

ENPM691 Homework 03: Address Layout and Memory Segmentation in C

Kalpesh Bharat Parmar
M.Eng, Cybersecurity
University of Maryland, College Park
██████████

UMD Directory ID ██████ Course and section - ENPM691 0101

Abstract—This paper analyzes the placement of variables in memory when executing a C program, using both direct execution and debugger inspection. The program demonstrates how different variable types such as local, global, static, and heap variables are allocated into specific memory segments including the stack, heap, .data, and .bss. The analysis combines runtime results, debugger output, and assembly inspection to illustrate the underlying memory model of a 32-bit Linux environment.

I. PURPOSE

The purpose of this assignment is to gain practical understanding of memory segmentation in C. By running and debugging the given program `address_layout.c`, the relationship between variable types and memory regions is explored. The exercise emphasizes variable storage locations, debugger usage, and interpretation of assembly instructions.

II. METHOD

The program was compiled and executed on a 32-bit Linux machine. The following compilation command was used:

```
gcc -m32 -g address_layout.c -o address_layout
```

Listing 1. Compilation Command

The flags have the following roles:

- `-m32`: Ensures compilation for a 32-bit environment. This is necessary to match the assignment requirements and observe 32-bit memory addressing.
- `-g`: Includes debugging symbols in the executable. This enables GDB to display variable names, line numbers, and addresses in source-level context [1].
- `-o address_layout`: Specifies the output executable file name.

Execution was performed in two ways:

- 1) Direct execution of the compiled program to print variable addresses.
- 2) Running under the GNU Debugger (GDB) to inspect variable addresses using `print` commands.

Additionally, the assembly code of `main()` was extracted using:

```
(gdb) disassemble main
```

III. RESULTS

A. Program Execution

The first experiment executed the program directly. The addresses printed are shown in Fig. 1.

```
File Actions Edit View Help
(kp@kali)-[~/Desktop/ENPM691/assignment3]
$ gcc -m32 -g address_layout.c -o address_layout
(kp@kali)-[~/Desktop/ENPM691/assignment3]
$ ./address_layout
Local var 1 address: 0xfc46424
Local var 2 address: 0xfc46420
Heap var 1 address: 0x57a481a0
Heap var 2 address: 0x57a48210
Global (uninit) var 1 address: 0x56645020
Global (uninit) var 2 address: 0x56645024
Static Local var 1 address: 0x56645028
Static Local var 2 address: 0x5664502c
Global var 1 address: 0x56645018
Global var 2 address: 0x5664501c
(kp@kali)-[~/Desktop/ENPM691/assignment3]
$
```

Fig. 1. Program execution output showing addresses of variables.

Sample output:

```
Local var 1 address: 0xffc46424
Local var 2 address: 0xffc46420
Heap var 1 address: 0x57a481a0
Heap var 2 address: 0x57a48210
Global (uninit) var 1 address: 0x56645020
Global (uninit) var 2 address: 0x56645024
Static Local var 1 address: 0x56645028
Static Local var 2 address: 0x5664502c
Global var 1 address: 0x56645018
Global var 2 address: 0x5664501c
```

B. Debugger Execution

The second experiment used GDB. The variable addresses are summarized in Fig. 2.

Representative outputs:

```
(gdb) print &local_var_1
$1 = (int *) 0xfffffcee4
(gdb) print &global_var_1
$3 = (int *) 0x56559018 <global_var_1>
(gdb) print &static_var_1
$7 = (int *) 0x56559028 <static_var_1>
```

The screenshot shows the GDB debugger interface. The command line at the bottom has the following text:

```

File Actions Edit View Help
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 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; for details.
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:
<https://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 ./address_layout ...
(gdb) break main
Breakpoint 1 at 0x11ba: file address_layout.c, line 12.
(gdb) run
Starting program: /home/kp/Desktop/ENPM691/assignment3/address_layout
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at address_layout.c:12
12      int local_var_1 = 0;
(gdb) print &local_var_1
$1 = (int *) 0xfffffe00
(gdb) print &local_var_2
$2 = (int *) 0xfffffe00
(gdb) print &global_var_1
$3 = (int *) 0x6559018 <global_var_1>
(gdb) print &global_var_2
$4 = (int *) 0x6559024 <global_var_2>
(gdb) print &global_uninit_var_1
$5 = (int *) 0x6559028 <global_uninit_var_1>
(gdb) print &global_uninit_var_2
$6 = (int *) 0x655902c <global_uninit_var_2>
(gdb) print &static_var_1
$7 = (int *) 0x6559028 <static_var_1>
(gdb) print &static_var_2
$8 = (int *) 0x655902c <static_var_2>
(gdb) print ptr_1
$9 = (int *) 0x0
(gdb) print ptr_2
$10 = (int *) 0x0
(gdb)

```

Fig. 2. Debugger output showing variable addresses using `print` command.

C. Assembly Inspection

The third experiment disassembled the `main` function. Excerpt shown in Fig. 3.

The screenshot shows the GDB debugger interface. The command line at the bottom has the following text:

```

File Actions Edit View Help
(gdb) disassemble main
Dump of assembler code for function main:
0x5655619d <>0:    lea    0x4(%esp),%ecx
0x565561a1 <>4:    and   $0xffffffff,%f0,%esp
0x565561a4 <>7:    push  -0x4(%ecx)
0x565561a7 <>10:   push  %ebp
0x565561a8 <>11:   mov   %esp,%ebp
0x565561a9 <>13:   push  %ebx
0x565561a9 <>14:   push  %ecx
0x565561a9 <>15:   sub   $0x10,%esp
0x565561af <>18:   call  0x565560a0 <_x86.get_pc_thunk.bx>
0x565561a4 <>23:   add   $0x2e40,%esp
0x565561a8 <>29:   movl $0x0,-0x14(%ebp)
0x565561c1 <>36:   movl $0x0,-0x18(%ebp)
0x565561c8 <>42:   sub   $0xc,%esp
0x565561c9 <>46:   push  $0x64
0x565561cd <>48:   call  0x56556050 <malloc@plt>
0x565561d2 <>53:   add   $0x10,%esp
0x565561d5 <>56:   mov   %eax,-0xc(%ebp)
0x565561d8 <>59:   sub   $0xc,%esp
0x565561d9 <>62:   push  $0x64
0x565561d9 <>64:   call  0x56556050 <malloc@plt>
0x565561e2 <>69:   add   $0x10,%esp
0x565561e5 <>72:   mov   %eax,-0x10(%ebp)
0x565561e8 <>75:   sub   $0x8,%esp
0x565561eb <>78:   lea   -0x14(%ebp),%eax
0x565561e1 <>81:   push  %eax
0x565561ef <>82:   lea   -0x1fec(%ebp),%eax
0x565561f5 <>88:   push  %eax
0x565561f6 <>89:   call  0x56556040 <printf@plt>
0x565561f7 <>94:   add   $0x10,%esp
0x565561f7 <>97:   sub   $0x8,%esp
0x56556201 <>100:  lea   -0x18(%ebp),%eax
0x56556204 <>103:  push  %eax
0x56556205 <>104:  lea   -0x1fd3(%ebp),%eax
0x5655620b <>110:  push  %eax
0x5655620c <>111:  call  0x56556040 <printf@plt>
0x56556211 <>116:  add   $0x10,%esp
0x56556214 <>119:  sub   $0x8,%esp
0x56556217 <>122:  push  -0xc(%ebp)
0x5655621a <>125:  lea   -0x1fba(%ebp),%eax
0x56556220 <>131:  push  %eax
0x56556221 <>132:  call  0x56556040 <printf@plt>
0x56556226 <>137:  add   $0x10,%esp
0x56556229 <>140:  sub   $0x8,%esp
0x5655622c <>143:  push  -0x10(%ebp)

```

Fig. 3. Disassembly of `main()` showing variable initialization and calls to `malloc`.

Example instructions:

`0x565561ba <+29>: movl $0x0,-0x14(%ebp)`

`0x565561cd <+48>: call 0x56556050 <malloc@plt>`
`0x565561ee <+81>: push %eax ; load local varia`

These instructions demonstrate stack-based addressing (%ebp offsets) for locals and external calls for heap allocation.

IV. DISCUSSION

The results demonstrate a consistent memory segmentation pattern across both normal execution and debugger inspection:

- **Local Variables:** Stored on the **stack**. The high memory addresses (0xffff...) confirm the downward-growing stack structure [2].
- **Heap Variables:** Allocated by `malloc`, observed in mid-range addresses (0x57a4...). Each call to `malloc` returns a new pointer from the upward-growing heap.
- **Global Initialized Variables:** (`global_var_1`, `global_var_2`) stored in the **.data** segment. Their addresses are close together.
- **Global Uninitialized Variables:** (`global_uninit_var_1`, `global_uninit_var_2`) stored in the **.bss** segment, reserved for zero-initialized storage.
- **Static Locals:** Though declared inside `main`, static locals (`static_var_1`, `static_var_2`) also reside in the **.data** segment, behaving like globals in storage but with limited scope.

Assembly inspection reinforces these findings. Local variables were manipulated via offsets from the base pointer (%ebp), confirming stack allocation. Global and static variables used absolute addresses in the **.data** region. Heap variables originated from `malloc` calls, with their returned addresses placed into stack slots.

Although absolute addresses varied between program execution and debugging runs due to address space layout randomization (ASLR), the relative placement of variables into stack, heap, data, and bss segments remained consistent.

V. APPENDIX

A. Code Listing

```

1 #include <stdio.h>
2 #include <malloc.h>
3
4 int global_var_1 = 0;
5 int global_var_2 = 0;
6
7 int global_uninit_var_1;
8 int global_uninit_var_2;
9
10 int main() {
11     int local_var_1 = 0;
12     int local_var_2 = 0;
13
14     static int static_var_1 = 0;
15     static int static_var_2 = 0;
16
17     int *ptr_1 = malloc(100);
18     int *ptr_2 = malloc(100);
19
20     printf("Local_var_1_address:%p\n", &local_var_1);
21     printf("Local_var_2_address:%p\n", &local_var_2);
22
23     printf("Heap_var_1_address:%p\n", ptr_1);

```

```

24 printf("Heap_var_2_address:%p\n", ptr_2);
25
26 printf("Global_(uninit)_var_1_address:_%p\n", &
27     global_uninit_var_1);
28 printf("Global_(uninit)_var_2_address:_%p\n", &
29     global_uninit_var_2);
30
31 printf("Static_Local_var_1_address:_%p\n", &
32     static_var_1);
33 printf("Static_Local_var_2_address:_%p\n", &
34     static_var_2);
35
36     printf("Global_var_1_address:_%p\n", &global_var_1
37         );
38     printf("Global_var_2_address:_%p\n", &global_var_2
39         );
40
41     return 0;
42 }
```

REFERENCES

- [1] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation, 2002.
- [2] A. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2015.