Item 1: A 'Global' Constant (section)

-- code snippet used: hello_world.cc

-- In this part of my report I use global liberally to mean that the scope of the variable is "global" in scope to this piece of code. This is obviously different than declaring a variable with the 'extern' keyword.

```
(gdb) b main
Breakpoint 1 at 0x400ba5: file hello_world.cc, line 7.
(gdb) r
Starting program: /acct/jha2/Fall2018/CSCE311/CSCE311-Pr2/HelloOb
warning: the debug information found in "/lib64/ld-2.23.so" does not m
RC mismatch).


Breakpoint 1, main (argc=1, argv=0x7fffffffde68) at hello_world.cc:7
7               std::cout << HELLO_WORLD << std::endl;
(gdb) info locals
No locals.
(gdb) p HELLO_WORLD
$1 = "Hello World!"
(gdb) p &HELLO_WORLD
$2 = (const std::__cxx11::string *) 0x6021e0 <HELLO_WORLD>
(gdb)
```

-- This shows that during execution, the identifier of the const global 'HELLO_WORLD' is registered at memory address 0x6021e0

```
 44: 00000000006021c0     1 OBJECT  LOCAL  DEFAULT   27 _ZStL8__ioinit
 45: 00000000006021e0    32 OBJECT  LOCAL  DEFAULT   27 _ZL11HELLO_WORLD
 46: 0000000000400bc8   200 FUNC    LOCAL  DEFAULT   14 _Z41__static_initializ
```

-- Here in the .symtab (the symbols table), we see this is where the variable name is stored, however, we that that its content is stored in the .rodata section of the elf file as seen below:

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2$ readelf -x .rodata HelloOb

Hex dump of section '.rodata':
  0x00400d30 01000200 48656c6c 6f20576f 726c6421 ....Hello World!
  0x00400d40 00                                  .
```

-

Item 2: Local Variables (segment)

-- code snippet used: input.cc

```
(gdb) info locals
input = ""
accepting = false
(gdb) p &input
$1 = (std::__cxx11::string *) 0x7fffffffdd40
(gdb) p &accepting
$2 = (bool *) 0x7fffffffdd3f
(gdb) info registers
rax            0x6021e0 6300128
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x7ffff783b780   140737345992576
rdi            0x7ffff783a620   140737345988128
rbp            0x7fffffffdd80   0x7fffffffdd80
rsp            0x7fffffffdd20   0x7fffffffdd20
r8             0x7ffff783b780   140737345992576
r9             0x7ffff7fa7740   140737353774912
r10            0x58     88
r11            0x7ffff74e27a0   140737342482336
r12            0x400c80 4197504
r13            0x7fffffffde60   140737488346720
r14            0x0      0
r15            0x0      0
rip            0x400dde 0x400dde <main(int, char**)+104>
eflags         0x213    [ CF AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

-- Here we see that the program has two local variable, a string ‘input’ and a boolean ‘accepting’ Their addresses are 0x7fffffffdd40 and 0x7fffffffdd3f respectively and both of these address exists between the top and bottom of the stack represented by the rsp and the rbp registers which point to 0x7fffffffdd20 and 0x7fffffffdd80 respectively. Thus, the local variable are held on the stack. Dynamic allocation is occurring on the call stack.

-

Item 3: Functions (segment)

-- code snippet used: square.cc

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2$ readelf -a FunctionOb
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400e10
  Start of program headers:          64 (bytes into file)
```

-- Here in the header of the ELF file, we see that the entry point for the execution of this program is 0x400e10

```
(gdb) x /200 0x400e10
0x400e10 <_start>:      0x8949ed31      0x89485ed1      0xe48348e2      0x495450f0
0x400e20 <_start+16>:   0x12a0c0c7      0xc7480040      0x401230c1      0xc7c74800
0x400e30 <_start+32>:   0x00400f2f      0xfffe87e8      0x0f66f4ff      0x0000441f
0x400e40 <deregister_tm_clones>:        0x6020dfb8      0x2d485500      0x006020d8      0x0ef88348
0x400e50 <deregister_tm_clones+16>:     0x76e58948      0x0000b81b      0x85480000      0x5d1174c0
0x400e60 <deregister_tm_clones+32>:     0x6020d8bf      0x66e0ff00      0x00841f0f      0x00000000
0x400e70 <deregister_tm_clones+48>:     0x1f0fc35d      0x2e660040      0x00841f0f      0x00000000
0x400e80 <register_tm_clones>:  0x6020d8be      0x81485500      0x6020d8ee      0xfec14800
0x400e90 <register_tm_clones+16>:       0xe5894803      0x48f08948      0x483fe8c1      0xd148c601
0x400ea0 <register_tm_clones+32>:       0xb81574fe      0x00000000      0x74c08548      0xd8bf5d0b
0x400eb0 <register_tm_clones+48>:       0xff006020      0x001f0fe0      0x0f66c35d      0x0000441f
0x400ec0 <__do_global_dtors_aux>:       0x14493d80      0x75000020      0x89485511      0xff6ee8e5
0x400ed0 <__do_global_dtors_aux+16>:    0xc65dffff      0x20143605      0xc3f30100      0x00401f0f
0x400ee0 <frame_dummy>: 0x601e00bf      0x3f834800      0xeb057500      0x001f0f93
0x400ef0 <frame_dummy+16>:      0x000000b8      0xc0854800      0x4855f174      0xd0ffe589
0x400f00 <frame_dummy+32>:      0xff7ae95d      0x4855ffff      0x0ff2e589      0xf2e84511
0x400f10 <GetSquare(double)+10>:        0xe845100f      0x45110ff2      0x100ff2f8      0x0ff2f845
0x400f20 <GetSquare(double)+26>:        0xf2f84559      0xf845110f      0x45100ff2      0x55c35df8
0x400f30 <main(int, char**)+1>: 0x53e58948      0x68ec8348      0x48ac7d89      0x64a07589
0x400f40 <main(int, char**)+17>:        0x25048b48      0x00000028      0xe8458948      0x8d48c031
0x400f50 <main(int, char**)+33>:        0x8948b645      0xfe66e8c7      0x8d48ffff      0x8d48b655
0x400f60 <main(int, char**)+49>:        0xbebec045      0x48004012      0x41e8c789      0x48ffffe
0x400f70 <main(int, char**)+65>:        0x48b6458d      0xd5e8c789      0x66fffffd      0xf2c0ef0f
0x400f80 <main(int, char**)+81>:        0xb845110f      0x4012c0be      0x2200bf00      0x7de80060
0x400f90 <main(int, char**)+97>:        0xbeffffd       0x00400da0      0xe8c78948      0xfffffdd0
0x400fa0 <main(int, char**)+113>:       0x01b745c6      0x00b77d80      0x008b840f      0x8d480000
0x400fb0 <main(int, char**)+129>:       0x8948c045      0x20e0bfc6      0xc1e80060      0x48ffffd
```

-- Examining memory from this position, we see that the definition for this function is stored in memory alongside the rest of the code required for the execution of this program.

This is further enforced in two ways:

```
GetSquare (input=3) at square.cc:5
5               double ret = input;
(gdb) s
6               ret *= ret;
(gdb) p r
No symbol "r" in current context.
(gdb) info r
rax            0x4008000000000000       4613937818241073152
rbx            0x0      0
rcx            0x0      0
rdx            0x4008000000000000       4613937818241073152
rsi            0x4000000000000000       4611686018427387904
rdi            0x18000000000000 6755399441055744
rbp            0x7fffffffdcf0   0x7fffffffdcf0
rsp            0x7fffffffdcf0   0x7fffffffdcf0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x7ffff74b0d80   140737342279040
r12            0x400e10 4197904
r13            0x7fffffffde50   140737488346704
r14            0x0      0
r15            0x0      0
rip            0x400f19 0x400f19 <GetSquare(double)+19>
```

-- When the execution is in the function GetSquare(), we can see that the memory address of the next instruction, held in the rip register is in the proper range for the rest of the code.

```
   95: 0000000000400da0     0 FUNC    GLOBAL DEFAULT  UND _ZSt4endlIcSt11char_trait
   96: 0000000000400f06    41 FUNC    GLOBAL DEFAULT   14 _Z9GetSquared
   97: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri
```

-- The entry for the function in the symbols table has an address that matches what we expected to see from the examination of the memory in the gdb stage.

-- This segment or program header is the first LOAD is the following picture (deduced from address comparison of these segments and the addresses of the lines of code see in the earlier pictures):

```
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001578 0x0000000000001578  R E    200000
  LOAD           0x0000000000001de8 0x0000000000601de8 0x0000000000601de8
                 0x00000000000002f0 0x0000000000000530  RW     200000
  DYNAMIC        0x0000000000001e08 0x0000000000601e08 0x0000000000601e08
                 0x00000000000001f0 0x00000000000001f0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000001320 0x0000000000401320 0x0000000000401320
                 0x0000000000000064 0x0000000000000064  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001de8 0x0000000000601de8 0x0000000000601de8
                 0x0000000000000218 0x0000000000000218  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame .gcc_except
_table
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

-- Lastly, it is seen that the first LOAD segment that has the .text section (or the actual code) spans memory from 0x400000 to 0x401578 which covers the address for the GetSquare() function. Also, this LOAD segment has Flag: R E for read and execute which make sense for executable lines of code.

-

Item 4: A static array - initialized (section)

-- code snippet: array_sum.cc

```
(gdb) info locals
inputs = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
acc = 0
(gdb) p &inputs
$1 = (int (*)[10]) 0x601080 <main::inputs>
(gdb)
```

-- Here we see that the array inputs (which was declared statically and initialized) has address 0x601080.

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2$ readelf -x .data ArrayOb

Hex dump of section '.data':
  0x00601060 00000000 00000000 00000000 00000000 ................
  0x00601070 00000000 00000000 00000000 00000000 ................
  0x00601080 01000000 01000000 01000000 01000000 ................
  0x00601090 01000000 01000000 01000000 01000000 ................
  0x006010a0 01000000 01000000                   ........
```

-- Here we see in a hex dump of the '.data' section the memory address that the 'inputs' array is located at. In addition, we see 10 little endian 1 values which correlates to the initialized 10 1s in the array starting from the address priorly mentioned for this variable.

-- Thus, we see that statically allocated arrays (initialized) are stored in the '.data' section of the ELF file.

Item 5: An int declared as extern and initialized

-- code snippet: extern_int.cc

```
(gdb) p MY_INT
$1 = 1024
(gdb) p &MY_INT
$2 = (int *) 0x601068 <MY_INT>
(gdb)
```

-- Here in the gdb debugger, we see that the address of our extern int is 0x601068

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2/johns_stuff$ readelf -x .data ExternIntOb

Hex dump of section '.data':
  0x00601058 00000000 00000000 00000000 00000000 ................
  0x00601068 00040000                            ....
```

-- Here in a hex dump of the .data section of the ELF file we see both the address 0x601068 and the value 1024 in hexadecimal little endian (that is the 00040000 located beside the corresponding address). So, that is where this programming element gets stored in the ELF file.

Item 6: Functions (sections)

-- code snippet: square.cc

```
GetSquare (input=3) at square.cc:5
5               double ret = input;
(gdb) s
6               ret *= ret;
(gdb) p r
No symbol "r" in current context.
(gdb) info r
rax            0x4008000000000000       4613937818241073152
rbx            0x0      0
rcx            0x0      0
rdx            0x4008000000000000       4613937818241073152
rsi            0x4000000000000000       4611686018427387904
rdi            0x18000000000000 6755399441055744
rbp            0x7fffffffdcf0   0x7fffffffdcf0
rsp            0x7fffffffdcf0   0x7fffffffdcf0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x7ffff74b0d80   140737342279040
r12            0x400e10 4197904
r13            0x7fffffffde50   140737488346704
r14            0x0      0
r15            0x0      0
rip            0x400f19 0x400f19 <GetSquare(double)+19>
```

-- When the execution is in the function GetSquare(), we can see that the memory address of the next instruction, held in the rip register is 0x400f19. So we know, address that contain the lines of code for the GetSquare() function are around this area.

```
[jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2$ readelf -x .text Fur

 Hex dump of section '.text':
   0x00400e10 31ed4989 d15e4889 e24883e4 f0505449 1.I..^H..H...PTI
   0x00400e20 c7c0a012 400048c7 c1301240 0048c7c7 ....@.H..0.@.H..
   0x00400e30 2f0f4000 e887feff fff4660f 1f440000 /.@.......f..D..
   0x00400e40 b8df2060 0055482d d8206000 4883f80e .. `.UH-. `.H...
   0x00400e50 4889e576 1bb80000 00004885 c074115d H..v......H..t.]
   0x00400e60 bfd82060 00ffe066 0f1f8400 00000000 .. `...f........
   0x00400e70 5dc30f1f 4000662e 0f1f8400 00000000 ]...@.f.........
   0x00400e80 bed82060 00554881 eed82060 0048c1fe .. `.UH... `.H..
   0x00400e90 034889e5 4889f048 c1e83f48 01c648d1 .H..H..H..?H..H.
   0x00400ea0 fe7415b8 00000000 4885c074 0b5dbfd8 .t......H..t.]..
   0x00400eb0 206000ff e00f1f00 5dc3660f 1f440000  `......].f..D..
   0x00400ec0 803d4914 20000075 11554889 e5e86eff .=I. ..u.UH...n.
   0x00400ed0 ffff5dc6 05361420 0001f3c3 0f1f4000 ..]..6. ......@.
   0x00400ee0 bf001e60 0048833f 007505eb 930f1f00 ...`.H.?.u......
   0x00400ef0 b8000000 004885c0 74f15548 89e5ffd0 .....H..t.UH....
   0x00400f00 5de97aff ffff5548 89e5f20f 1145e8f2 ].z...UH.....E..
   0x00400f10 0f1045e8 f20f1145 f8f20f10 45f8f20f ..E....E....E...
   0x00400f20 5945f8f2 0f1145f8 f20f1045 f85dc355 YE....E....E.].U
   0x00400f30 4889e553 4883ec68 897dac48 8975a064 H..SH..h.}.H.u.d
   0x00400f40 488b0425 28000000 488945e8 31c0488d H..%(...H.E.1.H.
   0x00400f50 45b64889 c7e866fe ffff488d 55b6488d E.H...f...H.U.H.
   0x00400f60 45c0bebe 12400048 89c7e841 feffff48 E....@.H...A...H
   0x00400f70 8d45b648 89c7e8d5 fdffff66 0fefc0f2 .E.H.......f....
   0x00400f80 0f1145b8 bec01240 00bf0022 6000e87d ..E....@..."`..}
   0x00400f90 fdffffbe a00d4000 4889c7e8 d0fdffff ......@.H.......
   0x00400fa0 c645b701 807db700 0f848b00 0000488d .E...}........H.
   0x00400fb0 45c04889 c6bfe020 6000e8c1 fdffff48 E.H.... `......H
   0x00400fc0 8d45c0be 08134000 4889c7e8 29020000 .E....@.H...)...
   0x00400fd0 84c07406 c645b700 ebca488d 45c0be00 ..t..E....H.E...
   0x00400fe0 00000048 89c7e805 01000066 480f7ec0 ...H.......fH.~.
```

-- Doing a hex dump of the data in the '.text' section of the ELF file shows that the same memory addresses that contain the executable instructions of the code are the the ones held in the '.text' section of the ELF file. Thus this is the section that holds the actual code (including function like the GetDouble() function in the code snippet).

-

Item 7: A static array - uninitialized (section)

-- code snippet: array_sum.cc

```
(gdb) info locals
uninitarr = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
acc = 0
(gdb) p &uninitarr
$1 = (int (*)[10]) 0x6012e0 <main::uninitarr>
(gdb)
```

-- Here we see that the array ‘uninitarr’ (which was declared statically) has address 0x6012e0.

```
     0000000000000010  0000000000000000  WA       0     0     8
[26] .bss              NOBITS           0000000000601080  00001070
     0000000000000288  0000000000000000  WA       0     0     32
[27] .comment          PROGBITS         0000000000000000  00001070
```

-- Here in the Section Headers part of the ELF file, we see that the ‘.bss’ section has index 26 and we see that is starts at address 0x601080 and has size of 288 which means that this section spans to address 0x601308. Thus, it can be see that the address for the ‘uninitarr’ array is within the .bss section. (This explains why the values in the array are automatically filled to zero.

```
43: 00000000006012c0     1 OBJECT  LOCAL  DEFAULT   26 _ZStL8__ioinit
44: 00000000006012e0    40 OBJECT  LOCAL  DEFAULT   26 _ZZ4mainE9uninitarr
45: 00000000004009dc    62 FUNC    LOCAL  DEFAULT   14 _Z41__static_initializati
```

-- Lastly, from the symbols table, we see on line 44, the entry for the ‘uninitarr’ has the same address as gdb reported, but we also see the number 26 there in the last column which represents the .bss index that was commented on earlier.

-- Thus, we see that statically allocated arrays (uninitialized) are stored (are initialized) in the ‘.bss’ section of the ELF file.

Item 8: A static array - initialized (segment)

-- code snippet used: array_sum.cc

```
(gdb) info locals
inputs = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
acc = 0
(gdb) p &inputs
$1 = (int (*)[10]) 0x601080 <main::inputs>
(gdb)
```

-- Here we see that the array ‘inputs’ (which was declared statically and initialized) has address 0x601080.

```
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000c6c 0x0000000000000c6c  R E    200000
  LOAD           0x0000000000000df8 0x0000000000600df8 0x0000000000600df8
                 0x00000000000002b0 0x0000000000000500  RW     200000
  DYNAMIC        0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x00000000000001e0 0x00000000000001e0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000af4 0x0000000000400af4 0x0000000000400af4
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000df8 0x0000000000600df8 0x0000000000600df8
                 0x0000000000000208 0x0000000000000208  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.versi
on_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

-- Looking at the Program Headers or the Segment Headers, we see that the .data section gets loaded during the second LOAD. This segment has Flags for reading and writing which makes

sense for content that is part of the data (since one should be able to read from and write to data). Analyzing the address and size of this segment, we see that this segments begins at 0x600df8 and spans to 0x6012f8 since it has size of 0x500. Note that this ‘inputs’ array which as stated earlier is in the .data section is located within the segment as well. Thus, it can be see that this segment, encapsulates the fixed data for the program.

-- Analyzing the symbol table, we see that the data section starts at address 0x602088 (line 70) and ends at address 0x602098 (line 83), and thus, the content of the ‘HELLO_WORLD’ constant is stored in the data segment of the ELF file.

Item 9: String Literals Not Associated With a Variable (Section)

-- using code snippet: square.cc

```
14
15        std::cout << "ENTER A NUMBER AND ILL GIVE YOU THE SQUARE OF IT (enter 'stop' to exit)" << std::endl;
16
```

-- Here on line 15, we see that there is a string that is being passed to standard output that is not associated with a variable. It is a string literal that will not and can not change.

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2/johns_stuff$ readelf -x .rodata FunctionOb

Hex dump of section '.rodata':
  0x004012b0 01000200 00000000 0073746f 64000000 .........stod...
  0x004012c0 454e5445 52204120 4e554d42 45522041 ENTER A NUMBER A
  0x004012d0 4e442049 4c4c2047 49564520 594f5520 ND ILL GIVE YOU
  0x004012e0 54484520 53515541 5245204f 46204954 THE SQUARE OF IT
  0x004012f0 2028656e 74657220 2773746f 70272074  (enter 'stop' t
  0x00401300 6f206578 69742900 73746f70 00546865 o exit).stop.The
  0x00401310 20737175 61726520 69733a20 00        square is: .
```

-- Here in a hex dump of the .rodata section of the ELF file, we see the entirety of this string literal.

-- Thus, string literals used in print statements and similar things (not associated to a variable) are stored in the .rodata section of the ELF file.

Item 10 and 11: A function in a class (Segment and Segment)

-- using code snippet: process_simulator.cc (using process_heap.cc and process.cc and the corresponding header files)

```
A syntax error in expression, near `%rip'.
(gdb) p $rip
$1 = (void (*)(void)) 0x401c3f <main(int, char**)+25>
(gdb) s
15              ProcessHeap heap = ProcessHeap(HEAP_SIZE);
(gdb) p $rip
$2 = (void (*)(void)) 0x401c4e <main(int, char**)+40>
(gdb) s
ProcessHeap::ProcessHeap (this=0x7fffffffd7f0, size=100) at process_heap.cc:8
8       ProcessHeap::ProcessHeap(int size){
(gdb) p $rip
$3 = (void (*)(void)) 0x4032fe <ProcessHeap::ProcessHeap(int)+24>
(gdb) where
#0  ProcessHeap::ProcessHeap (this=0x7fffffffd7f0, size=100) at process_heap.cc:8
#1  0x0000000000401c62 in main (argc=1, argv=0x7fffffffde38) at process_simulator.cc:15
```

-- There are a lot of things to discuss in this picture so bear with it for one second. First, from main, the constructor for a ProcessHeap is called. This is the ‘ProcessHeap heap = ProcessHeap(HEAP_SIZE) line. So from there, the execution moves to the ProcessHeap class code. This can be seen since the rip (instruction pointer register) is now holding an address that correlates to ProcessHeap::ProcessHeap(int)+24. In addition, when looking at ‘where’ we are, we see that we are within the ProcessHeap::ProcessHeap function called from main. And the address of the instructions being read is in the range of 0x4032fe as seen by the content of the rip register.

```
jha2@cocsce-l1d39-12:~/Fall2018/CSCE311/CSCE311-Pr2/johns_stuff$ readelf -l ProcSimOb

Elf file type is EXEC (Executable file)
Entry point 0x401b30
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000568e 0x000000000000568e  R E    200000
  LOAD           0x0000000000005dd8 0x0000000000605dd8 0x0000000000605dd8
                 0x00000000000003c4 0x0000000000000648  RW     200000
  DYNAMIC        0x0000000000005e08 0x0000000000605e08 0x0000000000605e08
                 0x00000000000001f0 0x00000000000001f0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x000000000000498c 0x000000000040498c 0x000000000040498c
                 0x00000000000001c4 0x00000000000001c4  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000005dd8 0x0000000000605dd8 0x0000000000605dd8
                 0x0000000000000228 0x0000000000000228  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rel
a.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame .gcc_except_table
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

-- Next, we look at the range of the code Segment or Program header again. We see the .text section, or the section containing the executable code, is mapped to the first LOAD segment. This segment has address starting at 0x400000 and spanning to 0x40568e which contains our address of execution within the constructor.

-- Thus, we see that included class code gets stored in the code segment of the ELF file.

```
(gdb) where
 #0  ProcessHeap::ProcessHeap (this=0x7fffffffd7f0, size=100) at process_heap.cc:8
 #1  0x0000000000401c62 in main (argc=1, argv=0x7fffffffde38) at process_simulator.cc:15
(gdb) p size
 $5 = 100
(gdb) p &size
 $6 = (int *) 0x7fffffffd774
```

-- Again inside of the constructor for the ProcessHeap, we are low looking at the parameter for the function, ‘size’.

-- We see that ‘size’ has address 0x7fffffffd774

```
(gdb) info registers
rax            0x7fffffffd7f0   140737488345072
rbx            0x0      0
rcx            0x1a0    416
rdx            0x7fffffffde48   140737488346696
rsi            0x64     100
rdi            0x7fffffffd7f0   140737488345072
rbp            0x7fffffffd7b0   0x7fffffffd7b0
rsp            0x7fffffffd770   0x7fffffffd770
r8             0x7ffff7dd4ac0   140737351862976
r9             0x7ffff7dc9780   140737351817088
r10            0x15b    347
r11            0x7ffff7b057e0   140737348917216
r12            0x401b30 4201264
r13            0x7fffffffde30   140737488346672
r14            0x0      0
r15            0x0      0
rip            0x4032fe 0x4032fe <ProcessHeap::ProcessHeap(int)+24>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

-- Looking at the registers for the stack pointer and the base (of the stack) pointer, we see that these registers have addresses 0x7fffffffd770 and 0x7fffffffd7b0 respectively. And note that our address for ‘size’ falls within this range.

-- Thus memory for parameters in functions are dynamically allocated on the Stack.

Report Questions:

-- *What are the APIs used to allocate memory in Linux user space, and when to use which.*
Typical user space memory allocation methods include:

Stack: alloca()
Heap: malloc()/calloc()/realloc()
Memory mapping: mmap(): Anonymous memory or with file descriptor

Generally alloca() is best used to allocate a small amount of memory (so that a stack overflow can be avoided). For larger memory sizes, malloc()/calloc()/realloc() should be used. Both malloc() and calloc() are used for dynamic memory allocation, but malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to zero. realloc() is used to resize the block of memory previously allocated by malloc()/calloc(). If you need to perform operations on a file that is larger than the available memory, mmap() should be used.

-- *When an address p is dereferenced, you may encounter a SIGSEGV. Describe how the system recognizes p is an invalid address and triggers a SIGSEGV.*

In this case, the virtual address p needs to be translated into a physical address, thus the TLB is searched. If a match is found, the physical address is returned and memory access can continue. However, if there is no match, the Memory Management Unit will typically look up the address mapping in the page table to see whether a mapping exists. If one exists, it is written back to the TLB, and the faulting instruction is restarted. This subsequent translation will find a TLB hit, and the memory access will continue.
However, the lookup may fail if there is no translation available for the virtual address, meaning that virtual address p is invalid. This will result in SIGSEGV being triggered.

-- *What information is saved at /proc/$pid/maps?*

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process or thread.
Each row has the following fields:
address - This is the starting and ending address of the region.
permissions - This describes how pages in the region can be accessed.
offset - If the memory region was mapped from a file, this is the offset in the file where the mapping begins. If the memory was not mapped from a file, the offset will be 0.
device - If the region was mapped from a file, this is the major and minor device number (in hex) where the file lives.
inode - If the region was mapped from a file, this is the file number.
pathname - If the region was mapped from a file, this is the name of the file. This field is blank for anonymous mapped regions. There are also special regions with names like [heap], [stack], or [vdso]. [vdso] stands for virtual dynamic shared object.