CS1100 - Lecture 16

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

Functions

In general, functions take some inputs, perform some operations with them and return the output. As a result of the operations performed, functions can also produce side effects. The inputs to a function are called arguments or parameters. The parameters of a function are its input and the return value is its output. Some of the commonly used functions used from the very beginning of this course are `printf()` and `scanf()`. The code for performing these operations was not included in our program. These are already defined in the header file `stdio.h`. So it has to be just included to our program using `#include<stdio.h>`. Every time we call these functions, we could give different inputs (parameters) to these functions, depending on the data that has to be printed to terminal or read from the terminal. A function written by ourself so far is the `main()` function. This function takes no arguments and returns an integer after performing some task.

Function Definition and Function Declaration

Let us first see the method of defining a function.

Let us consider writing a function to find the maximum of two integers. This function will have two parameters which are the integers that are to be compared. The return value of this function will be also an integer.

We will name this function as `max` and make its definition as follows.

```
int max(int a, int b)
/* this function returns the maximum of a and b */
{
    int x;
    if(a>b)
        x=a;
    else
        x=b;
    return (x);
}
```

The function definition has the entire C code that explains the sequence of operations to be performed on the input parameters to produce the output. This part is compulsory, if the function is user-defined i.e, not predefined in some library files. It is possible to define variables inside a function. The declaration `int x;` in the above function definition is an example of this. Such variables defined inside a function are called local variables.

The first line of a function definition is called the header of the function. In the above example, `int max(int a, int b)` is the function header. The parameters of the function which are used in the function header are called the formal parameters. In the example given, `a` and `b` are the formal parameters. The function header specifies the type of each parameter and also the type of the return value from the function. A function can have atmost one return value. It is a good coding practice to add a comment following the function header, explaining what are the parameters and what does the function do.

Now, we will see how to declare a function. It is advised that all user defined functions are declared in the beginning of the program. Declarations are made after statements like #include<stdio.h>. The function declaration specifies the name of the function, its return type and the type of its parameters. The declaration of the function `max()` can be done as follows:

```
int max(int,int);
```

Function Call

A *function call* or *function invocation* is the usage of the function. The usage of the function `max()` is done from inside the function `main()` in the following definition of `main()`.

```
int main()
{
    int i=10,j=20,m;
    m=max(i,j);
    printf("Maximum is %d",m);
}
```

The function `max()` is called using the instruction

```
m=max(i,j);
```

Here, `m` is an integer variable to which the return value is stored and `i` and `j` are the two integer variables whose maximum is to be computed using the `max()` function. The variables `i` and `j` which are passed as parameters while calling the `max()` function are called the *actual parameters*. Actual parameters and formal parameters need not have the same names. But it is necessary that the types of actual parameters in the function call match with the types of their corresponding formal parameters in the function definition. That is, the data type of the first parameter in the function call (variable `i` in the example) should be the same as the data type of the first parameter in the function definition (variable `a` in the example) and the data type of the second parameter in the function call (variable `j` in the example) should be the same as the data type of the second parameter in the function definition (variable `b` in the example) and so on. The data types of the return value from the function (variable `x` in the example) and the variable to which the return value is assigned using the function call instruction (variable `m` in the example) should also match.

Just before a function starts executing, a group of memory locations for storing the formal parameters and the local variables of the function getting called are created in memory. The group of locations created in this way is called the *activation record* or the *frame* of the function getting called.

Data flow and Control flow in the example of finding the maximum

Consider the previous example of finding the maximum of two numbers. When the `main()` function starts execution, the activation record or stack frame of the function `main()` gets created in memory stack. The memory for storing `i`, `j` and `m` gets allocated in this record. After taking the values of `i` and `j`, these values are stored to their corresponding locations. When the function `max()` is called, a new activation record gets created on top of that of the `main()` function. This record will allocate memory for storing all the formal parameters and local variables of function `max()` (i.e., for variables `x`, `a` and `b`). Now, the values of the actual parameters `i` and `j` are copied to the locations of formal parameters `a` and `b` respectively. Then the program control transfers to the first line of the function `max()`. During the execution of `max()`, values of `a` and `b` are compared and the maximum is placed in `x`.

While executing the `max()` function, the program does not have any access to the activation record of the `main()` function. During this time, the system has access to only the current activation record, which is the activation record of `max()`. Only manipulations on the local variables of the function `max()` are possible. After executing the `return` statement, the return value is copied to the left hand side of the instruction in the `main()` function where the function `max()` was called i.e, value of `x` is copied to `m`. Once the control returns from function `max()`, the activation record of `max()` is removed from the memory. Now the program control goes back to `main()` and continues from the line in `main()` after the function call `m=max(i,j);`. Then, the `printf()` statement is executed.

Data flow and Control flow in the general case

Initially when the program execution starts, the program control starts from the beginning of the `main()` function and proceeds. When the execution begins, an activation record for `main()` function is created in memory. This activation record will have memory locations for storing the variables defined in `main()`. Suppose the program control in a function `f1()` reaches a function call instruction, which calls a function `f2()`. Associated with the execution of this function call instruction, the following events occur.

- An activation record of the function getting called (`f2()`) is created and it is added to the top of the memory stack.
- The values of the actual parameters are copied from the activation record of the calling function `f1()` to the corresponding locations of formal parameters in the activation record of the called function `f2()`.
- From the line of the function call in `f1()`, the program control switches to the first line of the function `f2()` being called.
- The program control proceeds in the called function `f2()` until it reaches a `return` statement in `f2()` or the end of the function `f2()`.
- During this time, only the variables in the current activation record (activation record of `f2()`) can be accessed or modified.
- When the program control reaches a `return` statement or the end of the function `f2()`, the value of the expression given in the `return` instruction is copied to the location of the variable on the left side of the function call instruction in the calling function `f1()` or to a temporary location in memory, if the left side has no variables.

- After this, the program control returns to the instruction after the function call in the calling function `f1()`. Along with this, the activation record of `f2()` gets removed from the memory stack.

Example 2: Function for checking primality

In one of our earlier lectures we had discussed a program that checks whether a given number is prime or not. The program we discussed had no functions other than `main()`. Later, we also had some Exercise questions about printing all primes less than n etc. in which the primality checking code was again used. So it seems useful to define a separate function that checks if a given number is prime or not.

It is easy to see that this function should take one integer as its parameter. We will name the function as `isprime()` and define it in such a way that the function returns 1 if the parameter given is prime and returns 0 otherwise. The function `isprime()` is to print the first `n` prime numbers. One way of defining the function `isprime()` is given below.

```
int isprime(int k)
/*returns 1 if k is a prime number and returns 0 otherwise.*/
{
      int i, result=1;

      if(k<=1)
        result=0;
      else
      {
         for(i=2; i<=k/2; i++)
         {
            if (k%i == 0) //we found a non-trivial divisor of k
            {
               result=0;
               break;
            }
         }
      }
      return result;
}
```

In this function, the variable `k` is the only parameter and it will hold the value of the number to be checked for primality. The value of the variable `result` is initialized to `1`. We will design our function in such a way that the value of `result` remains the same if `k` is prime and the value of `result` changes to 0 otherwise. Finally, `result` is returned using the `return` statement.

Since no non-negative number less than 1 is prime, for `k` values less than or equal to 1, the `result` is made 0, for these values of the input parameter. For values of `k` greater than or equal to 2, it is checked in a loop if `k` has a non-trivial divisor `i` such that,$2 < i \leq k/2$. At any time, if any non-trivial divisor of `k` is found, then `result` is made 0 and it breaks from the loop. If no such divisors are found, the loop goes on executing without break and comes out of the loop when `i` becomes greater than or equal to `k/2`. In this case, the value of `result` remains as `1`. Finally, `result` is returned.

The declaration of this function should be done as follows.

```
int isprime(int);
```

This indicates that a function named `isprime()` takes an integer as input and returns an integer.

The `isprime()` function can be slightly modified to produce the same functionality, as given below.

```
int isprime(int k)
/*returns 1 if k is a prime number and returns 0 otherwise.*/
{
      int i;

      if(k<=1)
        return(0);
      else
      {
         for(i=2; i<=k/2; i++)
         {
            if (k%i == 0) //we found a non-trivial divisor of n
            {
               return(0);
            }
         }
       }
       return 1;
}
```

This function is similar to the previous one. The only difference is that instead of assigning the value to be returned to a variable (`result`) and returning it, this function directly returns that value. For `k` values less than or equal to 1, `0` is returned and for values of `k` greater than or equal to 2, it is checked if `k` is prime. If any non-trivial divisor of `k` is found, then `0` is returned (note that no `break` statement is necessary here). Otherwise `1` is returned.

Why do we need functions?

- Functions support code reuse. New libraries can be defined by someone and can be reused by others.
- After writing a function once, it can be called as many times as required. This helps to avoid repeating the same code again and again in a program. This way, functions help in reducing the length of the program.
- Modularity: Functions make the program look more simple and understandable. Functions help to break a big task into smaller and simpler sub tasks. For example,

consider a program to sort `n` numbers by finding the maximum among them and swapping it with the last element and repeating this for the rest the elements. The big task can be splitted to two smaller tasks using functions. One function can be used to find the position of the maximum element and a second function can be used to swap two elements.

- Each function has less number of variables that has to be dealt with at any particular instant. When a function is being executed, the programmer has to worry only about those variables declared within this function. The codes of a function do not affect another one, unless something is intentionally done.