

שאלה 1 (25 נק'): מבנה הזכרון של תוכנית בשפת C.
בשאלה זו תתנסו במבנה הזכרון של תוכנית בשפת C הכוללת את הקטעים **heap**, **stack**, **text** וכו'. נתונה התוכנית [program](#). עליכם לענות על כל השאלות בהערות (comments). למטרה זו באפשרותכם להשתמש בכלים הבאים: **objdump**, **nm**, **size** (למדו איך להשתמש בהם).

מה עליכם לבצע:

- (0) למדו איך להשתמש ב-**objdump**, **nm**, **size**. השתמשו ב-**man** או מקורות אחרים לפי בחירתכם. עליכם להכיר את הכלים האלה לפחות ברמה שתאפשר לכם לבצע את המטלה.
- (1) החליפו כל הערה (comment) שיש בה שאלה בתשובה, בתשובה של שורה אחת בתוך [התכנית המקורית](#). כל השאלות ממוספרות. שמרו על אותו מספור בתשובות שלכם.
- (2) יש ליצור קובץ pdf נפרד, שבו תסבירו כל אחת מתשובותיכם. כמו כן הוסיפו פלט של הכלים (הנ"ל) שהשתמשתם בהם, שמאשר את התשובה שלכם. יש להשתמש באותו מספור. לצורך נוחות הבדיקה, אנא העתיקו לפני כל תשובה את השאלה המקורית + שורת הקוד המתאימה מהתוכנית המקורית.

```
1  #define _BSD_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  char globBuf[65536];          /* 1. Where is allocated? */
6  int primes[] = { 2, 3, 5, 7 }; /* 2. Where is allocated? */
7
8  static int
9  square(int x)                 /* 3. Where is allocated? */
10 {
11     int result;               /* 4. Where is allocated? */
12
13     result = x * x;
14     return result;            /* 5. How the return value is passed? */
15 }
16
17 static void
18 doCalc(int val)               /* 6. Where is allocated? */
19 {
20     printf("The square of %d is %d\n", val, square(val));
21
22     if (val < 1000) {
23         int t;                /* 7. Where is allocated? */
24
25         t = val * val * val;
26         printf("The cube of %d is %d\n", val, t);
27     }
28 }
29
30 int
31 main(int argc, char* argv[]) /* 8. Where is allocated? */
32 {
33     static int key = 9973;     /* 9. Where is allocated? */
34     static char mbuf[1024000]; /* 10. Where is allocated? */
35     char* p;                  /* 11. Where is allocated? */
36
37
38     doCalc(key);
39
40     exit(EXIT_SUCCESS);
41 }
```

Where I learned from:

<https://stackoverflow.com/questions/14588767/where-in-memory-are-my-variables-stored-in-c>



You got some of these right, but whoever wrote the questions tricked you on at least one question:

215



- global variables -----> data (correct)
- static variables -----> data (correct)
- constant data types -----> code and/or data. Consider string literals for a situation when a constant itself would be stored in the data segment, and references to it would be embedded in the code
- local variables(declared and defined in functions) -----> stack (correct)
- variables declared and defined in `main` function -----> ~~heap~~ also stack (the teacher was trying to trick you)
- pointers(ex: `char *arr`, `int *arr`) -----> ~~heap~~ data or stack, depending on the context. C lets you declare a global or a `static` pointer, in which case the pointer itself would end up in the data segment.
- dynamically allocated space(using `malloc`, `calloc`, `realloc`) -----> ~~stack~~ heap

It is worth mentioning that "stack" is officially called "automatic storage class".

<https://linux.die.net/man/1/nm>

"B"

"b"

The symbol is in the uninitialized data section (known as BSS).

"D"

"d"

The symbol is in the initialized data section.

"T"

"t"

The symbol is in the text (code) section.

<https://linux.die.net/man/1/objdump>

The other common output format, usually seen with ELF based files, looks like this:

```
00000000 l    d  .bss  00000000 .bss
00000000 g    .text 00000000 fred
```

Here the first number is the symbol's value (sometimes referred to as its address). The next field is actually a set of characters and spaces indicating the flag bits that are set on the symbol. These characters are described below. Next is the section with which the symbol is associated or **ABS** if the section is absolute (ie not connected with any section), or **UND** if the section is referenced in the file being dumped, but not defined there.

After the section name comes another field, a number, which for common symbols is the alignment and for other symbols is the size. Finally the symbol's name is displayed.

The flag characters are divided into 7 groups as follows:

"l"

"g"

"u"

"!"

The symbol is a local (l), global (g), unique global (u), neither global nor local (a space) or both global and local (!). A symbol can be neither local or global for a variety of reasons, e.g., because it is used for debugging, but it is probably an indication of a bug if it is ever both local and global. Unique global symbols are a GNU extension to the standard set of ELF symbol bindings. For such a symbol the dynamic linker will make sure that in the entire process there is just one symbol with this name and type in use.

"O"

The symbol is the name of a function (F) or a file (f) or an object (O) or just a normal symbol (a space).

<https://stackoverflow.com/questions/6666805/what-does-each-column-of-objdumps-symbol-table-mean>



COLUMN ONE: the symbol's value

69

COLUMN TWO: a set of characters and spaces indicating the flag bits that are set on the symbol. There are seven groupings which are listed below:



group one: (l,g,,!) local, global, neither, both.



group two: (w,) weak or strong symbol.

group three: (C,) symbol denotes a constructor or an ordinary symbol.

group four: (W,) symbol is warning or normal symbol.

group five: (I,) indirect reference to another symbol or normal symbol.

group six: (d,D,) debugging symbol, dynamic symbol or normal symbol.

group seven: (F,f,O,) symbol is the name of function, file, object or normal symbol.

COLUMN THREE: the section in which the symbol lives, *ABS* means not associated with a certain section

COLUMN FOUR: the symbol's size or alignment.

COLUMN FIVE: the symbol's name.

If you want additional information try you man page ;-) or the following links:

<http://manpages.ubuntu.com/manpages/intrepid/man1/objdump.1.html> and

<http://sourceware.org/binutils/docs/binutils/objdump.html>

these follow

... ..

... ..

```
char globBuf[65536]; /* 1. Where is allocated? Uninitialized data segment – bss – global */
```

1) globBuf - Uninitialized data segment – bss - global

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "globBuf"
000000000009c8060 B globBuf
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "globBuf"
000000000009c8060 g 0 .bss 0000000000010000 globBuf
shilo@shilo-VirtualBox:~/Desktop/os_final$
```

```
int primes[] = { 2, 3, 5, 7 }; /* 2. Where is allocated? Initialized data segment. – data – global */
```

2) primes - Initialized data segment. – data - global

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "primes"
000000000004010 D primes
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "primes"
000000000004010 g 0 .data 0000000000000010 primes
shilo@shilo-VirtualBox:~/Desktop/os_final$
```

```
square(int x) /* 3. Where is allocated? Allocated in frame for square() – text – local*/
```

3) square() - Allocated in frame for square() – text - local

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "square"
000000000001169 t square
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "square"
000000000001169 l F .text 0000000000000019 square
shilo@shilo-VirtualBox:~/Desktop/os_final$
```

```
int result; /* 4. Where is allocated? stack*/
```

4) result – stack

```
return result; /* 5. How the return value is passed? Return value passed via register*/
```

5) return value - Return value passed via register

```
doCalc(int val) /* 6. Where is allocated? Allocated in frame for doCalc() – text – local*/
```

6) doCalc() - Allocated in frame for doCalc() – text - local

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "doCalc"
000000000001182 t doCalc
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "doCalc"
000000000001182 l F .text 0000000000000065 doCalc
shilo@shilo-VirtualBox:~/Desktop/os_final$
```

```
int t; /* 7. Where is allocated? stack*/
```

7) t - stack

```
main(int argc, char* argv[]) /*8. Where is allocated? Allocated in frame for main() – text – global*/
```

8) main() - Allocated in frame for main() – text - global

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "main"
U __libc_start_main@@GLIBC_2.2.5
0000000000011e7 T main
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "main"
0000000000000000 F *UND* 0000000000000000 __libc_start_main@@GLIBC_2.2.5
0000000000011e7 g F .text 000000000000002a main
shilo@shilo-VirtualBox:~/Desktop/os_final$
```

```
static int key = 9973; /*9. Where is allocated? Initialized data segment – data – local */
```

9) key - Initialized data segment – data - local

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "key"
0000000000004020 d key.2844
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "key"
0000000000004020 l 0 .data 0000000000000004 key.2844
```

```
static char mbuf[10240000]; /*10. Where is allocated? Uninitialized data segment – bss – local */
```

10) mbuf – Uninitialized data segment – bss - local

```
shilo@shilo-VirtualBox:~/Desktop/os_final$ nm process_layout_q | grep "mbuf"
0000000000004060 b mbuf.2845
shilo@shilo-VirtualBox:~/Desktop/os_final$ objdump -x process_layout_q | grep "mbuf"
0000000000004060 l 0 .bss 0000000000009c4000 mbuf.2845
```

```
char* p; /*11. Where is allocated? Allocated in frame for main() – Uninitialized */
```

11) p - Allocated in frame for main() - Uninitialized

(I adjusted the make file, so the name of file has changed.)

Extra explanation for in functions variables.

```
shilo@shilo-VirtualBox:~/Desktop/os_final/os_final_handing/task_1$ objdump -d q1_302537394
```

```
q1_302537394: file format elf64-x86-64
```

```
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

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
```