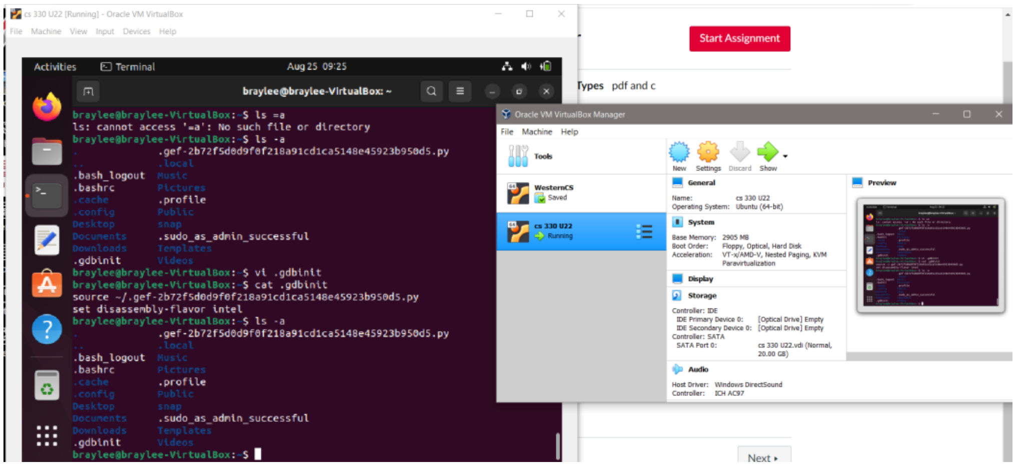
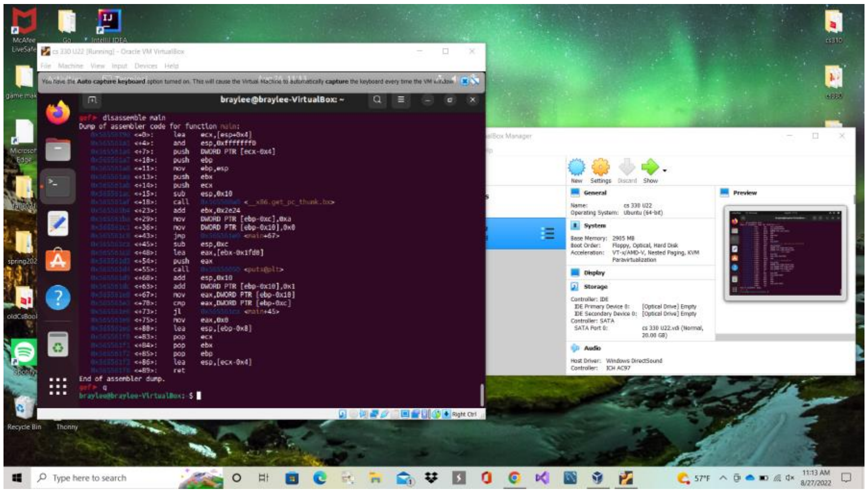
1)



```
braylee@braylee-VirtualBox:~$ ls =a
ls: cannot access '=a': No such file or directory
braylee@braylee-VirtualBox:~$ ls -a
.              .gef-2b72f5d0d9f0f218a91cd1ca5148e45923b950d5.py
..             .local
.bash_logout   Music
.bashrc        Pictures
.cache         .profile
.config        Public
Desktop        snap
Documents      .sudo_as_admin_successful
Downloads      Templates
.gdbinit       Videos
braylee@braylee-VirtualBox:~$ vi .gdbinit
braylee@braylee-VirtualBox:~$ cat .gdbinit
source ~/.gef-2b72f5d0d9f0f218a91cd1ca5148e45923b950d5.py
set disassembly-flavor intel
braylee@braylee-VirtualBox:~$ ls -a
.              .gef-2b72f5d0d9f0f218a91cd1ca5148e45923b950d5.py
..             .local
.bash_logout   Music
.bashrc        Pictures
.cache         .profile
.config        Public
Desktop        snap
Documents      .sudo_as_admin_successful
Downloads      Templates
.gdbinit       Videos
braylee@braylee-VirtualBox:~$
```

Name: cs 330 U22
Operating System: Ubuntu (64-bit)
Base Memory: 2905 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization
Controller: IDE
IDE Primary Device 0: [Optical Drive] Empty
IDE Secondary Device 0: [Optical Drive] Empty
Controller: SATA
SATA Port 0: cs 330 U22.vdi (Normal, 20.00 GB)
Host Driver: Windows DirectSound
Controller: ICH AC97



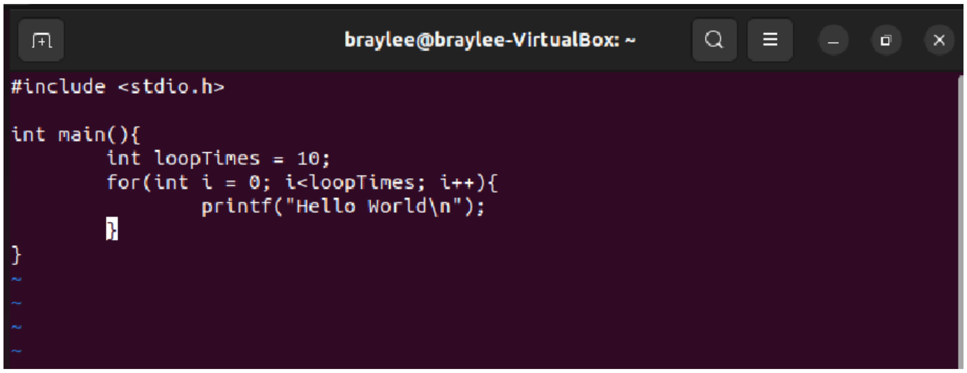
You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically **capture** the keyboard every time the VM window

```
gef➤  disassemble main
Dump of assembler code for function main:
   [illegible] <+0>:     lea    ecx,[esp+0x4]
   [illegible] <+4>:     and    esp,0xfffffff0
   [illegible] <+7>:     push   DWORD PTR [ecx-0x4]
   [illegible] <+10>:    push   ebp
   [illegible] <+11>:    mov    ebp,esp
   [illegible] <+13>:    push   ebx
   [illegible] <+14>:    push   ecx
   [illegible] <+15>:    sub    esp,0x10
   [illegible] <+18>:    call   [illegible] <__x86.get_pc_thunk.bx>
   [illegible] <+23>:    add    ebx,0x2e24
   [illegible] <+29>:    mov    DWORD PTR [ebp-0xc],0xa
   [illegible] <+36>:    mov    DWORD PTR [ebp-0x10],0x0
   [illegible] <+43>:    jmp    [illegible] <main+67>
   [illegible] <+45>:    sub    esp,0xc
   [illegible] <+48>:    lea    eax,[ebx-0x1fd0]
   [illegible] <+54>:    push   eax
   [illegible] <+55>:    call   [illegible] <puts@plt>
   [illegible] <+60>:    add    esp,0x10
   [illegible] <+63>:    add    DWORD PTR [ebp-0x10],0x1
   [illegible] <+67>:    mov    eax,DWORD PTR [ebp-0x10]
   [illegible] <+70>:    cmp    eax,DWORD PTR [ebp-0xc]
   [illegible] <+73>:    jl     [illegible] <main+45>
   [illegible] <+75>:    mov    eax,0x0
   [illegible] <+80>:    lea    esp,[ebp-0x8]
   [illegible] <+83>:    pop    ecx
   [illegible] <+84>:    pop    ebx
   [illegible] <+85>:    pop    ebp
   [illegible] <+86>:    lea    esp,[ecx-0x4]
   [illegible] <+89>:    ret
End of assembler dump.
gef➤  q
braylee@braylee-VirtualBox:~$
```

McAfee LiveSafe
Go
IntelliJ IDEA
cs 330 U22 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
braylee@braylee-VirtualBox: ~
Right Ctrl
Thonny
Recycle Bin
cs310
Type here to search
57°F
11:13 AM
8/27/2022

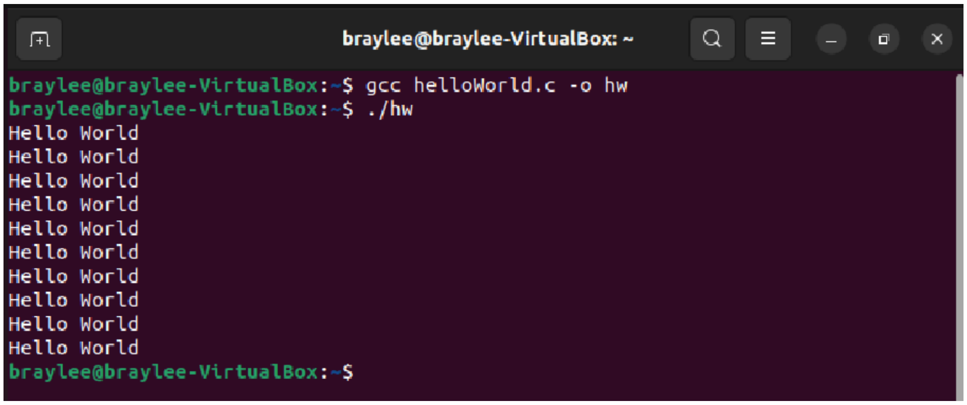
This is the environment I made. On the right is the machine shown running in virtual box and the left shows the terminal running after installing the necessary bits. This machine is stored on Oracle's virtual box. The second image in this series shows my personal computer desktop running in the background so we can see it running on my machine and not someone else's.

2)



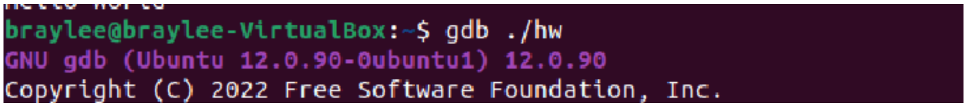
```
#include <stdio.h>

int main(){
        int loopTimes = 10;
        for(int i = 0; i<loopTimes; i++){
                printf("Hello World\n");
        }
}
~
~
~
~
```

This is the c code I used to print Hello World to the console 10 times created in vi. It is called helloWorld.c for future reference. This is the base code that will be compiled as a 64 and 32 bit executable for the rest of this problem.

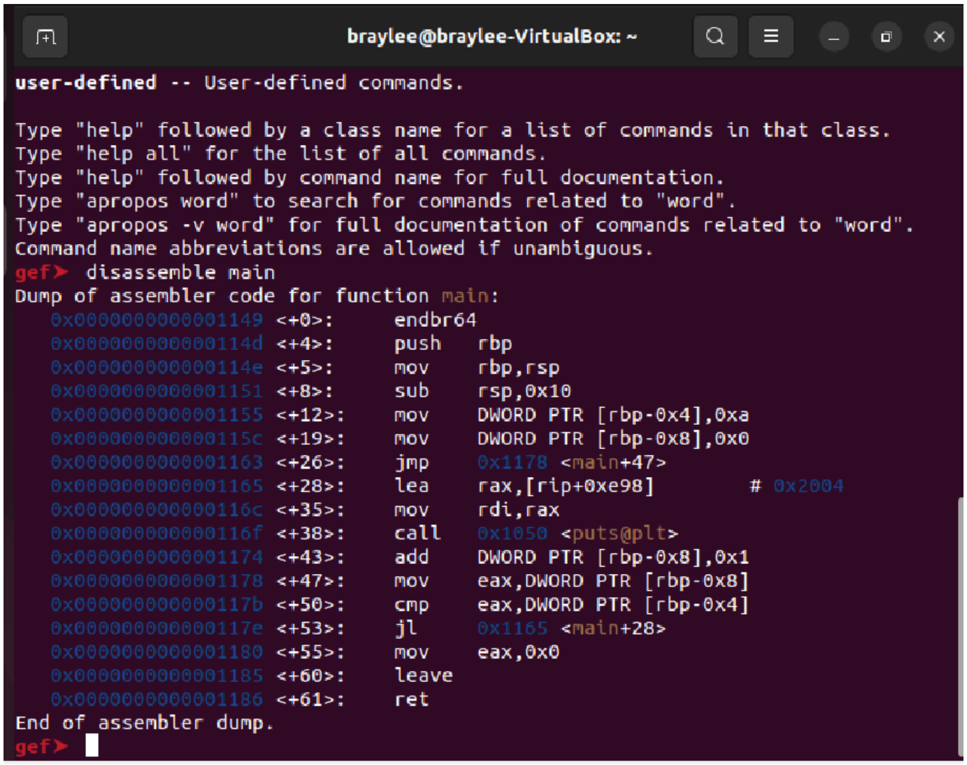


```
braylee@braylee-VirtualBox:~$ gcc helloWorld.c -o hw
braylee@braylee-VirtualBox:~$ ./hw
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
braylee@braylee-VirtualBox:~$
```

This is how the code runs when it is run as a 64-bit executable. The executable is named ./hw. It prints out Hello World ten times. We know this is a 64-bit executable because the operating system is a 64-bit so the ISA is 64-bit on default.

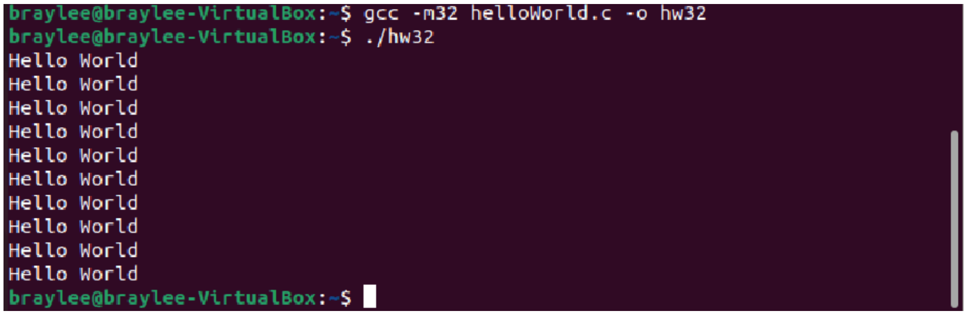


```
braylee@braylee-VirtualBox:~$ gdb ./hw
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
```



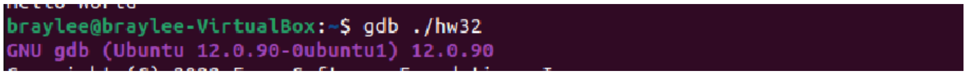
```
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     sub    rsp,0x10
   0x0000000000001155 <+12>:    mov    DWORD PTR [rbp-0x4],0xa
   0x000000000000115c <+19>:    mov    DWORD PTR [rbp-0x8],0x0
   0x0000000000001163 <+26>:    jmp    0x1178 <main+47>
   0x0000000000001165 <+28>:    lea    rax,[rip+0xe98]        # 0x2004
   0x000000000000116c <+35>:    mov    rdi,rax
   0x000000000000116f <+38>:    call   0x1050 <puts@plt>
   0x0000000000001174 <+43>:    add    DWORD PTR [rbp-0x8],0x1
   0x0000000000001178 <+47>:    mov    eax,DWORD PTR [rbp-0x8]
   0x000000000000117b <+50>:    cmp    eax,DWORD PTR [rbp-0x4]
   0x000000000000117e <+53>:    jl     0x1165 <main+28>
   0x0000000000001180 <+55>:    mov    eax,0x0
   0x0000000000001185 <+60>:    leave
   0x0000000000001186 <+61>:    ret
End of assembler dump.
gef➤
```

This is the disassembly of the 64-bit executable. The first command in this stream was gdb ./hw where ./hw was the executable in the 64-bit form. To disassemble the main, I used the command disassemble main with the gdb open and running in the command line. (Note: this screenshot is in two because between these commands I was messing around in help to see what was all available in gdb/gef).

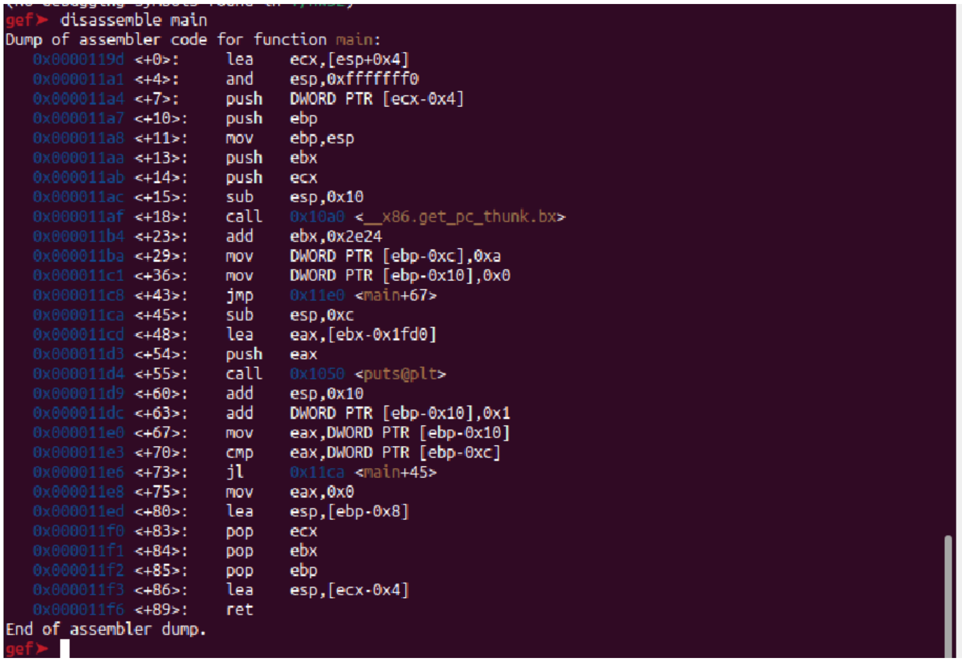


```
braylee@braylee-VirtualBox:~$ gcc -m32 helloWorld.c -o hw32
braylee@braylee-VirtualBox:~$ ./hw32
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
braylee@braylee-VirtualBox:~$
```

Here is it compiled and running as a 32-bit executable. Note the executable for this set is called ./hw32. It prints out Hello World 10 times. The first image in this series shows the code used to print this. The difference here is that it is compiled as a 32 bit with the option –m32 in the command line.



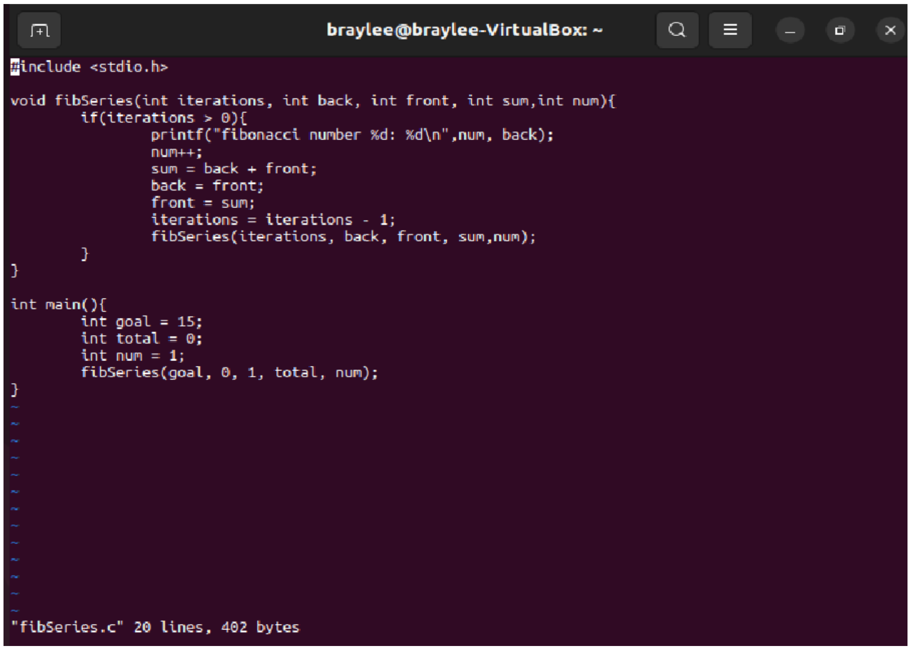
```
braylee@braylee-VirtualBox:~$ gdb ./hw32
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
```



```
(No debugging symbols found in ./hw32)
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000119d <+0>:     lea    ecx,[esp+0x4]
   0x000011a1 <+4>:     and    esp,0xfffffff0
   0x000011a4 <+7>:     push   DWORD PTR [ecx-0x4]
   0x000011a7 <+10>:    push   ebp
   0x000011a8 <+11>:    mov    ebp,esp
   0x000011aa <+13>:    push   ebx
   0x000011ab <+14>:    push   ecx
   0x000011ac <+15>:    sub    esp,0x10
   0x000011af <+18>:    call   0x10a0 <__x86.get_pc_thunk.bx>
   0x000011b4 <+23>:    add    ebx,0x2e24
   0x000011ba <+29>:    mov    DWORD PTR [ebp-0xc],0xa
   0x000011c1 <+36>:    mov    DWORD PTR [ebp-0x10],0x0
   0x000011c8 <+43>:    jmp    0x11e0 <main+67>
   0x000011ca <+45>:    sub    esp,0xc
   0x000011cd <+48>:    lea    eax,[ebx-0x1fd0]
   0x000011d3 <+54>:    push   eax
   0x000011d4 <+55>:    call   0x1050 <puts@plt>
   0x000011d9 <+60>:    add    esp,0x10
   0x000011dc <+63>:    add    DWORD PTR [ebp-0x10],0x1
   0x000011e0 <+67>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000011e3 <+70>:    cmp    eax,DWORD PTR [ebp-0xc]
   0x000011e6 <+73>:    jl     0x11ca <main+45>
   0x000011e8 <+75>:    mov    eax,0x0
   0x000011ed <+80>:    lea    esp,[ebp-0x8]
   0x000011f0 <+83>:    pop    ecx
   0x000011f1 <+84>:    pop    ebx
   0x000011f2 <+85>:    pop    ebp
   0x000011f3 <+86>:    lea    esp,[ecx-0x4]
   0x000011f6 <+89>:    ret
End of assembler dump.
gef➤
```

This is the gdb disassembly for main for the 32-bit. The gdb was opened with gdb ./hw32 where ./hw32 is the executable we made that used a 32-bit. To disassemble main, I used the command disassemble main shown at the top of the image. (Note: the image is in two to cut out the help messages that come when you first open gdb).

3)



```
#include <stdio.h>

void fibSeries(int iterations, int back, int front, int sum,int num){
        if(iterations > 0){
                printf("fibonacci number %d: %d\n",num, back);
                num++;
                sum = back + front;
                back = front;
                front = sum;
                iterations = iterations - 1;
                fibSeries(iterations, back, front, sum,num);
        }
}

int main(){
        int goal = 15;
        int total = 0;
        int num = 1;
        fibSeries(goal, 0, 1, total, num);
}
~
~
~
~
~
~
~
~
~
~
~
~
~
"fibSeries.c" 20 lines, 402 bytes
```

This is the recursive program I wrote for this problem. It is named fibSeries.c for future reference in the command line (can be seen in bottom left of image). The function fibSeries is the recursive as fibSeries is called within void fibSeries.

Variable notes:

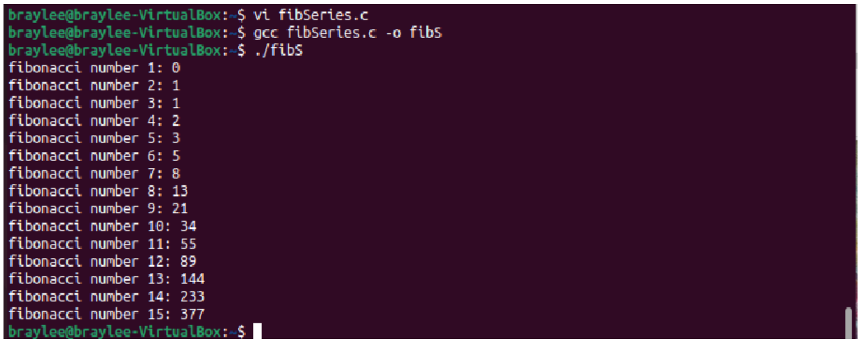
Goal/iterations: used to dictate how many of the Fibonacci numbers we wanted printed in a row. Also serves to stop the recursion so that it does not run infinitely.

Back: initially set to 0 because the first number of series is 0. Used to keep track of previous values as the Fibonacci sequence is built by adding the two previous numbers together. This is the value I printed because it will eventually be updated to the next Fibonacci number, and it starts at the first number in the sequence 0.

Front: keeps track of the next number in sequence. Initially set to 1 because the second number of the sequence is known to be 1.

Sum/total: keeps track of the summation of back and front to get the next value in sequence. Initially it is zero because we have not started building the sequence when we call the function.

Num: I used num to make the print statement easier to read. This keeps track of what number in the sequence our printed value is. Starts at 1 and continues to 15 in this case to show the first 15 Fibonacci numbers.



```
braylee@braylee-VirtualBox:~$ vi fibSeries.c
braylee@braylee-VirtualBox:~$ gcc fibSeries.c -o fibS
braylee@braylee-VirtualBox:~$ ./fibS
fibonacci number 1: 0
fibonacci number 2: 1
fibonacci number 3: 1
fibonacci number 4: 2
fibonacci number 5: 3
fibonacci number 6: 5
fibonacci number 7: 8
fibonacci number 8: 13
fibonacci number 9: 21
fibonacci number 10: 34
fibonacci number 11: 55
fibonacci number 12: 89
fibonacci number 13: 144
fibonacci number 14: 233
fibonacci number 15: 377
braylee@braylee-VirtualBox:~$ 
```

This image shows the output of the fibSeries.c code. We can see that it is executed with gcc fibSeries.c – o fibs. The executable is run with ./fibs.

4)

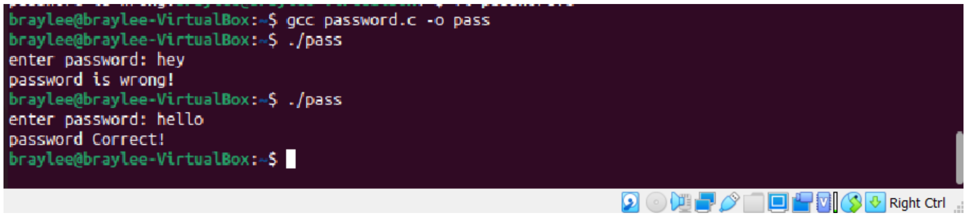
This program is uploaded as the password.c separately.

Quick notes:

The correct password is hello

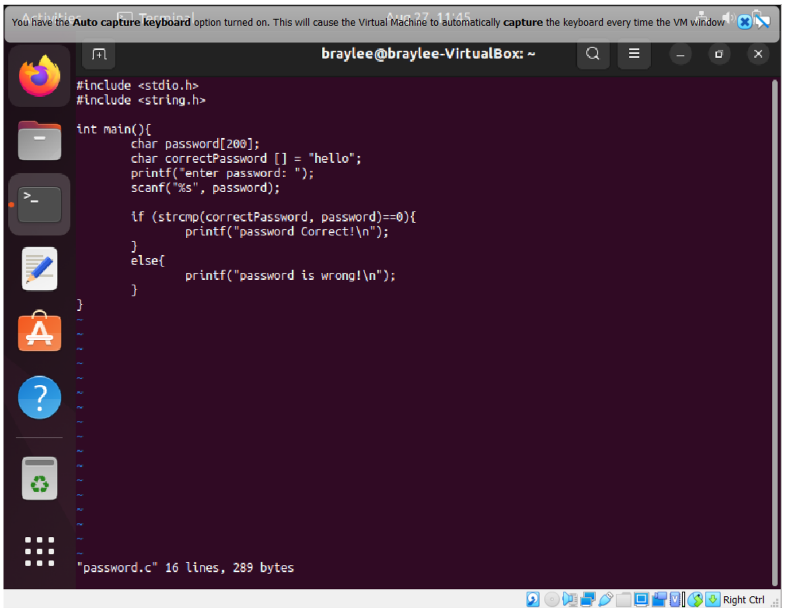
The success message is password Correct!

The error message is password is wrong!



```
braylee@braylee-VirtualBox:~$ gcc password.c -o pass
braylee@braylee-VirtualBox:~$ ./pass
enter password: hey
password is wrong!
braylee@braylee-VirtualBox:~$ ./pass
enter password: hello
password Correct!
braylee@braylee-VirtualBox:~$ 
```

The code for my future reference looks like this



You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically **capture** the keyboard every time the VM window

```
#include <stdio.h>
#include <string.h>

int main(){
        char password[200];
        char correctPassword [] = "hello";
        printf("enter password: ");
        scanf("%s", password);

        if (strcmp(correctPassword, password)==0){
                printf("password Correct!\n");
        }
        else{
                printf("password is wrong!\n");
        }
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"password.c" 16 lines, 289 bytes
```