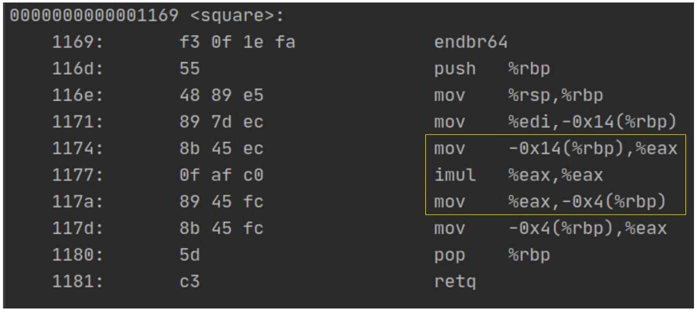


```
static int
square(int x)                  /* 3. text-code segment */
{
    int result;                /* 4. stack frame of square function */

```

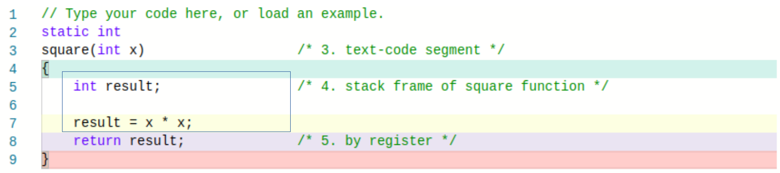
```
                 U printf@@GLIBC_2.2.5
00000000000005f0 t register_tm_clones
000000000000068a t square
```

**4.** **Question:** Where is allocated? int result;
**Answer:** stack frame of square function
**Proof:** we can see that after execute the “objdump -d” command, we received the code of our program in Assembly Language.
We can be sure that **-0x14(%rbp)** hold the value of x variable because the “**mov %edi, - 0x14(%rbp)**” command.
The “result=x*x” command can be seen in the yellow square.
It can be seen that after the calculation of x * x is performed, the value passed by the rbp-base pointer register, which points to the base of the square stack frame.
The **-0x4(%rbp)** signals us that space has been allocated in a square stack frame the size of 4 bytes, which is exactly the size of the result variable.



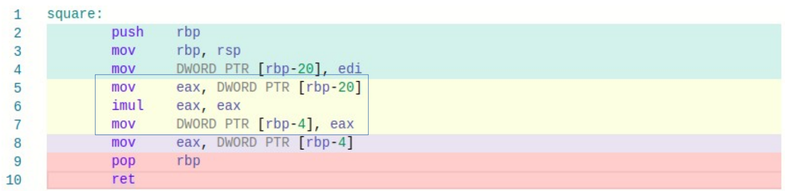
```
0000000000001169 <square>:
    1169:	f3 0f 1e fa          	endbr64
    116d:	55                   	push   %rbp
    116e:	48 89 e5             	mov    %rsp,%rbp
    1171:	89 7d ec             	mov    %edi,-0x14(%rbp)
    1174:	8b 45 ec             	mov    -0x14(%rbp),%eax
    1177:	0f af c0             	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
```

In addition, a screenshot is attached that proves to us that the “result=x*x” command is indeed translated into the yellow square command (by the “godbolt” website).



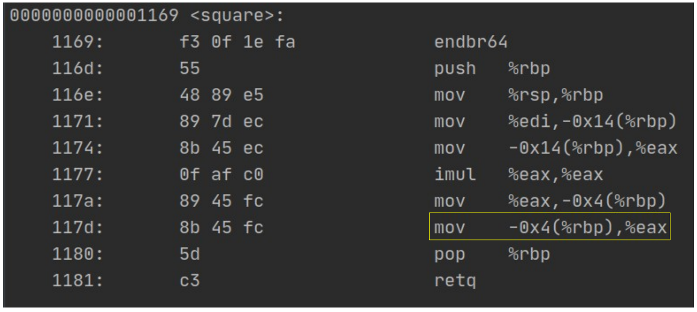
```
// Type your code here, or load an example.
static int
square(int x)                  /* 3. text-code segment */
{
    int result;                /* 4. stack frame of square function */

    result = x * x;
    return result;             /* 5. by register */
}
```



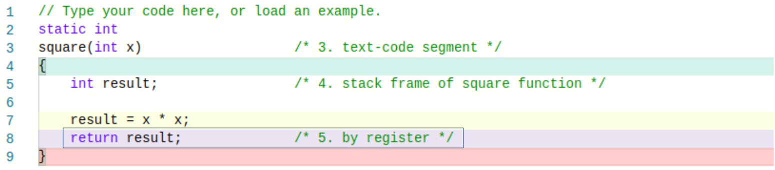
```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

5. **Question:** How the return value is passed? return result;
**Answer:** by register
**Proof:** we can see that after execute the “objdump -d” command, we received the code of our program in Assembly Language.
The “return result” command can be seen in the yellow square.
It can be seen that after the calculation of x * x is performed, it is passed to the eax register which is responsible among other things for the return values from the function, so we can be sure that the result value is returned by register.



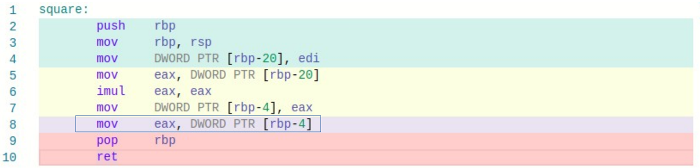
```
0000000000001169 <square>:
    1169:       f3 0f 1e fa             endbr64
    116d:       55                      push   %rbp
    116e:       48 89 e5                mov    %rsp,%rbp
    1171:       89 7d ec                mov    %edi,-0x14(%rbp)
    1174:       8b 45 ec                mov    -0x14(%rbp),%eax
    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
```

In addition, a screenshot is attached that proves to us that the “return result” command is indeed translated into the yellow square command (by the “godbolt” website).



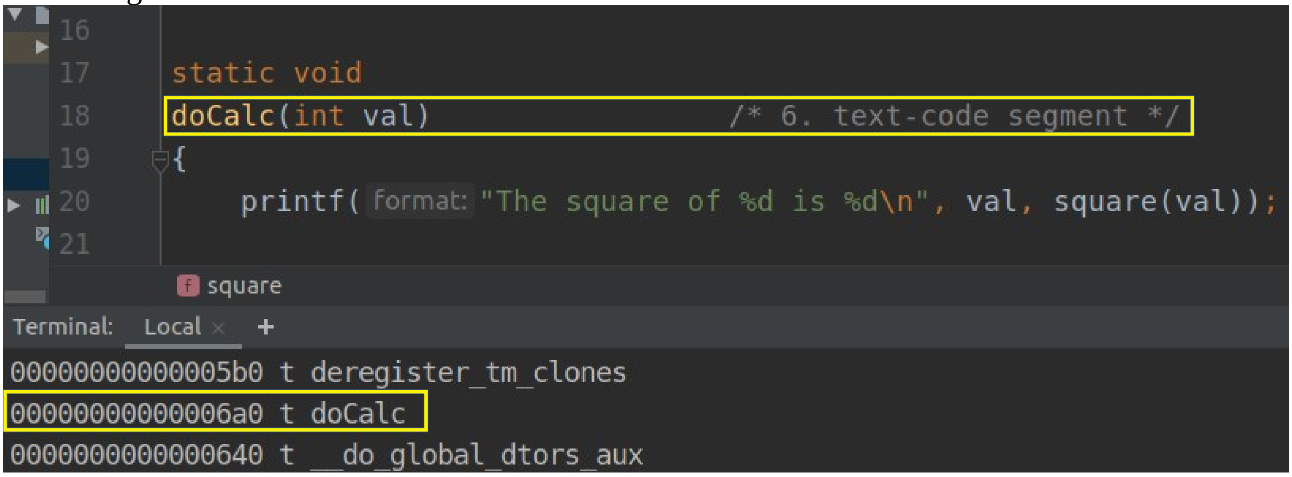
```
// Type your code here, or load an example.
static int
square(int x)                   /* 3. text-code segment */
{
    int result;                 /* 4. stack frame of square function */

    result = x * x;
    return result;              /* 5. by register */
}
```



```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

6. **Question:** Where is allocated? doCalc(int val)
**Answer:** text-code segment (The pointer of function).
**proof:** we can see that after execute the “nm” command we got the type of “doCalc” to be t, that means according to “man” that “doCalc” is a local symbol and it’s found in text-code segment :



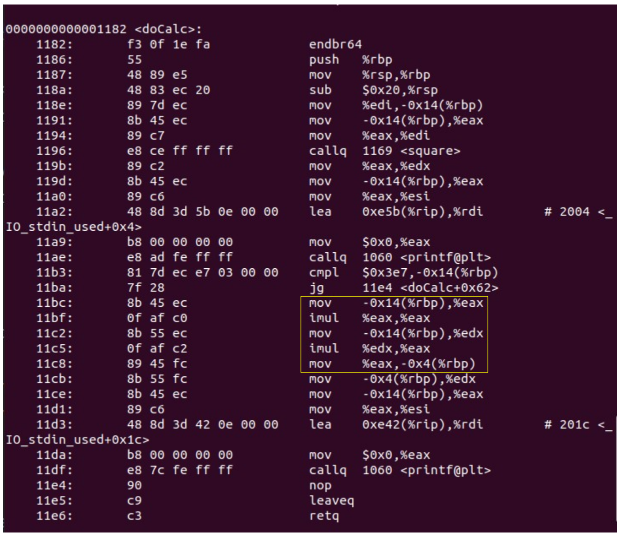
```

static void
doCalc(int val)                    /* 6. text-code segment */
{
    printf( format: "The square of %d is %d\n", val, square(val));

```

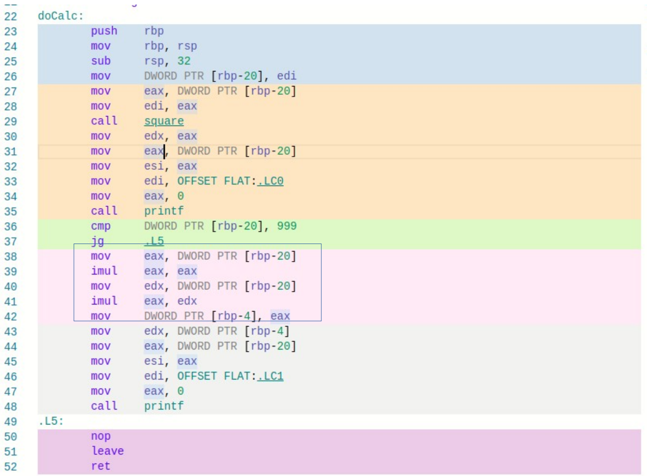
```
00000000000005b0 t deregister_tm_clones
00000000000006a0 t doCalc
0000000000000640 t __do_global_dtors_aux
```

7. **Question:** Where is allocated? int t;
**Answer:** stack frame of doCalc function
**Proof:** we can see that after execute the “objdump -d” command, we received the code of our program in Assembly Language.
The “t=val*val*val” command can be seen in the yellow square.
We can be sure that **-0x14(%rbp)** hold the value of val variable because the “**mov %edi, - 0x14(%rbp)**” command.
This value passed into eax register.
After that, the value of eax multiplied by itself and saved into eax register.
Now, the value of **-0x14(%rbp)** passed into edx register.
After that, the value of eax multiplied by edx and saved into eax register.
At last, eax register hold the value of val*val*val, ans this value copied into -0x4(%rbp).
That prove that t variable is allocated on the doCalc stack frame.

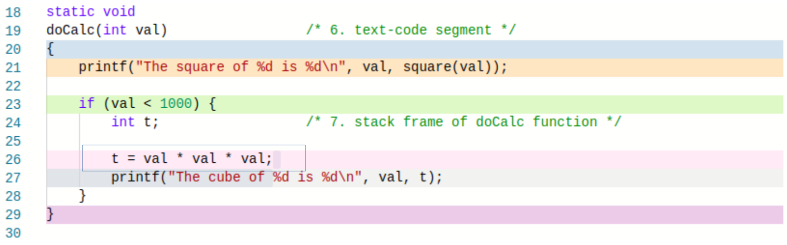


```
0000000000001182 <doCalc>:
    1182:       f3 0f 1e fa             endbr64
    1186:       55                      push   %rbp
    1187:       48 89 e5                mov    %rsp,%rbp
    118a:       48 83 ec 20             sub    $0x20,%rsp
    118e:       89 7d ec                mov    %edi,-0x14(%rbp)
    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        # 2004 <_
IO_stdin_used+0x4>
    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        # 201c <_
IO_stdin_used+0x1c>
    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
```

In addition, a screenshot is attached that proves to us that the “t=val*val*val” command is indeed translated into the yellow square command (by the “godbolt” website).



```
doCalc:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        mov     edi, eax
        call    square
        mov     edx, eax
        mov     eax, DWORD PTR [rbp-20]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        cmp     DWORD PTR [rbp-20], 999
        jg      .L5
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     edx, DWORD PTR [rbp-20]
        imul    eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-20]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
.L5:
        nop
        leave
        ret
```

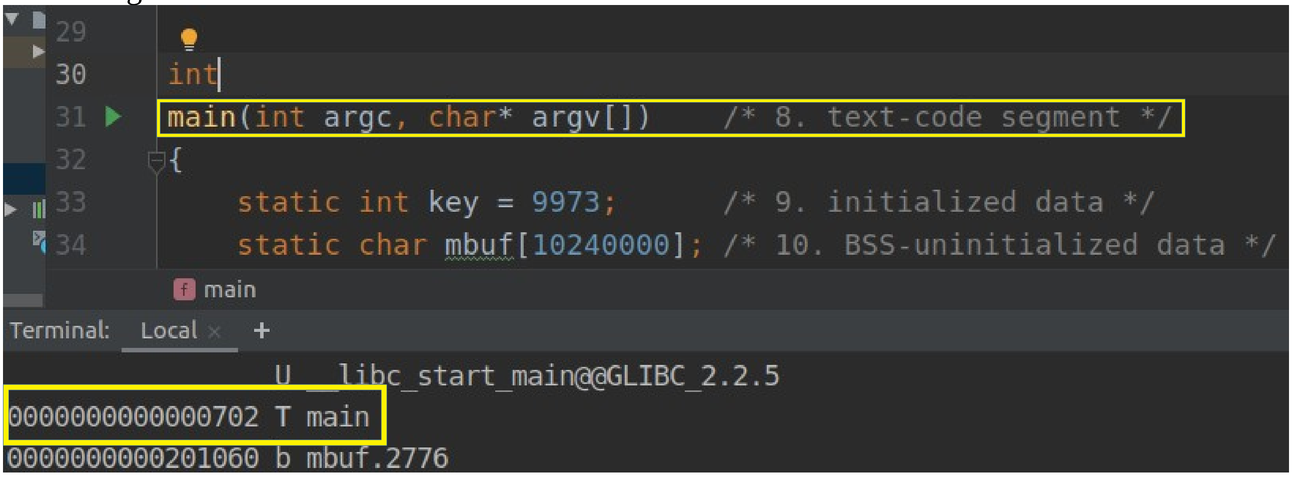


```
static void
doCalc(int val)                 /* 6. text-code segment */
{
    printf("The square of %d is %d\n", val, square(val));

    if (val < 1000) {
        int t;                  /* 7. stack frame of doCalc function */

        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}
```

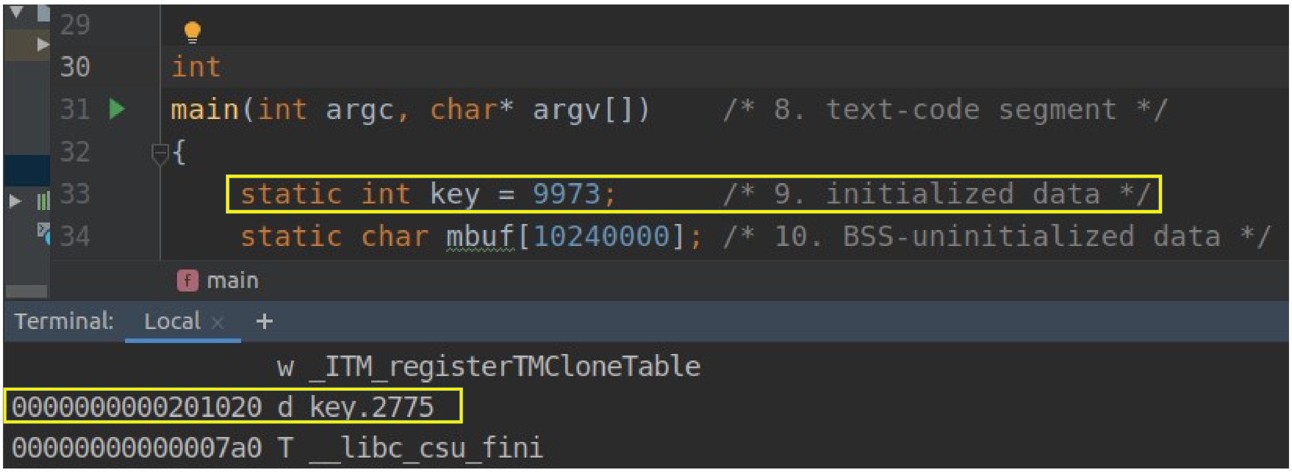
8. **Question:** Where is allocated? main(int argc, char*\ argv[])
**Answer:** text-code segment (The pointer of function).
**Proof:** we can see that after execute the “nm” command we got the type of “main”  to be T, that means according to “man” that “main” is a global symbol and it’s found in text-code segment  :



```
int
main(int argc, char* argv[])     /* 8. text-code segment */
{
    static int key = 9973;       /* 9. initialized data */
    static char mbuf[10240000]; /* 10. BSS-uninitialized data */
```

```
                 U __libc_start_main@@GLIBC_2.2.5
0000000000000702 T main
0000000000201060 b mbuf.2776
```

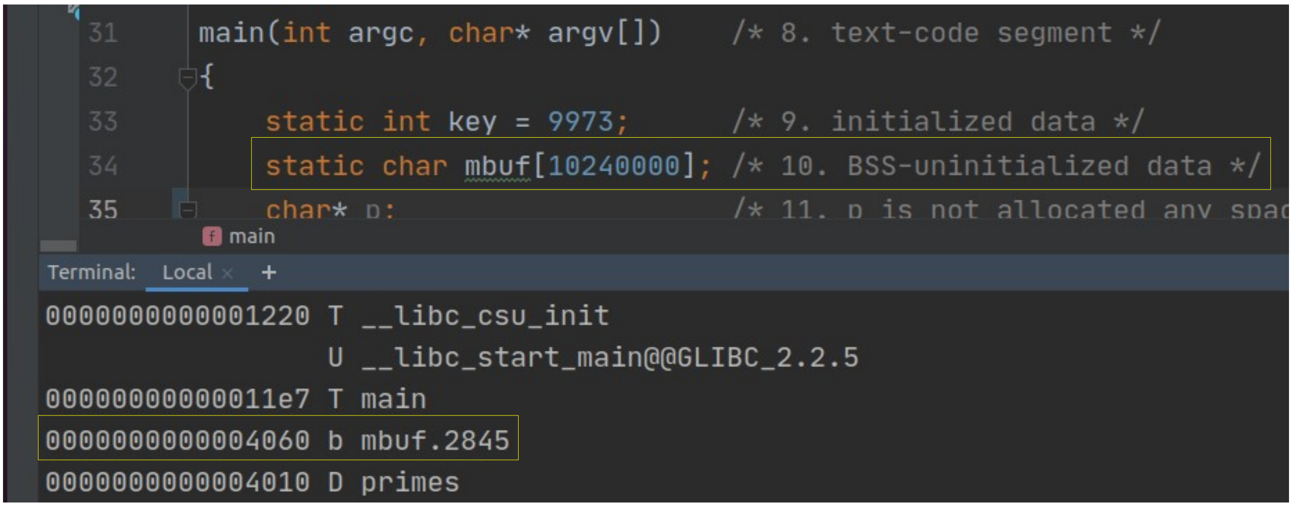
9. **Question:** Where is allocated?  static int key = 9973;
**Answer:** initialized data
**proof:** we can see that after execute the “nm” command we got the type of  “key”  to be d, that means according to “man” that “key” is a local symbol and it’s found in initialized data:



```
int
main(int argc, char* argv[])     /* 8. text-code segment */
{
    static int key = 9973;       /* 9. initialized data */
    static char mbuf[10240000]; /* 10. BSS-uninitialized data */
```

```
                 w _ITM_registerTMCloneTable
0000000000201020 d key.2775
00000000000007a0 T __libc_csu_fini
```

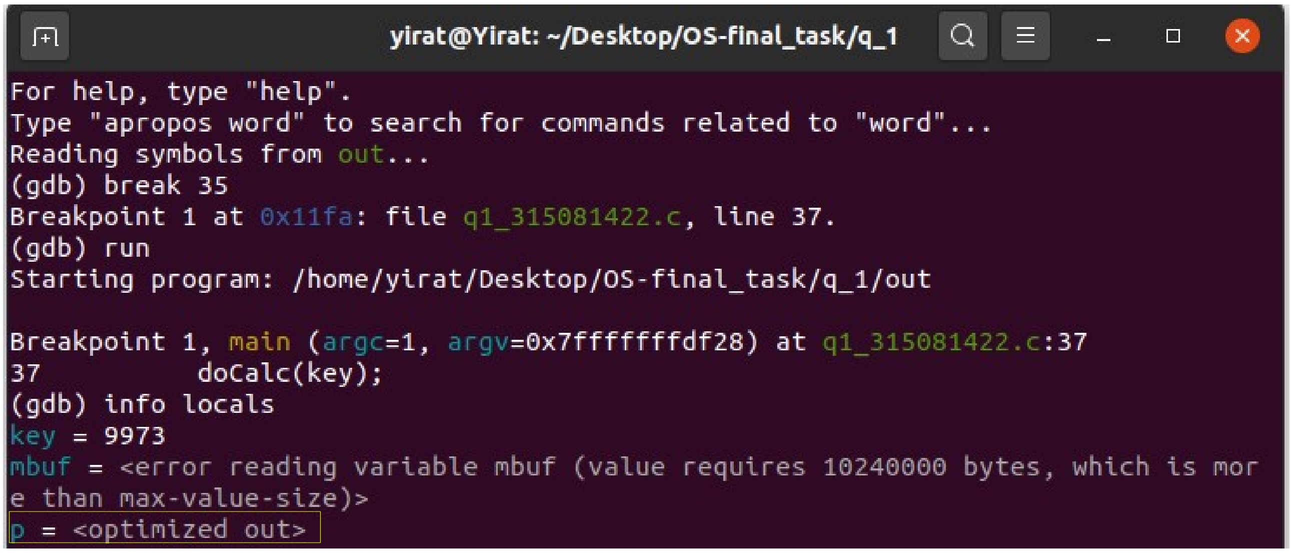
**10.** **Question:** Where is allocated? static char mbuf[10240000];
**Answer:** BSS-uninitialized data
**proof:** we can see that after execute the “nm” command we got the type of “mbuf” to be b, that means according to “man” that “mbuf” is a local symbol and it’s found in BSS :



```
main(int argc, char* argv[])    /* 8. text-code segment */
{
    static int key = 9973;      /* 9. initialized data */
    static char mbuf[10240000]; /* 10. BSS-uninitialized data */
    char* p;                    /* 11. p is not allocated any spac
```

```
0000000000001220 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000000011e7 T main
0000000000004060 b mbuf.2845
0000000000004010 D primes
```

**11.** **Question:** Where is allocated? char* p;
**Answer:** p is not allocated any space in memory because it is not initialized during the program.
If it was initialized, it would be allocated space in the stack frame of main function.(Because the compiler has certain optimizations).
NOTE: There are compilers with certain optimizations that will not allocate space in the memory for a variable that will not be initialized during the program.
In our program we are referring to a compiler without these optimizations.
**Proof:** We can see that we ran with gdb the info locals command that shows the variables that are on the stack.
It can be seen in the yellow square that the variable p was optimized and therefore it is not on the stack, but without the optimization it would have been there.



```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from out...
(gdb) break 35
Breakpoint 1 at 0x11fa: file q1_315081422.c, line 37.
(gdb) run
Starting program: /home/yirat/Desktop/OS-final_task/q_1/out

Breakpoint 1, main (argc=1, argv=0x7fffffffdf28) at q1_315081422.c:37
37          doCalc(key);
(gdb) info locals
key = 9973
mbuf = <error reading variable mbuf (value requires 10240000 bytes, which is mor
e than max-value-size)>
p = <optimized out>
```