Objectives:
(a) Demonstrate the ability to analyze simple programs using functions, properly identifying function definition, function call, return value, arguments, and parameters.
(b) Given the source code of a C program which contains functions and the output of the debugger, draw a diagram of the stack (both before and after the function call) with proper ordering of main variables, function arguments, saved base pointer, return address, and function variables.
(c) Demonstrate the ability to examine the stack values of a running program.

I. Functions
1. Introduction It is often best to solve a large problem by successively decomposing it into smaller and smaller subproblems until the subproblems are easy enough to directly implement in C.

C facilitates this process by providing a mechanism for building up a large program from small subprograms called functions. A large complicated program can be constructed from combining a number of smaller programs (functions), each of which performs specific simple tasks.

To use a function we must invoke it with a function call. The function call specifies:
- the function name
- the arguments—i.e., the inputs—provided to the function

The syntax for a function call is:

\[ \text{function}_\text{name} \left( \text{argument}_1, \text{argument}_2, ..., \text{argument}_n \right) \]

Again, the arguments are the inputs to the function. Functions might (and often do) have only one argument. In fact, some functions have no arguments! If a function has arguments, the arguments can be numbers, variables or more complicated expressions.

The value a function computes is called the return value. The return value can be thought of as the output of a function. Functions can only have at most one return value. Many functions have no return value.

This probably all sounds a bit vague, so let's look at a concrete example. Suppose there was a function named \text{sqrt}, used to determine the square root of a number. In the statement

\[ y = \text{sqrt}( x ) \]

we have
- \( x \) as the function’s argument (input)
- \( \text{sqrt}( x ) \) is a function call

The value computed by the function is the function’s output, or return value. For example, if \( x \) has the value
9.0, then the function’s return value is 3.0, and this value is placed into the variable y.

Functions promote the writing of good programs. If we have to solve a large problem, we successively decompose the large problem into smaller sub-problems until the sub-problems are easy to directly implement as statements in the C programming language. Once we have finished dividing a large problem into individual sub-problems, we write small programs – called functions – to solve each of these individual subtasks.

2. User-Defined Functions

C has predefined functions which we can use, but we can also write our own functions. Functions we write ourselves are called *user-defined functions*. To use our own functions, we must write the code that permits the function to perform its required task.

A. The Function Definition

The function definition describes how the function accomplishes its task. A function definition is a small program. When we call the function, we run this small program. The syntax is:

```
data type of the result returned by the function

the data type of parameter_1

the data type of parameter_n

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

B. Arguments vs. Parameters

In a function call, the inputs to the function are called the arguments. These values of the arguments are plugged in for the parameters in the function definition before the body of the function is executed.

A parameter should be thought of as a placeholder that “stands in” for an argument. The person writing the function may not know the names chosen for the arguments, so he just picks his own parameter names that will serve to stand in for the arguments.

C. The return Statement

The **return** statement consists of the keyword **return** followed by an expression. The value of the expression is what is returned to the statement which called the function. In other words, the value of the expression after the **return** keyword is the function’s output. The function ends when the **return** statement executes.

3. Example

Suppose a person wants to write a function that calculates the absolute value of an integer. Utilizing the if-else statement which you learned in Lecture 3, the absolute value of an integer (given the name number in the program snippet below) can be calculated as:

```
if(number >= 0)
{
   AV = number;
}
else
{
   AV = -1 * number;
}
```
If we had a function named AbsVal( x ) which returns the absolute value of a given integer, x, we could write the program as:

```c
#include <stdio.h>
int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d" , &x );
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n" , y);
}
```

**Problem:** The program above will not work because there is no built-in function named AbsVal, so we’ll define one. The following program will work:

```c
#include <stdio.h>
int AbsVal( int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d" , &x );
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n" , y);
}
```

The program on the preceding page starts executing at the line of code that has the word `main` in it. The next
line declares the variables x and y, and since they are not initialized, they have random garbage values.

```c
#include <stdio.h>
int AbsVal( int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```

When the user is prompted to enter an integer, let’s say that he enters the value -5. This value is then placed in the variable x.

```c
#include <stdio.h>
int AbsVal( int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```

Now we reach the line with the function call ( y = AbsVal(x); ) . The program jumps to the first line of
the function named `AbsVal`.

```c
#include <stdio.h>
int AbsVal( int number) {
    int AV;
    if(number >=0) {
        AV = number;
    } else {
        AV = -1*number;
    }
    return AV;
}
int main() {
    int x, y;         x -5 y 23972
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```

And the value of the argument `x` is plugged into the parameter `number`.

```c
#include <stdio.h>
int AbsVal( int number) number -5
{
    int AV;
    if(number >=0) {
        AV = number;
    } else {
        AV = -1*number;
    }
    return AV;
}
int main() {
    int x, y;         x -5 y 23972
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```

Now the function declares its own variable named `AV` which initially has a garbage value, but is set equal to 5 in the else statement.
```c
#include <stdio.h>
int AbsVal(int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```

Now, when we reach the return statement (return AV;), we jump back to the original function call and the value of AV (which is 5) is placed in the variable y.

```
#include <stdio.h>
int AbsVal(int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```
4. **void functions**

Functions that produce no values for the rest of the program to use are called **void functions**. A common example: We want a function to send a message or some output to the screen. The output is sent to the screen, but is not sent back for use in the rest of the program.

The syntax for the function definition of a **void** function is then:

```c
void function_name(type_1 parameter_1,... , type_n parameter_n)
{
    body of the function
    return ;
}
```

The function call would be simply

```c
function_name(argument_1, ... , argument_n);
```

So, there are three key differences between the syntax of **void** functions and the syntax of other functions:

- keyword **void** is used instead of a return type.
- the **return** statement in the function definition does not contain any expression to be returned. In fact, the return statement can be entirely omitted.
- The function call is not used as the right side of an assignment statement.

We can rewrite our earlier example, using a **void** function, called **output**, to provide this output value. This is shown below. The function named **output** will be used to replace the **printf** statement that displays your absolute value.

```c
#include <stdio.h>

int AbsVal( int number)
{
    int AV;
    if(number >=0)
    {
        AV = number;
    }
    else
    {
        AV = -1*number;
    }
    return AV;
}

void output( int Abs_Num )
{
    printf("The absolute value of the integer is %d\n" , Abs_Num);
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d" , &x);
    y = AbsVal(x);
    output(y);
}
```

Note the new function named **output**.

The function named **output** is called here. The value of the argument **y** is plugged into the parameter **Abs_Num**.
5. The main function

Would you believe that you have been using functions all along! In fact, main is a function. It is a very special function in that all programs begin executing at the main function.

Practice Problem

Circle the appropriate words to complete the statements below:

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

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

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

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

Solution:

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

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

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

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

Practice Problem

What is the primary purpose of a function in a programming language (i.e., why are they used)?

Solution:

Practice Problem

Explain the error made during the call to the addthendisplay() function below.

```c
#include<stdio.h>
void addthendisplay( int first_num, int second_num )
{
    int sum_of_num = first_num + second_num;
    printf("The sum of the numbers is: %d\n\n", sum_of_num);
}
int main()
{
    int num1 = 27, num2 = 34, num3 = 13;
    addthendisplay( num1 , num2 , num3 );
}
```

Solution:
The way that functions are handled by the CPU is the last piece that we need before understanding an important attack called the buffer overflow attack. Would you believe that the way functions are handled by the CPU can, in the words of an infamous hacker, “produce some of the most insidious data-dependent bugs known to mankind.” Put another way, these functions intended to help us can open the door to allow your computer to be hijacked.

II. The Stack

Let’s think about what happens when a program is loaded into memory. Recall that the source code that we write is translated into machine language instructions, and these machine language instructions are fetched, decoded, and then executed, one-by-one.

So… you would surely agree that the program itself must reside in main memory. When the operating system executes a program, it allocates a block of memory for the machine language code that comprises the program. This section of memory is termed the text segment.

When the program is placed in the text segment, additional adjacent memory is given to the program to hold the values that it needs to successfully execute (e.g., values of variables). As we have mentioned, this section is called the stack.

Let’s look at an example of how the previous program would run in memory. For this first example, we will make some simplifying assumptions:

- We’ll look at source code instead of object code
- We’ll assume everything (an instruction, a character, an integer, etc.) consumes one address.

So, let’s suppose the program is loaded as shown below.
All programs begin execution at `main`. So, when execution begins, the `eip` register (the instruction pointer) holds the address of the next instruction to be executed: `0x080483c3`

This specific instruction is fetched, decoded and executed, and the `eip` register is then incremented so that it points to the next instruction. (Actually, this incrementing occurs after the fetch, but before the decode and execute.) As the program executes, this process (fetch, increment `eip`, decode the fetched instruction, execute) is repeated.

Early on in the program we declare variables `x` and `y`. Space has to be allotted for these variables. This is where the stack comes in! These variables are stored on the stack.

Also, since we are calling a function and passing a value, the argument for the function is stored in memory. Since, in this program, we are passing the value of the variable `x`, the compiler will copy the value of the variable `x` and store it in memory. In this program, the name of the copied value is `number`. 

```c
#include <stdio.h>
AbsVal( int number )
{
    int AV;
    if(number >= 0)
    {
        AV = number;
    } else
    {
        AV = -1*number;
    }
    return AV;
}

int main()
{
    int x, y;
    printf("Enter an integer: ");
    scanf("%d", &x);
    y = AbsVal(x);
    printf("The absolute value of the integer is %d\n", y);
}
```
So, now our picture looks like this:

Recall that two important registers—ebp and esp—are used to keep track of the location of the stack in memory.

- The stack pointer register, esp, points to the memory address at the top of the stack.
- The base register ebp points to the memory location at the bottom (the base) of the stack. Literally, the very next address after the bottom variable.

The program continues to run along, line by line, until 0x080483c8 is reached. At this point we are in trouble because we have to jump to an instruction that is not in order. The next instruction is at memory location 0x080483b5.

Consider the difficulty the CPU now encounters:

- We go to a different function (Abs_Val) that has its own variables
- We have to know where to jump back to when the function Abs_Val is done.
- We certainly don’t want to lose main’s variables when Abs_Val is done

So, here is what we do. Each function that is called gets its own section of the stack to work with, called its stack frame. So, for the stack frame for main, at this point in time, comprises addresses

\[ 0xbffff7d7 - 0xbffff7d9 \]

Now, we are going to give Abs_Val a stack frame for it to use for its own variables. After we are done executing the function Abs_Val, we will go back to the main function, and it will go back to using main's own stack frame.

Now, think about this, if we are going to give Abs_Val a stack frame to play with, what information will we need to restore the situation to the way it was before Abs_Val was called? To restore things to the way they were before, we need

- The proper return address for eip
- The prior value of the base pointer ebp
How should we “remember” these values? By placing these two items in the stack frame for the main function:

The stack

Now, we can safely jump to the function Abs_Val. This function has one variable, so the stack now looks like this:

The stack

What happens when we reach the return statement in the function Abs_Val? At that point we restore the stack frame for main. But… how do we do that?

Easy! We know where to reset esp (the stack pointer): In the picture above, we can reset esp to be ebp + 2.

We know what instruction address should be placed in eip: In the picture above, we place the value stored in address ebp + 1 into eip.

We know where to reset ebp (the base pointer): In the picture above, we reset ebp to the value pointed to be ebp. In this case, ebp points to the value of 0xbffff7da, so we reset ebp to point to address 0xbffff7da.

Practice Problem

Place the following elements in the order the will appear (from bottom to top) on the stack during a function call from main (while executing the instructions for that function).

- Return Address
- main’s Variables
- Function’s Variables
- Saved value of prior ebp
- Function’s Arguments

Solution:
Practice Problem

(a) Given the following source code and debugger output, construct the stack frame for the function `main` in the diagram below part b. Show where the base pointer (label as EBP-Main) and stack pointer (label as ESP-Main) are pointing to, and show where the arguments to `exam_function` are stored in memory.

```c
#include<stdio.h>
void exam_function( int x, int y, int z)
{
    int some_class;
    int best_class;
    int my_class;

    best_class = x;
    my_class = z;
    some_class = y;
}
int main()
{
    exam_function( 2005, 2003, 2015 );
}
```

(b) Using your answer from part a), and the additional debugger output below, construct the stack frame for the function `exam_function`. Show the location of the base pointer (label as EBP-Exam) and stack pointer (label as ESP-Exam) on the figure. Note on your figure:
- the location of `best_class`, `some_class`, and `my_class`
- the location of the return address
- the location of the prior value of the base pointer (EBP-Main)

```
(gdb) i r esp ebp eip
esp    0xbffff800 0xbffff800
ebp    0xbffff818 0xbffff818
eip    0x804836e 0x804836e <main+16>
(gdb) disassemble main
Dump of assembler code for function main:
0x0804835e <main+0>: push ebp
0x0804835f <main+1>: mov ebp,esp
0x08048361 <main+3>: sub esp,0x18
0x08048364 <main+6>: and esp,0xffffffff0
0x08048367 <main+9>: mov eax,0x0
0x0804836c <main+14>: sub esp,eax
0x0804836e <main+16>: mov DWORD PTR [esp+8],0x7df
0x08048376 <main+24>: mov DWORD PTR [esp+4],0x7d3
0x0804837e <main+32>: mov DWORD PTR [esp],0x7d5
0x08048385 <main+39>: call 0x8048344 <exam_function>
0x0804838a <main+44>: leave
0x0804838b <main+45>: ret
```
(gdb) i r esp ebp eip
esp  0xbffff7ec  0xbffff7ec
ebp  0xbffff7f8  0xbffff7f8
eip  0x804834a  0x804834a <exam_function+6>
(gdb) disassemble exam Function
Dump of assembler code for function exam Function:
0x08048344 <exam_function+0>:  push  ebp
0x08048345 <exam_function+1>:  mov  ebp,esp
0x08048347 <exam_function+3>:  sub  esp,0xc
0x0804834a <exam_function+6>:  mov  eax,DWORD PTR [ebp+8]
0x0804834d <exam_function+9>:  mov  DWORD PTR [ebp-8],eax
0x08048350 <exam_function+12>: mov  eax,DWORD PTR [ebp+16]
0x08048353 <exam_function+15>: mov  DWORD PTR [ebp-12],eax
0x08048356 <exam_function+18>: mov  eax,DWORD PTR [ebp+12]
0x08048359 <exam_function+21>: mov  DWORD PTR [ebp-4],eax
0x0804835c <exam_function+24>: leave
0x0804835d <exam_function+25>: ret
End of assembler dump.

Solution:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>BFFFF81C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFFFF820</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>