2.3 First Code

Question 2.1

```
Lab02 — vagrant@ubuntu1804: ~/lab — ssh ◂ vagrant ssh — 89×29
    Lab02: Setting up gcc-7-base:amd64 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up binutils-common:amd64 (2.30-21ubuntu1~18.04.9) ...
    Lab02: Setting up libmpx2:amd64 (8.4.0-1ubuntu1~18.04) ...
    Lab02: Setting up gdbserver (8.1.1-0ubuntu1) ...
    Lab02: Setting up libmpc3:amd64 (1.1.0-1) ...
    Lab02: Setting up libc-dev-bin (2.27-3ubuntu1.6) ...
    Lab02: Setting up manpages-dev (4.15-1) ...
    Lab02: Setting up libc6-dev:amd64 (2.27-3ubuntu1.6) ...
    Lab02: Setting up libitm1:amd64 (8.4.0-1ubuntu1~18.04) ...
    Lab02: Setting up libbabeltrace1:amd64 (1.5.5-1) ...
    Lab02: Setting up libisl19:amd64 (0.19-1) ...
    Lab02: Setting up libasan4:amd64 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up libbinutils:amd64 (2.30-21ubuntu1~18.04.9) ...
    Lab02: Setting up libcilkrts5:amd64 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up libubsan0:amd64 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up libgcc-7-dev:amd64 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up cpp-7 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up gdb (8.1.1-0ubuntu1) ...
    Lab02: Setting up binutils-x86-64-linux-gnu (2.30-21ubuntu1~18.04.9) ...
    Lab02: Setting up cpp (4:7.4.0-1ubuntu2.3) ...
    Lab02: Setting up binutils (2.30-21ubuntu1~18.04.9) ...
    Lab02: Setting up gcc-7 (7.5.0-3ubuntu1~18.04) ...
    Lab02: Setting up gcc (4:7.4.0-1ubuntu2.3) ...
    Lab02: Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
    Lab02: Processing triggers for libc-bin (2.27-3ubuntu1.6) ...
Alisas-MacBook-Pro:Lab02 alisatodorova$ vagrant ssh
---------------------- VIRTUALBOX ----------------------
vagrant@ubuntu1804:~$ cd lab
vagrant@ubuntu1804:~/lab$
```

Question 2.2

1. There's a missing semicolon (;) after *printf("Helloworld!\n")*.
2. The printf statement is usually written "Hello, world!".
3. Since the main function has return type int, we need to make sure it returns an integer.

The corrected code:

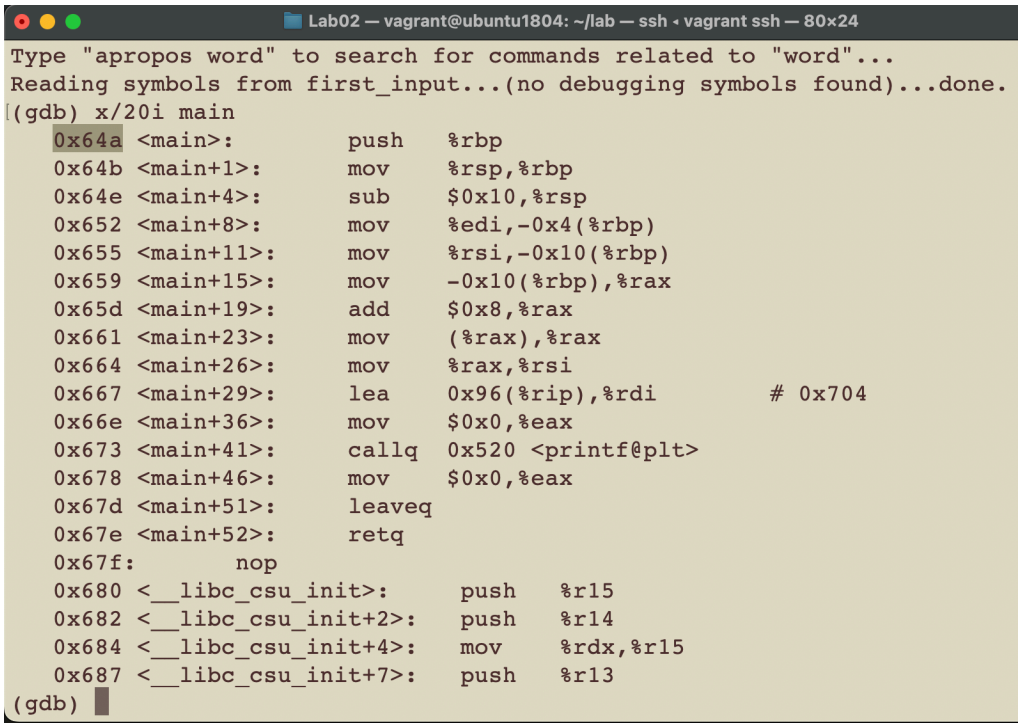
```
#include <stdio.h>

 int main(int argc, char** argv){
    printf("Hello, world!\n");
    return 0;
}
```

Question 2.3

```
vagrant@ubuntu1804:~/lab$ gcc -Wall -o hello hello.c
vagrant@ubuntu1804:~/lab$ ./hello
Hello, world!
vagrant@ubuntu1804:~/lab$
```

Question 2.4

The address at which the code of the main function is written is 0x64a.



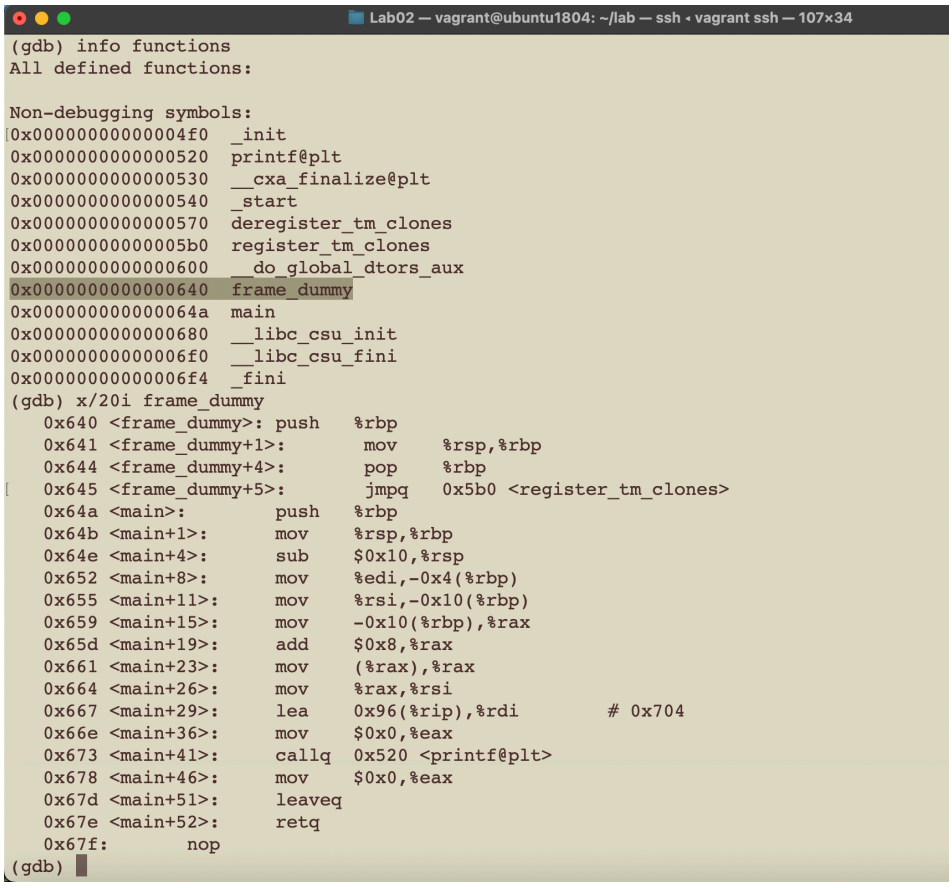
Question 2.5

The code in gdb is written in assembly language.

Question 2.6

The instruction in the main function that is related to printing the inputs is **callq printf**:
0x673 <main+41>: callq 0x520 <printf@plt>

Question 2.7



Question 2.8

We use commands **x/20i main** and **x/100b main** to make the following correspondence table:

x/100b main	x/20i main
0x55	push %rbp
0x48 89 e5	mov %rsp, %rbp
0x48 83 ec10	sub $0x10, %rsp
0x89 7d fc	mov %edi,-0x4(%rbp)
0x48 89 75 f0	mov %rsi,-0x10(%rbp)
0x48 8b 45 f0	mov -0x10(%rbp),%rax
0x48 83 c0 08	add $0x8,%rax
0x48 8b 00	mov (%rax),%rax
0x48 89 c6	mov %rax,%rsi
0x48 8d 3d 96 00 00 00	lea 0x96(%rip),%rdi # 0x704
0xb8 00 00 00 00	mov $0x0,%eax
0xe8 a8 fe ff ff	callq 0x520 <printf@plt>
0xb8 00 00 00 00	mov $0x0,%eax
0xc9	leaveq
0xc3	retq
0x90	nop

Question 2.9

The breakpoint is not set exactly on the first assembly instruction of the main function. The breakpoint is at address 0x64e, which is instruction **sub $0x10,%rsp**. This is the third instruction of the main function, and it indicates the beginning of the **main** function.

```
(gdb) b main
Breakpoint 1 at 0x64e
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000000064e <main+4>
```

The instructions executed nonetheless are:

```
0x64a <main>:  push   %rbp
0x64b <main+1>:   mov    %rsp,%rbp
0x64e <main+4>:   sub    $0x10,%rsp
```

The role of these instructions is to set up the stack frame. In assembly code, the first two instructions are always “push %rbp” (which saves the current base pointer onto the stack), followed by “mov %rsp, %rbp” (which updates the base pointer to the current stack pointer). Then, the instruction "sub $0x10, %rsp” allocates space for local variables.

The breakpoint is set after them so that we can inspect the initial state of the program (i.e., initial registers, memory, variables) before we proceed with its execution.

Question 2.10

We try to run the program **first_input** with the argument "hello", but we are stopped by the breakpoint set at address 0x000055555555464e (i.e., 0x64e).

```
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555464e in main ()
```

A breakpoint allows us to stop the execution of the program at a certain location (or point). Then, through GDB (which is our debugger for this lab) we can step through the code interactively in order to debug and inspect the current state of the program (i.e., see resisters, memory, variables).

Question 2.11

I am choosing the following instruction:

0x555555554659 <main+15>: mov -0x10(%rbp),%rax

```
(gdb) b *0x555555554659
Breakpoint 2 at 0x555555554659
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555464e in main ()
(gdb) continue
Continuing.

Breakpoint 2, 0x0000555555554659 in main ()
(gdb) continue
Continuing.
argv[1]: hello
[Inferior 1 (process 2007) exited normally]
(gdb)
```

```
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555464e in main ()
(gdb) ni
0x0000555555554652 in main ()
(gdb) ni
0x0000555555554655 in main ()
(gdb) ni

Breakpoint 2, 0x0000555555554659 in main ()
(gdb) ni
0x000055555555465d in main ()
(gdb) ni
0x0000555555554661 in main ()
(gdb) ni
0x0000555555554664 in main ()
(gdb) ni
0x0000555555554667 in main ()
(gdb) ni
0x000055555555466e in main ()
(gdb) ni
0x0000555555554673 in main ()
(gdb) ni
argv[1]: hello
0x0000555555554678 in main ()
(gdb) ni
0x000055555555467d in main ()
(gdb) ni
0x000055555555467e in main ()
(gdb) ni
__libc_start_main (main=0x55555555464a <main>, argc=2, argv=0x7fffffffe4c8, init=<optimized out>,
    fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7fffffffe4b8) at ../csu/libc-start.c:344
344     ../csu/libc-start.c: No such file or directory.
(gdb) ni
0x00007ffff7a03c89      344     in ../csu/libc-start.c
(gdb) ni
[Inferior 1 (process 2009) exited normally]
(gdb)
```

Question 2.12

With **info reg** we see the values of all the registers in hexadecimal (on the left) and decimal (on the right) formats at the current state of the program (i.e., at the breakpoint at address 0x000055555555464e).

```
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555464e in main ()
(gdb) info reg
rax            0x55555555464a   93824992233034
rbx            0x0      0
rcx            0x555555554680   93824992233088
rdx            0x7fffffffe4e0   140737488348384
rsi            0x7fffffffe4c8   140737488348360
rdi            0x2      2
rbp            0x7fffffffe3e0   0x7fffffffe3e0
rsp            0x7fffffffe3e0   0x7fffffffe3e0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554540   93824992232768
r13            0x7fffffffe4c0   140737488348352
r14            0x0      0
r15            0x0      0
rip            0x55555555464e   0x55555555464e <main+4>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

The “rip 0x55555555464e 0x55555555464e <main+4>” is the instruction pointer. Its value 0x55555555464e indicates the location of the next instruction to be executed. This value corresponds to the <main+4> location, which is the beginning of the **main** function (more specifically, it's 4 bytes into it). Therefore, this value is logical.

The “rsp 0x7fffffffe3e0 0x7fffffffe3e0” is the stack pointer. Its value 0x7fffffffe3e0 indicates the address of the current top of the stack.

Question 2.13

In order to generate string **hello** in ASCII, we need to write the following hexadecimal codes:"\x68\x65\x6C\x6C\x6F".

Question 2.14

```
(gdb) run `bash inputGenerator.sh`
Starting program: /home/vagrant/lab/first_input `bash inputGenerator.sh`

Breakpoint 1, 0x000055555555464e in main ()
(gdb) continue
Continuing.

Breakpoint 2, 0x0000555555554659 in main ()
(gdb) continue
Continuing.
argv[1]: hello
[Inferior 1 (process 2038) exited normally]
(gdb)
```

Question 2.15

A stack is a special region of the computer's memory that stores temporary variables created by each function (including the **main()** function). The stack grows and shrinks as functions push and pop local variables. According to the article, since the stack is "LIFO" (last in, first out) data structure, it grows downward in memory. Furthermore, if we look at **Question 2.16** (picture on the right):
We can see that the addresses are: rbp>rsp (i.e., address 0x7fffffffe3e0 is bigger than address 0x7fffffffe3d0), which shows that the stack grows downwards.

```
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555466e in main ()
(gdb) info registers
rax            0x7fffffffe73e   140737488348990
rbx            0x0      0
rcx            0x555555554680   93824992233088
rdx            0x7fffffffe4e0   140737488348384
rsi            0x7fffffffe73e   140737488348990
rdi            0x555555554704   93824992233220
rbp            0x7fffffffe3e0   0x7fffffffe3e0
rsp            0x7fffffffe3d0   0x7fffffffe3d0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554540   93824992232768
r13            0x7fffffffe4c0   140737488348352
r14            0x0      0
r15            0x0      0
rip            0x55555555466e   0x55555555466e <main+36>
eflags         0x212    [ AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

Reference:
Gribble, Paul. “7. Memory : Stack vs Heap”. (2012). Gribblelab. https://gribblelab.org/teaching/CBootCamp/7_Memory_Stack_vs_Heap.html

Question 2.16

PID: 2192
We have:
0x55555555466e <main+36>: mov $0x0,%eax
0x555555554673 <main+41>: callq 0x555555554520 <printf@plt>
Thus, we put a breakpoint: break *0x55555555466e

From /proc/2192/maps, we can find the location of the stack:
7fffce366000-7fffce387000 rw-p 00000000 00:00 0 [stack]
The stack is located in the address range “7fffce366000" to “7fffce387000”.

Lab02 — vagrant@ubuntu1804: ~/lab — ssh ‹ vagrant ssh — 118×40

```
7f2bbd158000-7f2bbd159000 r--p 0002d000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f2bbd159000-7f2bbd15a000 rw-p 0002e000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f2bbd15a000-7f2bbd15d000 r-xp 00000000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f2bbd15d000-7f2bbd35c000 ---p 00003000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f2bbd35c000-7f2bbd35d000 r--p 00002000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f2bbd35d000-7f2bbd35e000 rw-p 00003000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f2bbd35e000-7f2bbd37a000 r-xp 00000000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f2bbd37a000-7f2bbd579000 ---p 0001c000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f2bbd579000-7f2bbd57a000 r--p 0001b000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f2bbd57a000-7f2bbd57b000 rw-p 0001c000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f2bbd57b000-7f2bbd5bc000 r-xp 00000000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f2bbd5bc000-7f2bbd7bb000 ---p 00041000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f2bbd7bb000-7f2bbd7bd000 r--p 00040000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f2bbd7bd000-7f2bbd7c3000 rw-p 00042000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f2bbd7c3000-7f2bbd7c4000 rw-p 00000000 00:00 0
7f2bbd7c4000-7f2bbd7ed000 r-xp 00000000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f2bbd829000-7f2bbd82a000 r--p 00000000 08:03 5376272                    /usr/lib/locale/C.UTF-8/LC_NUMERIC
7f2bbd82a000-7f2bbd82b000 r--p 00000000 08:03 5376275                    /usr/lib/locale/C.UTF-8/LC_TIME
7f2bbd82b000-7f2bbd99e000 r--p 00000000 08:03 5376261                    /usr/lib/locale/C.UTF-8/LC_COLLATE
7f2bbd99e000-7f2bbd99f000 r--p 00000000 08:03 5376268                    /usr/lib/locale/C.UTF-8/LC_MONETARY
7f2bbd99f000-7f2bbd9d0000 r--p 00000000 08:03 5376263                    /usr/lib/locale/C.UTF-8/LC_CTYPE
7f2bbd9d0000-7f2bbd9d7000 r--s 00000000 08:03 5381397                    /usr/lib/x86_64-linux-gnu/gconv/gconv-modules
.cache
7f2bbd9d7000-7f2bbd9e6000 rw-p 00000000 00:00 0
7f2bbd9e6000-7f2bbd9e7000 r--p 00000000 08:03 5376273                    /usr/lib/locale/C.UTF-8/LC_PAPER
7f2bbd9e7000-7f2bbd9e8000 r--p 00000000 08:03 5376270                    /usr/lib/locale/C.UTF-8/LC_NAME
7f2bbd9e8000-7f2bbd9e9000 r--p 00000000 08:03 5376260                    /usr/lib/locale/C.UTF-8/LC_ADDRESS
7f2bbd9e9000-7f2bbd9ea000 r--p 00000000 08:03 5376274                    /usr/lib/locale/C.UTF-8/LC_TELEPHONE
7f2bbd9ea000-7f2bbd9eb000 r--p 00000000 08:03 5376265                    /usr/lib/locale/C.UTF-8/LC_MEASUREMENT
7f2bbd9eb000-7f2bbd9ec000 r--p 00000000 08:03 5376264                    /usr/lib/locale/C.UTF-8/LC_IDENTIFICATION
7f2bbd9ec000-7f2bbd9ed000 r--p 00000000 08:03 5376266                    /usr/lib/locale/C.UTF-8/LC_MESSAGES/SYS_LC_ME
SSAGES
7f2bbd9ed000-7f2bbd9ee000 r--p 00029000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f2bbd9ee000-7f2bbd9ef000 rw-p 0002a000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f2bbd9ef000-7f2bbd9f0000 rw-p 00000000 00:00 0
7fffce366000-7fffce387000 rw-p 00000000 00:00 0                          [stack]
7fffce3a6000-7fffce3a9000 r--p 00000000 00:00 0                          [vvar]
7fffce3a9000-7fffce3ab000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
vagrant@ubuntu1804:~/lab$
```

When we hit the breakpoint, the stack starts at 0x7ffffffde000, and ends at 0x7fffffff000.

```
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555466e in main ()
(gdb) info proc mappings
process 2217
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /home/vagrant/lab/first_input
      0x555555754000     0x555555755000     0x1000        0x0 /home/vagrant/lab/first_input
      0x555555755000     0x555555756000     0x1000     0x1000 /home/vagrant/lab/first_input
      0x7ffff79e2000     0x7ffff7bc9000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7bc9000     0x7ffff7dc9000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dc9000     0x7ffff7dcd000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcd000     0x7ffff7dcf000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000        0x0
      0x7ffff7dd3000     0x7ffff7dfc000    0x29000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7fee000     0x7ffff7ff0000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
---Type <return> to continue, or q <return> to quit---
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x29000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2a000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

We also look at rbp and rsp:
The rbp shows that the top of the stack is at 0x7fffffffe3e0, while the rsp shows that the address of the current top of the stack is at 0x7fffffffe3d0.

```
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555466e in main ()
(gdb) info registers
rax            0x7fffffffe73e   140737488348990
rbx            0x0      0
rcx            0x555555554680   93824992233088
rdx            0x7fffffffe4e0   140737488348384
rsi            0x7fffffffe73e   140737488348990
rdi            0x555555554704   93824992233220
rbp            0x7fffffffe3e0   0x7fffffffe3e0
rsp            0x7fffffffe3d0   0x7fffffffe3d0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554540   93824992232768
r13            0x7fffffffe4c0   140737488348352
r14            0x0      0
r15            0x0      0
rip            0x55555555466e   0x55555555466e <main+36>
eflags         0x212    [ AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

Question 2.17

Please see page 13 of this PDF. I had to redo it and couldn't add it here without messing up the rest.

Question 2.18

```
(gdb) info address new_function
Symbol "new_function" is at 0x68a in a file compiled without debugging.
```

The instruction **callq new_function** will make the code jump from **main** to **new_function**.

```
(gdb) x/20i main
   0x6a1 <main>:        push   %rbp
   0x6a2 <main+1>:      mov    %rsp,%rbp
   0x6a5 <main+4>:      sub    $0x10,%rsp
   0x6a9 <main+8>:      mov    %edi,-0x4(%rbp)
   0x6ac <main+11>:     mov    %rsi,-0x10(%rbp)
   0x6b0 <main+15>:     mov    -0x10(%rbp),%rax
   0x6b4 <main+19>:     add    $0x8,%rax
   0x6b8 <main+23>:     mov    (%rax),%rax
   0x6bb <main+26>:     mov    %rax,%rsi
   0x6be <main+29>:     lea    0xa3(%rip),%rdi        # 0x768
   0x6c5 <main+36>:     mov    $0x0,%eax
   0x6ca <main+41>:     callq  0x560 <printf@plt>
   0x6cf <main+46>:     mov    $0x0,%eax
   0x6d4 <main+51>:     callq  0x68a <new_function>
   0x6d9 <main+56>:     mov    $0x0,%eax
   0x6de <main+61>:     leaveq
   0x6df <main+62>:     retq
   0x6e0 <__libc_csu_init>:     push   %r15
   0x6e2 <__libc_csu_init+2>:   push   %r14
   0x6e4 <__libc_csu_init+4>:   mov    %rdx,%r15
```

The following two instructions make **new_function** jump back to **main**:

```
0x69f <new_function+21>:   pop  %rbp
0x6a0 <new_function+22>:   retq d
```

```
(gdb) x/20i new_function
   0x68a <new_function>:        push   %rbp
   0x68b <new_function+1>:      mov    %rsp,%rbp
   0x68e <new_function+4>:      lea    0xcf(%rip),%rdi        # 0x764
   0x695 <new_function+11>:     callq  0x550 <puts@plt>
   0x69a <new_function+16>:     mov    $0x0,%eax
   0x69f <new_function+21>:     pop    %rbp
   0x6a0 <new_function+22>:     retq
   0x6a1 <main>:        push   %rbp
   0x6a2 <main+1>:      mov    %rsp,%rbp
   0x6a5 <main+4>:      sub    $0x10,%rsp
   0x6a9 <main+8>:      mov    %edi,-0x4(%rbp)
   0x6ac <main+11>:     mov    %rsi,-0x10(%rbp)
   0x6b0 <main+15>:     mov    -0x10(%rbp),%rax
   0x6b4 <main+19>:     add    $0x8,%rax
   0x6b8 <main+23>:     mov    (%rax),%rax
   0x6bb <main+26>:     mov    %rax,%rsi
   0x6be <main+29>:     lea    0xa3(%rip),%rdi        # 0x768
   0x6c5 <main+36>:     mov    $0x0,%eax
   0x6ca <main+41>:     callq  0x560 <printf@plt>
   0x6cf <main+46>:     mov    $0x0,%eax
```

The program knows where (i.e. at which instruction) to go back in main thanks to **retq** instruction. When **callq new_function** is executed, the address of the next instruction is pushed onto the stack

as the return address. Now we jump to **new_function**. When **new_function** reaches the instruction **retq**, it pops the return address from the stack and jumps back to that address. Now we jump to **main**, and more specifically, to the next instruction to execute which is:
0x6d9 <main+56>: mov $0x0,%eax

Question 2.19
Please see page 15 of this PDF.

Question 2.20
Since we have that **int c = 0;** the if-condition **if (1 == c)** will never be satisfied. Thus, the output at every execution of the program will be the **else**-statement: **Troy shall never fall.**

Question 2.21
No optimisation:

```
Alisas-MacBook-Pro:lab alisatodorova$ gcc -Wall -o no_opt change_path.c
Alisas-MacBook-Pro:lab alisatodorova$ objdump -d no_opt

no_opt: file format Mach-O 64-bit x86-64


Disassembly of section __TEXT,__text:

0000000100000ef0 _main:
100000ef0: 55                          pushq   %rbp
100000ef1: 48 89 e5                    movq    %rsp, %rbp
100000ef4: 48 83 ec 20                 subq    $32, %rsp
100000ef8: c7 45 fc 00 00 00 00        movl    $0, -4(%rbp)
100000eff: 89 7d f8                    movl    %edi, -8(%rbp)
100000f02: 48 89 75 f0                 movq    %rsi, -16(%rbp)
100000f06: c7 45 ec 00 00 00 00        movl    $0, -20(%rbp)
100000f0d: b8 01 00 00 00              movl    $1, %eax
100000f12: 3b 45 ec                    cmpl    -20(%rbp), %eax
100000f15: 0f 85 13 00 00 00           jne     19 <_main+0x3e>
100000f1b: 48 8d 3d 44 00 00 00        leaq    68(%rip), %rdi
100000f22: b0 00                       movb    $0, %al
100000f24: e8 1b 00 00 00              callq   27 <dyld_stub_binder+0x100000f44>
100000f29: e9 0e 00 00 00              jmp     14 <_main+0x4c>
100000f2e: 48 8d 3d 5f 00 00 00        leaq    95(%rip), %rdi
100000f35: b0 00                       movb    $0, %al
100000f37: e8 08 00 00 00              callq   8 <dyld_stub_binder+0x100000f44>
100000f3c: 31 c0                       xorl    %eax, %eax
100000f3e: 48 83 c4 20                 addq    $32, %rsp
100000f42: 5d                          popq    %rbp
100000f43: c3                          retq

Disassembly of section __TEXT,__stubs:

0000000100000f44 __stubs:
100000f44: ff 25 b6 10 00 00           jmpq    *4278(%rip)

Disassembly of section __TEXT,__stub_helper:

0000000100000f4c __stub_helper:
100000f4c: 4c 8d 1d b5 10 00 00        leaq    4277(%rip), %r11
100000f53: 41 53                       pushq   %r11
100000f55: ff 25 a5 00 00 00           jmpq    *165(%rip)
100000f5b: 90                          nop
100000f5c: 68 00 00 00 00              pushq   $0
100000f61: e9 e6 ff ff ff              jmp     -26 <__stub_helper>
Alisas-MacBook-Pro:lab alisatodorova$
```

With optimization:

```
Alisas-MacBook-Pro:lab alisatodorova$ gcc -Wall -O2 -o with_opt change_path.c
Alisas-MacBook-Pro:lab alisatodorova$ objdump -d with_opt

with_opt:       file format Mach-O 64-bit x86-64


Disassembly of section __TEXT,__text:

0000000100000f60 _main:
100000f60: 55                           pushq   %rbp
100000f61: 48 89 e5                     movq    %rsp, %rbp
100000f64: 48 8d 3d 2b 00 00 00         leaq    43(%rip), %rdi
100000f6b: e8 04 00 00 00               callq   4 <dyld_stub_binder+0x100000f74>
100000f70: 31 c0                        xorl    %eax, %eax
100000f72: 5d                           popq    %rbp
100000f73: c3                           retq

Disassembly of section __TEXT,__stubs:

0000000100000f74 __stubs:
100000f74: ff 25 86 10 00 00            jmpq    *4230(%rip)

Disassembly of section __TEXT,__stub_helper:

0000000100000f7c __stub_helper:
100000f7c: 4c 8d 1d 85 10 00 00         leaq    4229(%rip), %r11
100000f83: 41 53                        pushq   %r11
100000f85: ff 25 75 00 00 00            jmpq    *117(%rip)
100000f8b: 90                           nop
100000f8c: 68 00 00 00 00               pushq   $0
100000f91: e9 e6 ff ff ff               jmp     -26 <__stub_helper>
Alisas-MacBook-Pro:lab alisatodorova$
```

We can see that with the optimisation (-O2), we have a concise compilation and reduced redundant instructions, which leads to a better and faster performance.

With the optimisation (-O2), the stack frame adjustment is implicit as we don't have the instructions **addq** and **subq** (like we have them in the non-optimisation).

With the optimisation (-O2), the **callq** instruction uses a relative address, whereas an absolute address is used with the non-optimisation.

With the optimisation (-O2), the instruction **xorl** is used to set **%eax** to zero, and then directly the **popq** instruction is executed. While, with the non-optimisation, the code checks the if-condition **if (1 == c)** and thus, we have instruction branching (i.e., the instructions jump to different executions).

Question 2.22

The two assembly instructions enabling to decide what branch to execute are two conditional branch instructions: **cmpl** and **jne**.

The instruction **cmpl** compares two operands, while the instruction **jne** is a conditional jump instruction.

In our case we have:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000055555555463a <+0>:     push   %rbp
   0x000055555555463b <+1>:     mov    %rsp,%rbp
=> 0x000055555555463e <+4>:     sub    $0x20,%rsp
   0x0000555555554642 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000555555554645 <+11>:    mov    %rsi,-0x20(%rbp)
   0x0000555555554649 <+15>:    movl   $0x0,-0x4(%rbp)
   0x0000555555554650 <+22>:    cmpl   $0x1,-0x4(%rbp)
   0x0000555555554654 <+26>:    jne    0x555555554664 <main+42>
   0x0000555555554656 <+28>:    lea    0xab(%rip),%rdi        # 0x555555554708
   0x000055555555465d <+35>:    callq  0x555555554510 <puts@plt>
   0x0000555555554662 <+40>:    jmp    0x555555554670 <main+54>
   0x0000555555554664 <+42>:    lea    0xca(%rip),%rdi        # 0x555555554735
   0x000055555555466b <+49>:    callq  0x555555554510 <puts@plt>
   0x0000555555554670 <+54>:    mov    $0x0,%eax
   0x0000555555554675 <+59>:    leaveq
   0x0000555555554676 <+60>:    retq
End of assembler dump.
(gdb)
```

The instruction **cmpl** (0x0000555555554650 <+22>: cmpl $0x1,-0x4(%rbp)) gives the result of the if-condition **if (1 == c)**. Then, based on this result, the instruction **jne** (0x0000555555554654 <+26>: jne 0x555555554664 <main+42>) jumps to codeblock **<main+42>**.

Question 2.23

```
(gdb) run change_path
Starting program: /home/vagrant/lab/change_path change_path

Breakpoint 1, 0x000055555555463e in main ()
(gdb) dissaseble main
Undefined command: "dissaseble".  Try "help".
(gdb) disassemble main
Dump of assembler code for function main:
   0x000055555555463a <+0>:     push   %rbp
   0x000055555555463b <+1>:     mov    %rsp,%rbp
=> 0x000055555555463e <+4>:     sub    $0x20,%rsp
   0x0000555555554642 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000555555554645 <+11>:    mov    %rsi,-0x20(%rbp)
   0x0000555555554649 <+15>:    movl   $0x0,-0x4(%rbp)
   0x0000555555554650 <+22>:    cmpl   $0x1,-0x4(%rbp)
   0x0000555555554654 <+26>:    jne    0x555555554664 <main+42>
   0x0000555555554656 <+28>:    lea    0xab(%rip),%rdi        # 0x555555554708
   0x000055555555465d <+35>:    callq  0x555555554510 <puts@plt>
   0x0000555555554662 <+40>:    jmp    0x555555554670 <main+54>
   0x0000555555554664 <+42>:    lea    0xca(%rip),%rdi        # 0x555555554735
   0x000055555555466b <+49>:    callq  0x555555554510 <puts@plt>
   0x0000555555554670 <+54>:    mov    $0x0,%eax
   0x0000555555554675 <+59>:    leaveq
   0x0000555555554676 <+60>:    retq
End of assembler dump.
(gdb) b *0x0000555555554650
Breakpoint 2 at 0x555555554650
(gdb) run change_path
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vagrant/lab/change_path change_path

Breakpoint 1, 0x000055555555463e in main ()
(gdb) continue
Continuing.

Breakpoint 2, 0x0000555555554650 in main ()
(gdb) info registers
rax            0x55555555463a   93824992233018
rbx            0x0      0
rcx            0x555555554680   93824992233088
rdx            0x7fffffffe4d0   140737488348368
rsi            0x7fffffffe4b8   140737488348344
rdi            0x2      2
rbp            0x7fffffffe3d0   0x7fffffffe3d0
rsp            0x7fffffffe3b0   0x7fffffffe3b0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554530   93824992232752
r13            0x7fffffffe4b0   140737488348336
r14            0x0      0
r15            0x0      0
rip            0x555555554650   0x555555554650 <main+22>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/4xg $rbp
0x7fffffffe3d0: 0x0000555555554680      0x00007ffff7a03c87
0x7fffffffe3e0: 0x0000000000000002      0x00007fffffffe4b8
(gdb) x/4xg $rbp
0x7fffffffe3d0: 0x0000555555554680      0x00007ffff7a03c87
0x7fffffffe3e0: 0x0000000000000002      0x00007fffffffe4b8
(gdb) x/g $rbp-0x4
0x7fffffffe3cc: 0x5555468000000000
(gdb) set *0x7fffffffe3cc=0x00000001
(gdb) x/g 0x7fffffffe3cc
0x7fffffffe3cc: 0x5555468000000001
(gdb) set *0x7fffffffe3cc = 0x00000001
(gdb) x/g 0x7fffffffe3cc
0x7fffffffe3cc: 0x5555468000000001
(gdb) continue
Continuing.
Wait, that was not an offering for the gods?
[Inferior 1 (process 2517) exited normally]
(gdb) run change_path
```

Question 2.17

I had to redo this question as originally my input was too short to be found in the stack. However, when I redid this question, my PID changed to 1892. (All the other questions remain with the old PID: 2192).

Therefore, with this change now we have:
From /proc/1892/maps, we can find the location of the stack:
7ffc65081000-7ffc650a2000 rw-p 00000000 00:00 0 [stack]
The stack is located in the address range “7ffc65081000" to “7ffc650a2000”.

```
Lab02 — vagrant@ubuntu1804: ~ — ssh ‹ vagrant ssh — 158×42
7f79fa1f0000-7f79fa21d000 r-xp 00000000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f79fa21d000-7f79fa41d000 ---p 0002d000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f79fa41d000-7f79fa41e000 r--p 0002d000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f79fa41e000-7f79fa41f000 rw-p 0002e000 08:03 1310859                    /lib/x86_64-linux-gnu/libncursesw.so.5.9
7f79fa41f000-7f79fa422000 r-xp 00000000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f79fa422000-7f79fa621000 ---p 00003000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f79fa621000-7f79fa622000 r--p 00002000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f79fa622000-7f79fa623000 rw-p 00003000 08:03 1316020                    /lib/x86_64-linux-gnu/libdl-2.27.so
7f79fa623000-7f79fa63f000 r-xp 00000000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f79fa63f000-7f79fa83e000 ---p 0001c000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f79fa83e000-7f79fa83f000 r--p 0001b000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f79fa83f000-7f79fa840000 rw-p 0001c000 08:03 1311170                    /lib/x86_64-linux-gnu/libz.so.1.2.11
7f79fa840000-7f79fa881000 r-xp 00000000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f79fa881000-7f79faa80000 ---p 00041000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f79faa80000-7f79faa82000 r--p 00040000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f79faa82000-7f79faa88000 rw-p 00042000 08:03 1311239                    /lib/x86_64-linux-gnu/libreadline.so.7.0
7f79faa88000-7f79faa89000 rw-p 00000000 00:00 0
7f79faa89000-7f79faab2000 r-xp 00000000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f79faaee000-7f79faaef000 r--p 00000000 08:03 5376272                    /usr/lib/locale/C.UTF-8/LC_NUMERIC
7f79faaef000-7f79faaf0000 r--p 00000000 08:03 5376275                    /usr/lib/locale/C.UTF-8/LC_TIME
7f79faaf0000-7f79fac63000 r--p 00000000 08:03 5376261                    /usr/lib/locale/C.UTF-8/LC_COLLATE
7f79fac63000-7f79fac64000 r--p 00000000 08:03 5376268                    /usr/lib/locale/C.UTF-8/LC_MONETARY
7f79fac64000-7f79fac95000 r--p 00000000 08:03 5376263                    /usr/lib/locale/C.UTF-8/LC_CTYPE
7f79fac95000-7f79fac9c000 r--s 00000000 08:03 5381397                    /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7f79fac9c000-7f79facab000 rw-p 00000000 00:00 0
7f79facab000-7f79facac000 r--p 00000000 08:03 5376273                    /usr/lib/locale/C.UTF-8/LC_PAPER
7f79facac000-7f79facad000 r--p 00000000 08:03 5376270                    /usr/lib/locale/C.UTF-8/LC_NAME
7f79facad000-7f79facae000 r--p 00000000 08:03 5376260                    /usr/lib/locale/C.UTF-8/LC_ADDRESS
7f79facae000-7f79facaf000 r--p 00000000 08:03 5376274                    /usr/lib/locale/C.UTF-8/LC_TELEPHONE
7f79facaf000-7f79facb0000 r--p 00000000 08:03 5376265                    /usr/lib/locale/C.UTF-8/LC_MEASUREMENT
7f79facb0000-7f79facb1000 r--p 00000000 08:03 5376264                    /usr/lib/locale/C.UTF-8/LC_IDENTIFICATION
7f79facb1000-7f79facb2000 r--p 00000000 08:03 5376266                    /usr/lib/locale/C.UTF-8/LC_MESSAGES/SYS_LC_MESSAGES
7f79facb2000-7f79facb3000 r--p 00029000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f79facb3000-7f79facb4000 rw-p 0002a000 08:03 1316013                    /lib/x86_64-linux-gnu/ld-2.27.so
7f79facb4000-7f79facb5000 rw-p 00000000 00:00 0
7ffc65081000-7ffc650a2000 rw-p 00000000 00:00 0                          [stack]
7ffc650f0000-7ffc650f3000 r--p 00000000 00:00 0                          [vvar]
7ffc650f3000-7ffc650f5000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

We have:

```
  0x55555555466e <main+36>:    mov    $0x0,%eax
  0x555555554673 <main+41>:    callq  0x555555554520 <printf@plt>
```

Thus, we put a breakpoint: break *0x55555555466e
When we hit the breakpoint, the stack starts at 0x7fffffde000, and ends at 0x7fffffff000.

```
(gdb) break *0x55555555466e
Breakpoint 1 at 0x55555555466e
(gdb) run hello
Starting program: /home/vagrant/lab/first_input hello

Breakpoint 1, 0x000055555555466e in main ()
(gdb) info proc mappings
process 1926
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /home/vagrant/lab/first_input
      0x555555754000     0x555555755000     0x1000        0x0 /home/vagrant/lab/first_input
      0x555555755000     0x555555756000     0x1000     0x1000 /home/vagrant/lab/first_input
      0x7ffff79e2000     0x7ffff7bc9000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7bc9000     0x7ffff7dc9000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dc9000     0x7ffff7dcd000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcd000     0x7ffff7dcf000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000        0x0
      0x7ffff7dd3000     0x7ffff7dfc000    0x29000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7fee000     0x7ffff7ff0000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x29000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2a000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7fffffffe000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

We also look at rbp and rsp:
The rbp shows that the top of the stack is at 0x7fffffffe3e0, while the rsp shows that the address of the current top of the stack is at at 0x7fffffffe3d0.

```
(gdb) info registers
rax            0x7fffffffe73e   140737488348990
rbx            0x0      0
rcx            0x555555554680   93824992233088
rdx            0x7fffffffe4e0   140737488348384
rsi            0x7fffffffe73e   140737488348990
rdi            0x555555554704   93824992233220
rbp            0x7fffffffe3e0   0x7fffffffe3e0
rsp            0x7fffffffe3d0   0x7fffffffe3d0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554540   93824992232768
r13            0x7fffffffe4c0   140737488348352
r14            0x0      0
r15            0x0      0
rip            0x55555555466e   0x55555555466e <main+36>
eflags         0x212    [ AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

To make the input more obvious, my inout is 100 capital letters A:

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vagrant/lab/first_input AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x000055555555466e in main ()
```

With the instruction:

```
(gdb) x/1000x $rsp
```

I find my input (The hexadecimal representation for the capital letter A is 0x41.) from address “0x7fffffffe6e0" to "0x7fffffffe740":

```
0x7fffffffe6e0: 0x41414141      0x41414141      0x41414141      0x41414141
---Type <return> to continue, or q <return> to quit---
0x7fffffffe6f0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe700: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe710: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe720: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe730: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe740: 0x00414141      0x435f534c      0x524f4c4f      0x73723d53
```

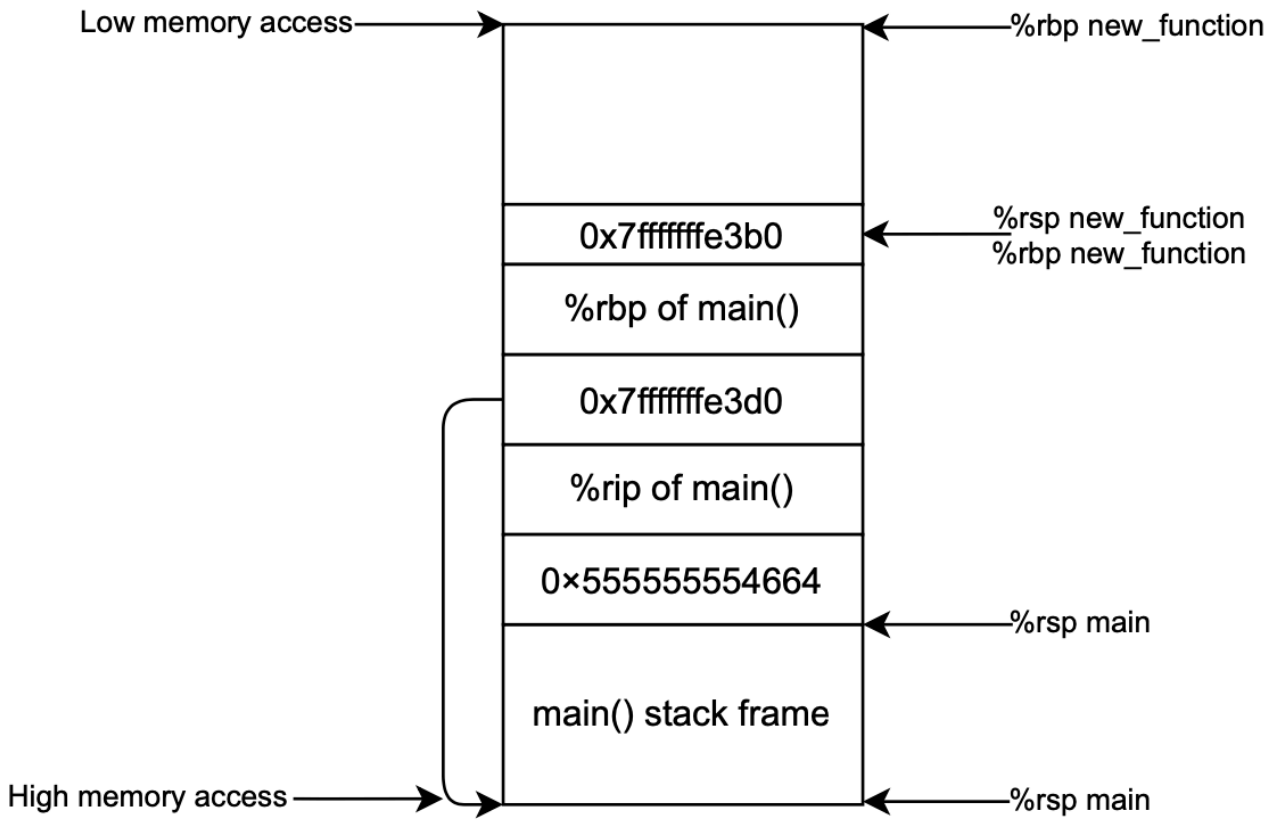
Question 2.19

```
(gdb) x/20i main
   0x5555555546a1 <main>:       push   %rbp
   0x5555555546a2 <main+1>:     mov    %rsp,%rbp
   0x5555555546a5 <main+4>:     sub    $0x10,%rsp
   0x5555555546a9 <main+8>:     mov    %edi,-0x4(%rbp)
   0x5555555546ac <main+11>:    mov    %rsi,-0x10(%rbp)
   0x5555555546b0 <main+15>:    mov    -0x10(%rbp),%rax
   0x5555555546b4 <main+19>:    add    $0x8,%rax
   0x5555555546b8 <main+23>:    mov    (%rax),%rax
   0x5555555546bb <main+26>:    mov    %rax,%rsi
   0x5555555546be <main+29>:    lea    0xa3(%rip),%rdi        # 0x555555554768
   0x5555555546c5 <main+36>:    mov    $0x0,%eax
   0x5555555546ca <main+41>:    callq  0x555555554560 <printf@plt>
   0x5555555546cf <main+46>:    mov    $0x0,%eax
   0x5555555546d4 <main+51>:    callq  0x55555555468a <new_function>
   0x5555555546d9 <main+56>:    mov    $0x0,%eax
   0x5555555546de <main+61>:    leaveq
   0x5555555546df <main+62>:    retq
   0x5555555546e0 <__libc_csu_init>:    push   %r15
   0x5555555546e2 <__libc_csu_init+2>:  push   %r14
   0x5555555546e4 <__libc_csu_init+4>:  mov    %rdx,%r15
(gdb) b *0x5555555546b4
Breakpoint 9 at 0x5555555546b4
```

```
(gdb) x/20i new_function
   0x55555555468a <new_function>:       push   %rbp
   0x55555555468b <new_function+1>:     mov    %rsp,%rbp
   0x55555555468e <new_function+4>:     lea    0xcf(%rip),%rdi        # 0x555555554764
   0x555555554695 <new_function+11>:    callq  0x555555554550 <puts@plt>
   0x55555555469a <new_function+16>:    mov    $0x0,%eax
   0x55555555469f <new_function+21>:    pop    %rbp
   0x5555555546a0 <new_function+22>:    retq
   0x5555555546a1 <main>:       push   %rbp
   0x5555555546a2 <main+1>:     mov    %rsp,%rbp
   0x5555555546a5 <main+4>:     sub    $0x10,%rsp
   0x5555555546a9 <main+8>:     mov    %edi,-0x4(%rbp)
   0x5555555546ac <main+11>:    mov    %rsi,-0x10(%rbp)
   0x5555555546b0 <main+15>:    mov    -0x10(%rbp),%rax
   0x5555555546b4 <main+19>:    add    $0x8,%rax
   0x5555555546b8 <main+23>:    mov    (%rax),%rax
   0x5555555546bb <main+26>:    mov    %rax,%rsi
   0x5555555546be <main+29>:    lea    0xa3(%rip),%rdi        # 0x555555554768
   0x5555555546c5 <main+36>:    mov    $0x0,%eax
   0x5555555546ca <main+41>:    callq  0x555555554560 <printf@plt>
   0x5555555546cf <main+46>:    mov    $0x0,%eax
(gdb) b *0x555555554695
Breakpoint 10 at 0x555555554695
```

```
(gdb) run hello
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vagrant/lab/first_input_mod hello

Breakpoint 9, 0x00005555555546b4 in main ()
(gdb) info registers
rax            0x7fffffffe4b8   140737488348344
rbx            0x0      0
rcx            0x5555555546e0   93824992233184
rdx            0x7fffffffe4d0   140737488348368
rsi            0x7fffffffe4b8   140737488348344
rdi            0x2      2
rbp            0x7fffffffe3d0   0x7fffffffe3d0
rsp            0x7fffffffe3c0   0x7fffffffe3c0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x0      0
r11            0x0      0
r12            0x555555554580   93824992232832
r13            0x7fffffffe4b0   140737488348336
r14            0x0      0
r15            0x0      0
rip            0x5555555546b4   0x5555555546b4 <main+19>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) continue
Continuing.
argv[1]: hello

Breakpoint 10, 0x0000555555554695 in new_function ()
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x555555756260   93824994337376
rdi            0x555555554764   93824992233316
rbp            0x7fffffffe3b0   0x7fffffffe3b0
rsp            0x7fffffffe3b0   0x7fffffffe3b0
r8             0x0      0
r9             0x5      5
r10            0xffffffffb      4294967291
r11            0x246    582
r12            0x555555554580   93824992232832
r13            0x7fffffffe4b0   140737488348336
r14            0x0      0
r15            0x0      0
rip            0x555555554695   0x555555554695 <new_function+11>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

We can see that for **new_function** we have that the addresses of rbp and rsp are the same: 0x7fffffffe3b0. This means that **new_function** is not actively using the stack for local variables. Therefore, the drawing of the corresponding stacks is as follows:


Low memory access
%rbp new_function
0x7fffffffe3b0
%rsp new_function
%rbp new_function
%rbp of main()
0x7fffffffe3d0
%rip of main()
0×555555554664
%rsp main
main() stack frame
High memory access
%rsp main