Lesson 5: User Defined Functions and Stack Mechanics

Objectives:
(a) Demonstrate the ability to analyze simple programs that use library and user defined functions.
(b) Describe the organization and contents of a program’s stack throughout a program’s execution.
(c) Demonstrate the ability to examine the stack values of a running program using the debugger.

1. Functions

It is often best to solve a large problem by successively decomposing it into smaller and smaller sub-problems. In this manner we introduce and implement functions in C programs. A function is a small subprogram that performs a specific task in a program.

Functions promote the writing of efficient programs. If we have large tasks that we want our program to accomplish, we can break it down into small, more manageable subtasks which individual functions can solve. The use of functions makes the C programming language more useful and robust. There are three types of functions that we discuss in this chapter: the main function, library functions and user-defined functions.

1.1 int main () Since we introduced C programs in Chapter 2 we have included the function main in every C program. The main function is a very special function, as it is the only function required in a C program and is the point of entry from which every C program begins executing. The contents of the body of main can vary widely as we have seen from chapter to chapter, but the structure always follows this form:

```c
int main () // Main function
{ 
    variable declarations; // body of main
    statement 1;
    (etc., etc.)
    statement n;
}
```

1.2 Library Functions To date we have seen C programs which consist solely of the main function and/or import C libraries to utilize pre-defined library functions like printf, scanf or strcpy by adding a #include line at the beginning of the program for each C library we draw library functions from. The simple program layout changes when using C library functions to the following form:

```c
#include <library> // Included C Libraries (Optional)
int main( ) // Main Function(Required)
{ 
    variable declarations;
    statement 1;
    (etc., etc.)
    statement n;
}
```
The library functions definitions are standardized, and to properly use a library in a C program the include <library> verbiage must be written at the beginning of the program. Recall that we first encountered the use of C libraries in Chapter 2 when we included the standard input/output library (stdio.h) in order to be able to utilize the printf and scanf functions for output and input statements. For example:

```c
#include <stdio.h>   // Necessary to utilize printf
int main ()     // Main Function
{
    printf ("Hello World!"); // referencing a library function
}
```

To reiterate, if we had a C program which did not utilize these library functions, the include <library> statement would not be necessary at the start of a C program. The C program below is correct and would compile without issue.

```c
int main ()     // Main Function
{
    int a = 4, b = 10, c;
    c = a + b;
}
```

1.3 User Defined Functions The case will certainly arise where we would find library functions limiting and want to define our own. The C programming language allows you to define functions according to your need. The layout of a C program utilized user defined functions would be of the form:

```c
#include <library>   // Included C Libraries (as needed)
<Function declaration>     // User defined function (Optional)
{                        // body of function
    body of function
}
int main( )    // Main Function (Required)
{
    variable declarations;
    statement 1;
    (etc., etc.) // a function call is an example
    statement n;  // of a statement
}
```

Note that much like a variable must be declared prior to its use in a C program, a function must be declared prior to its use. The function declaration will precede main in the C program.

2. Programming with Functions

2.1 Defining a Function To use our own functions, we must write the code that specifies the function to perform its required task. The function definition specifies the operations the function performs.

Syntax to Declare a User Defined Function The syntax for a user defined function in a C program is:

```
type Returned function_name (type_1 parameter_1, ... type_n parameter_n)
{
    body of function
}
```

**Type Returned** The *type returned* is the data type of the output value returned by the function

**Function Parameters** The *parameters* of a function are used in the body of the function definition and should be thought of as a placeholder that stands in for the inputs to a function. This means that in the execution of the program wherever we see parameters, the value of the inputs to the function (function arguments defined in 2.2.2.) will be plugged in. The type of each parameter must agree with the variable type of each input.
2.1.1 The Body of a Function
The syntax that has governed our C programs up to this point continue with the body of our function, with all variable declarations in the function coming before any statements. The variables declared in our function definition are known as function variables.

```c
{  
    variable declarations;
    statement 1;
    (etc., etc.)
    statement n;
    return statement;
}
```

The Return Value and Statement
The value a function computes for the program that calls it is called a return value. The return value can be thought of as the output of a function. Functions can have at most one return value but may also have no return value. The data type of the return value must agree with the type returned in the function definition.

The return statement consists of the keyword return followed by the name of one of the functions variables or an expression. This is what is returned as the return value to the function call. In other words the value of the expression after the return statement is the functions output.

2.1.2 Void functions
Functions which have no return value are called void functions. The word void will be used in the function definition in place of a type returned. Void functions do not require a return statement in the body of the function.

2.2 Utilizing a Function

2.2.1 Function calls
To use a function we must invoke it with a function call. The function call will fall into the category of “statements” in the body of the main function. The function call specifies:

- The function name- which is tied to the function definition instructions to be executed.
- The function arguments- the inputs provided to a function.

Syntax to call a function.
The syntax for a function call is:

```
function_name(fxn_arg_1, fxn_arg_2, ..., fxn_arg_n);
```

2.2.2 Function arguments (arg)
The function arguments, abbreviated arg, are the inputs to a function. The function arguments can be numbers, variables, or other more complicated statements.

An Example. Suppose we had previously defined a function named abs_val used to compute and return the absolute value of a number. In the body of main we would invoke it as follows:

```
y = abs_val(x);
```

The statement abs_val(x) would be the function call and the variable x would be the function argument. The variable y would be assigned the return value. We said that function arguments could be numbers or variables, so the following would also be correct

```
y = abs_val(-4);
```

where -4 is a number which is not a program variable. In both cases the variable y, is being assigned the return value from the function call.

A note on function calls
There must be an agreement on the type returned from the function definition and the variable type used in the assignment statement (i.e. the = sign). In the example above, if abs_val is defined to return an integer, then y must be a variable of type integer. For void functions, the function call will not have an assignment statement because there is no return value.
### Practice Problem 5.1

Circle the appropriate words to complete the statements below. Each set of bold terms separated with a slash indicates that you should select one of the choices.

1. To use a function we must invoke it with a **return value** / function call / prototype.

2. The **values** / parameters / arguments are the inputs to a function.

3. A value / parameter / argument is a placeholder that “stands in” for a value / parameter / argument.

4. The result from a function is called the **return value** / function call / prototype.

### 2.3 Utility of Functions, An Example

It may not immediately be clear what the advantage of using a function is compared to simply including the code in the body of main. Functions are extremely useful for code reuse. Once a function has been defined it can be called multiple times in a program. Consider this example:

**Without User Defined Functions:**
```c
int main ()
{
    int a = 1986, x;
    x = a*a +2*a+ 8;
}
```

**With a User Defined Function (quadratic_fxn):**
```c
int quadratic_fxn(int j)  // Function Definition
{      // Execute these calculations each
    int output;    // time the function call is
    output = j*j+2*j+ 8;  // invoked
    return output;
}
int main ()
{
    int a = 1986;
    int x;
    x = quadratic_fxn(a);  // Function Call (use Fxn Defn)
}
```

This may look longer at first glance. Consider though if you had multiple main variables that you wanted to perform the same operation on. You would be able to do this with a simple user defined function and a function call.

### Practice Problem 5.2

You currently have the following C program.

```c
#include<stdio.h>
int main()
{
    int x, y;
    printf("Enter an Integer: ");
    scanf("%d", &x);
    if (x >=0)
        y = x;
    else
        y = x*-1;
}
```

We want to new C program that utilizes a user-defined function to perform the if/else computation. The body of main has been written for you. Write the body of the function for `abs_val`
The mechanics of how functions are handled by the CPU and in memory is the last piece we need to know to understand the vulnerability of functions to a **buffer overflow attack**.

### 3. Mechanics of Executing a Function and the Stack

We recall there are two distinct sections in main memory when a program is executed, the text segment and the stack. When the command is given to execute a program from the command line, a copy of the machine language instructions for the program, is placed in main memory and forms the text segment for the program. The CPU works though the machine language instructions in the text segment using the “fetch, decode, execute” cycle. The stack is additional memory to execute programs. The stack serves as our “scratch pad” and is where program variables are stored.

When we have a C program with multiple functions, each function will have its own text segment and stack frame. As we have seen with our examples, it is possible for the body of each function in a C program to have its own variables and thus, it is necessary to have a stack frame for each function to store its variables. (Thinking back to our recipe analogy, if you were to make frosting to accompany your cake, you would have a new recipe, different ingredients and use a new mixing bowl!) We will again use our debugger to perform a program autopsy and gain a better sense of what is happening in main memory as the program is executed. Having multiple text segments and stack frames makes for a more complex process. Using the debugger we will see the conventions for dealing with this challenge in an efficient manner.

**Program Autopsy with User Defined Functions** We will use the debugger again to run this program one line at a time and at each step in the process see how the main memory mechanics change with the use of user-defined functions in the program.

**Step 1:** Today you will use the program *fxn_demo.c*, which has been written for you and placed in the *ec310code* directory. Copy this file to the *work* directory by carefully entering the following at the home directory prompt:

```
 cp ec310code/fxn_demo.c  work
```

Verify that you have *fxn_demo.c* in your *work* directory by changing to the *work* directory and then listing the files in the work directory:

```
 cd work
 ls
```

Compile and execute the program by entering the following commands:

```
 gcc -g -o fxn_demo.exe fxn_demo.c
 ./fxn_demo.exe
```
Look at our program for today in *nano*, you will see this file contain the code from section 2.3.

```c
int quadratic_fxn (int j)
{
    int output;
    output = j*j+ 2*j +8;
    return output;
}

int main()
{
    int a = 1986;
    int x;
    x = quadratic_fxn(a);
}
```

We see that our program has 2 functions- *main* and *quadratic_fxn*. It is helpful to “diagram” the C program and identify many of the components discussed in this chapter, as seen in the following:

Step 2- Two Functions in Main Memory: We will begin by starting the debugger by entering the following command:

```
gdb -q ./fxn_demo.exe
```

When the program is executed, the text segments for both functions are placed in main memory. To see how the function *quadratic_fxn* is stored in memory as assembly code, we enter:

```
disass quadratic_fxn
```

You should see this:
Now let’s examine how main function is stored in memory. Enter:

```
disass main
```

You should see this:

```
The Importance of Main: From Section 1.1, one role of the main function is that it serves the point of entry into the program. From the output of disass main, the first instruction of the main function has been placed in main memory at the address 0x0804835f. When the program is run, the CPU begins executing the program from this address. As the program progresses the eip register advances as each instruction is fetched, decoded and executed.

Step 3- Storing Main Variables: When we begin executing our program at main, we see that we have two variables declared; a, with a value of 1986 on line 9 and x, which currently has a garbage value on line 10.
```

```
When this C program is compiled, the compiler will allocate memory on the stack based on the variable declarations in the program and there will be machine language instructions that store these variables on the stack. From previous security exercises, when we view the associated assembly language instructions, we expect the instruction to be of the form:

```
mov DWORD PTR [ebp-#], 0x(value)
```

Run the program up to the point that the main variables have not yet been stored on the stack. Enter:

```
break 9
run
```

and then to view the addresses of the processor registers at the break point, enter:

```
ir eip esp ebp
```

You should see the following:
The `ebp` and `esp` processor registers hold the addresses of the top and bottom of the stack. The address of the next instruction to be executed is `0x0804836f` which corresponds to the assembly language instruction `mov DWORD PTR [ebp-4], 0x7c2`. In this assembly instruction, the value `0x7c2` is the hexadecimal equivalent of 1986, which is the value assigned to variable `a`. Execute this instruction and examine the stack, confirming the hexadecimal value `0x7c2` is on the stack at the location pointed to by `ebp-4`, by entering the following:

```
nexti
x/4xb 0xbffff804
```

At this point, the value `0x7c2` has been placed on the stack for the variable `a`, and the stack looks like the following diagram.

As `x` does not yet have a value, there is no instruction to place a value for it on the stack, and the value at its location in memory is garbage. The stack mechanics dictate that the stack is built from the bottom up, which means that the first variable declared in a C program is placed on the bottom of the stack and built upon from there.

**Step 4: Function Arguments:** After all main variables are placed on the stack, the function arguments are placed on the stack. If a function argument is a main variable, this will result in a copy of the variable’s value placed on the stack.

Execute two instructions to advance the program to the function call as follows:

```
Nexti 2
```

The two instructions executed here are responsible for copying the value of the main variable `a` and placing it in on the stack frame for `main` as a function argument.

We are now able to confirm the organization of the stack frame for `main` with the command:

```
x/24b $esp
```

which examines the 24 bytes in main memory (size of the stack frame `main`) starting from the address that the stack pointer holds.
Step 5- Preparing for a function call: The program continues executing up to the point in the text segment of `main` when the function call is reached. The instruction at the address is `0x0804837c`, and the assembly language at this address is `call 0x08048344 <quadratic_fxn>`, as seen in the following debugger display of the main assembly code:

Based on the debugger display of those 24 bytes, our diagram of the stack would now look like the following:

```
(gdb) i r eip
eip    0x804837c  0x804837c <main+29>
(gdb) x/i $eip
0x804837c <main+29>: call 0x8048344 <quadratic_fxn>
(gdb) x/24b $esp
0xbffff7f0: 0xc2 0x07 0x00 0x00 0xa0 0x83 0x04 0x08
0xbffff7f8: 0x00 0x00 0x00 0x00 0xf4 0x5f 0xfd 0xb7
0xbffffff0: 0xe0 0x0c 0x0b 0x08 0xc2 0x07 0x00 0x00
```

```
(gdb) i r eip
0xbffff7f0 0xc2 0x07 0x00 0x00 0xa0 0x83 0x04 0x08
0xbffff7f8 0x00 0x00 0x00 0x00 0xf4 0x5f 0xfd 0xb7
0xbffffff0 0xe0 0x0c 0x0b 0x08 0xc2 0x07 0x00 0x00
```

```
(gdb) i r eip
0xbffff7f0 0xc2 0x07 0x00 0x00 0xa0 0x83 0x04 0x08
0xbffff7f8 0x00 0x00 0x00 0x00 0xf4 0x5f 0xfd 0xb7
0xbffffff0 0xe0 0x0c 0x0b 0x08 0xc2 0x07 0x00 0x00
```

Based on the debugger display of those 24 bytes, our diagram of the stack would now look like the following:

![Stack Diagram](image_url)
What is the meaning of this instruction? Look again at the disass quadratic_fxn output:

This is the address of the first instruction in the text segment for the user-defined function. When the CPU executes this instruction, eip will advance to hold this address, that is to say, after this instruction is executed, the next instruction to be executed will be the first instruction in the text segment for the quadratic function.

Consider the difficulty the CPU now encounters. The processor registers eip, ebp, esp each have the capacity to hold exactly only one address at a time.

- The next instruction is not in the text segment for main and is out of order. After the instructions in the function call are complete, what value will eip hold next?

- We are going to a different function that has its own variables and requires its own stack frame. The ebp and esp processor registers hold the address of the top and the bottom of the stack, which means they will now hold the address of the top and bottom of the stack frame for the user defined function quadratic_fxn, not main.

Making the jump to the text segment for the user defined function is a straightforward process. But in order to “pick up where we left off” in main, there are two values we need to record before we jump.

- The proper return address for eip to hold after the function call instructions are complete so we may return to the text segment for main. The proper return address is the address of the instruction in main that immediately follows the function call. In our example it would be the address 0x08048381

- The prior value of the base pointer ebp (ebp_main) so that the ebp register may again hold the address of the bottom of the stack frame for main.

The stack is our “scratch pad” and thus we continue to build the stack from the bottom up placing the return address and prior value of ebp above the function arguments. In order to examine these in memory, let’s move onto the stack frame of quadratic_fxn.
The diagram of the stack is now as follows:

In order to properly view the return address in memory, let’s move into the stack frame of `quadratic_fxn` (execute the function call) by entering the following commands:

```
break 3
continue
x/8xb 0xbffff7e8
```

Step 6- The stack frame of `quadratic_fxn`: With the return address and prior ebp placed on the stack, eip, ebp, and esp can hold the values for instructions in the text segment of the user-defined function and of the top and bottom of the stack frame for `quadratic_fxn`. To examine the stack frame for the function `quadratic_fxn`, we choose a break point at the end of the function, just prior to returning to `main`.

```
(gdb) x/8b 0xbffff7e8
0xbffff7e8: 0x08 0xf8 0xff 0xbf 0x81 0x83 0x04 0x08
```

In the debugger, enter the following commands to set the break point and continue program executions, then view the contents of eip, ebp and esp, and to verify that the correct value of the function’s variable `output` is now on the stack.

```
break 6
continue
i r eip ebp esp
x/xw $esp
```
Chapter 5: User Defined Functions and Stack Mechanics

The output of the debugger and the complete diagram of the stack are as follows:

```
(gdb) break 6
Breakpoint 2 at 0x804835d: file fxn_demo.c, line 6.
(gdb) continue
Continuing.
Breakpoint 2, quadratic_fxn (j=1986) at fxn_demo.c:6
6 }
(gdb) i r ebp esp
ebp 0x804835d 0x804835d <quadratic_fxn+25>
ebp 0xbfffe7e8 0xbfffe7e8
esp 0xbfffe7e4 0xbfffe7e4
(gdb) x/xw $esp
0xbfffe7e4: 0x003c3e90
```

The diagram of the stack is now as follows:

![Stack Diagram]

**Step 7: Returning to main:** From the function definition of `quadratic_fxn` we know that the return value of the function call is an integer, and the return value is being assigned to the integer `x`. Return to main by entering the following commands and explore the stack frame for `main` once again. Note that line 12 is the last line in the body of `main`, so it is chosen as a breakpoint to return to.

```
break 12
continue
i r ebp esp
x/24xb $esp  //24 bytes corresponds to the size of the stack frame
```
Chapter 5: User Defined Functions and Stack Mechanics

The output of the debugger and the complete diagram of the stack are as follows:

```
(gdb) break 12
Breakpoint 3 at 0x8048384: file fxn_demo.c, line 12.
(gdb) continue
Continuing.

Breakpoint 3, main () at fxnDemo.c:12
12 }
(gdb) i r eip ebp esp
eip 0x8048384 0x8048384 <main+37>
ebp 0xbffff808 0xbffff808
esp 0xbffff7f0 0xbffff7f0
(gdb) x/24b$esp
0xbffff7f0: 0xc2 0x67 0x80 0xa0 0x03 0x04 0x08
0xbffff7f8: 0x00 0x00 0x08 0x00 0x4f 0x5f 0xffffffff 0xb7
0xbffff800: 0x90 0x3e 0x3c 0x80 0xc2 0x07 0x00 0x00
```

The diagram of the stack is now as follows:

```
Note: We only need to store the prior ebp for main and the return address, not the esp for main. This is because the ebp of the function will always point to the address where the prior ebp for main is stored (which is 4 bytes), and that sits on top of the return address (also 4 bytes). Directly below the return address is the address of esp for main. Mathematically speaking, then, to reset the esp for the esp of main, esp_main = ebp_fxn + 8.

In this example, the address eip now holds corresponds to the assembly language instruction leave (as seen below). We have reached the end of the fxn_demo program. This completes our program autopsy with functions.
```

```
{gdb} i r eip
0x8048384 <main+37>: leave
0x8048385 <main+38>: ret
```
Lesson 5 Problems

Name:_________________________

1. From your home directory copy the program hwk7.c which has been written for you and placed in the ec310code directory by entering:

   midshipman@EC310:~ $ cp ec310code/hwk7.c work

   Now change your directory back to your work directory by typing cd work

The program hwk7.c is shown below:

```c
void test_function( int a, int b, int c, int d)
{
    int flag;
    char buffer[8];

    flag = 31337;
    buffer[0] = 'A';
}

int main( )
{
    int w=1, x=2, y=3, z=4;
    test_function(w, x, y, z) ;
}
```

Compile your program and open the debugger by entering:

```bash
gcc -g -o hwk7.exe hwk7.c

gdb -q ./hwk7.exe
```

(a) Now we want to run the program up to the call to test_function – that is, the breakpoint should correlate with the line that reads test_function(w,x,y,z), which is line 12.

   break 12
   run
   nexti 8

On the chart on the following page, identify the address of ebp (main) and esp (main), and identify where the main variables and the arguments to the test function are actually stored.

(b) Run the program up to the point of reaching the closing brace of test_function. To do this, enter:

   break 8
   continue

Show the contents of the stack frame for test_function as well as the additional memory locations below the base pointer. Use the figure on the next page. Show the location of the base pointer and stack pointer on the chart. Label on your chart the following:

- the location of flag
- the location of buffer
- the location of the return address
- the location of the prior value of the base pointer
- the location of all your function arguments
- the value of ebp(test_function) and esp(test_function)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bffff7d0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7d4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7d8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7dc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7e0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7e4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7e8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7ec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7f0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7f4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff7fc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff804</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff808</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff80c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff810</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff814</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bffff818</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>