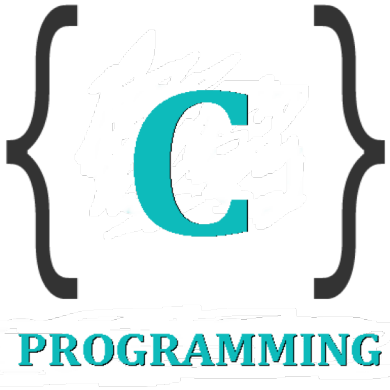

C Programming

Lecture - 15

- Every function in C has the following...
  - **Function Declaration (Function Prototype)**
  - **Function Definition**
  - **Function Call**

Function Declaration

- The function declaration tells the compiler about function name, datatype of the return value and parameters.
- The function declaration is also called as function prototype.
- The function declaration is performed before main function or inside main function or inside any other function.

- Function declaration syntax –
  - **returnType functionName(parametersList);**

- In the above syntax, **returnType** specifies the datatype of the value which is sent as a return value from the function definiton. The **functionName** is a user defined name used to identify the function uniquely in the program.
  The **parametersList** is the data values that are sent to the function definition.

Function Definition

- The function definition provides the actual code of that function. The function definition is also known as **body of the function**.
- The actual task of the function is implemented in the function definition. That means the actual instructions to be performed by a function are written in function definition.
- The actual instructions of a function are written inside the braces "{ }". The function definition is performed before main function or after main function.
- Function definition syntax –

```
returnType functionName(parametersList)
{
Actual code...
}
```

Function Call

- The function call tells the compiler when to execute the function definition. When a function call is executed, the execution control jumps to the function definition where the actual code gets executed and returns to the same functions call once the execution completes.

- The function call is performed inside main function or inside any other function or inside the function itself.

- Function call syntax –
  - **functionName(parameters);**

Example

- Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
}
```

```
/* function returning the max between two
numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Example

```
#include <stdio.h>
main(){
   int num1, num2, result ;
   int addition(int,int) ; // function declaration
   clrscr() ;
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);

   result = addition(num1, num2) ; // function call

   printf("SUM = %d", result);
}
int addition(int a, int b)  // function definition
{
   return a+b ;
}
```

In the example program, the function declaration statement **"int addition(int,int)"** tells the compiler that there is a function with name **addition** which takes two integer values as parameters and returns an integer value. The function call statement takes the execution control to the **additon()** definition along with values of **num1** and **num2**. Then **function definition** executes the code written inside it and comes back to the **function call** along with **return value**.

In the above example program, the function declaration statement **"int addition(int,int)"** tells the compiler that there is a function with name **addition**which takes two integer values as parameters and returns an integer value. The function call statement takes the execution control to the **additon()**definition along with values of **num1** and **num2**. Then **function definition** executes the code written inside it and comes back to the **function call** along with **return value**.

- Based on the data flow between the calling function and called function, the functions are classified as follows...

  - **Function without Parameters and without Return value**
  - **Function with Parameters and without Return value**
  - **Function without Parameters and with Return value**
  - **Function with Parameters and with Return value**

Function without Parameters and without Return value

- In this type of functions there is no data transfer between calling function and called function.
- Simply the execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function.

- For example, consider the program...

```
#include <stdio.h>
main()
{
        void addition() ; // function declaration
        addition() ; // function call
}
void addition()  // function definition
{
  int num1, num2 ;
  printf("Enter any two integer numbers : ") ;
  scanf("%d%d", &num1, &num2);
  printf("Sum = %d", num1+num2 ) ;
}
```

Function with Parameters and without Return value

- In this type of functions there is data transfer from calling function to called function (parameters) but there is no data transfer from called function to calling function (return value).
- The execution control jumps from calling function to called function along with the parameters and executes called function, and finally comes back to the calling function.
- For example, consider the program...

```
#include <stdio.h>
main()
{
        int num1, num2 ;
        void addition(int, int) ; // function declaration
        printf("Enter any two integer numbers : ") ;
        scanf("%d%d", &num1, &num2);
        addition(num1, num2) ; // function call
}
void addition(int a, int b)  // function definition
{
        printf("Sum = %d", a+b ) ;
}
```

Function without Parameters and with Return value

- In this type of functions there is no data transfer from calling function to called function (parameters) but there is data transfer from called function to calling function (return value).

- The execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function along with a return value.

- For example, consider the program...

```
#include <stdio.h>
#include<conio.h>
main(){
   int result ;
   int addition() ; // function declaration
   result = addition() ; // function call
   printf("Sum = %d", result) ;
}
int addition()  // function definition
{
   int num1, num2 ;
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);
   return (num1+num2) ;
}
```

Function with Parameters and with Return value

- In this type of functions there is data transfer from calling function to called function (parameters) and also from called function to calling function (return value).

- The execution control jumps from calling function to called function along with parameters and executes called function, and finally comes back to the calling function along with a return value.

- For example, consider the program...

```
#include <stdio.h>
main(){
   int num1, num2, result ;
   int addition(int, int) ; // function declaration
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);
   result = addition(num1, num2) ; // function call
   printf("Sum = %d", result) ;
}
int addition(int a, int b)  // function definition
{
   return (a+b) ;
}
```

Inter Function Communication in C

- When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate each other to exchange information. The process of exchanging information between calling and called functions is called as inter function communication.

- In C, the inter function communication is classified as follows...
  - **Downward Communication**
  - **Upward Communication**
  - **Bi-directional Communication**

Downward Communication

- In this type of inter function communication, the data is transferred from calling function to called function but not from called function to calling function.
- The **functions with parameters and without return value** are considered under downward communication.

- In the case of downward communication, the execution control jumps from calling function to called function along with parameters and executes the function definition, and finally comes back to the calling function without any return value.
- For example consider the program...

```
#include <stdio.h>
main(){
   int num1, num2 ;
   void addition(int, int) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;
   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
   addition(num1, num2) ; // calling function
}
void addition(int a, int b)  // called function
{
   printf("SUM = %d", a+b) ;
}
```

Upward Communication

- In this type of inter function communication, the data is transferred from called function to calling function but not from calling function to called function.
- The **functions without parameters and with return value** are considered under upward communication.

- In the case of upward communication, the execution control jumps from calling function to called function without parameters and executes the function definition, and finally comes back to the calling function along with a return value.
- For example consider the program...

```
#include <stdio.h>
main()
{
   int result ;
   int addition() ; // function declaration
   result = addition() ; // calling function
   printf("SUM = %d", result) ;
}
int addition()  // called function
{
   int num1, num2 ;
   num1 = 10;
   num2 = 20;
   return (num1+num2) ;
}
```

Bi - Directional Communication

- In this type of inter function communication, the data is transferred from calling function to called function and also from called function to calling function.
- The **functions with parameters and with return value** are considered under bi-directional communication.

- In the case of bi-directional communication, the execution control jumps from calling function to called function along with parameters and executes the function definition, and finally comes back to the calling function along with a return value.
- For example consider the following program...

```
#include <stdio.h>
main()
{
   int num1, num2, result ;
   int addition(int, int) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;
   result = addition(num1, num2) ; // calling function
   printf("SUM = %d", result) ;
}
int addition(int a, int b)  // called function
{
   return (a+b) ;
}
```

Standard Functions in C

- The standard functions are built-in functions. In C programming language, the standard functions are declared in header files and defined in .dll files. In simple words, the standard functions can be defined as "the ready made functions defined by the system to make coding more easy". The standard functions are also called as **library functions** or **pre-defined functions**.

- In C when we use standard functions, we must include the respective header file using **#include** statement. For example, the function **printf()** is defined in header file **stdio.h** (Standard Input Output header file). When we use **printf()** in our program, we must include **stdio.h** header file using **#include<stdio.h>**statement.

- C Programming Language provides the following header files with standard functions.

Header File	Purpose	Example Functions
stdio.h	Provides functions to perform standard I/O operations	printf(), scanf()
conio.h	Provides functions to perform console I/O operations	clrscr(), getch()
math.h	Provides functions to perform mathematical operations	sqrt(), pow()
string.h	Provides functions to handle string data values	strlen(), strcpy()
stdlib.h	Provides functions to perform general functions	calloc(), malloc()
time.h	Provides functions to perform operations on time and date	time(), localtime()
ctype.h	Provides functions to perform - testing and mapping of character data values	isalpha(), islower()
setjmp.h	Provides functions that are used in function calls	setjump(), longjump()
signal.h	Provides functions to handle signals during program execution	signal(), raise()

assert.h	Provides Macro that is used to verify assumptions made by the program	assert()
locale.h	Defines the location specific settings such as date formats and currency symbols	setlocale()
stdarg.h	Used to get the arguments in a function if the arguments are not specified by the function	va_start(), va_end(), va_arg()
errno.h	Provides macros to handle the system calls	Error, errno
float.h	Provides constants related to floating point data values	
limits.h	Defines the maximum and minimum values of various variable types like char, int and long	
stddef.h	Defines various variable types	
graphics.h	Provides functions to draw graphics.	circle(), rectangle()

Scope of Variable in C

- When we declare a variable in a program, it can not be accessed against the scope rules. Variables can be accessed based on their scope. Scope of a variable decides the portion of a program in which the variable can be accessed.

- Scope of the variable is defined as follows...
  - **Scope of a variable is the portion of the program where a defined variable can be accessed.**

- The variable scope defines the visibility of variable in the program. Scope of a variable depends on the position of variable declaration.

- In C programming language, a variable can be declared in three different positions and they are as follows...
  - **Before the function definition (Global Declaration)**
  - **Inside the function or block (Local Declaration)**
  - **In the function definition parameters (Formal Parameters)**

Before the function definition (Global Declaration)

- Declaring a variable before the function definition (outside the function definition) is called **global declaration**.
- The variable declared using global declaration is called **global variable**.

- The global variable can be accessed by all the functions that are defined after the global declaration.
- That means the global variable can be accessed any where in the program after its declaration. The global variable scope is said to be **file scope**.

```
#include <stdio.h>
int num1, num2 ; /*Global Variables */
void main(){
   void addition() ;
   void subtraction() ;
   void multiplication() ;
   num1 = 10 ;
   num2 = 20 ;
   printf("num1 = %d, num2 = %d", num1, num2) ;
   addition() ;
   subtraction() ;
   multiplication() ;
   getch() ;
}
```

```
void addition()
{
   int result ;
   result = num1 + num2 ;
   printf("\naddition = %d", result) ;
}
void subtraction()
{
   int result ;
   result = num1 - num2 ;
   printf("\nsubtraction = %d", result) ;
}
void multiplication()
{
   int result ;
   result = num1 * num2 ;
   printf("\nmultiplication = %d", result) ;
}
```

- In the above example program, the variables **num1** and **num2** are declared as global variables.

- They are declared before the main() function. So, they can be accessed by function main() and other functions that are defined after main().

- In the above example, the functions main(), addition(), subtraction() and multiplication() can access the variables num1 and num2.

Inside the function or block (Local Declaration)

- Declaring a variable inside the function or block is called **local declaration**.
- The variable declared using local declaration is called **local variable**.

- The local variable can be accessed only by the function or block in which it is declared.
- That means the local variable can be accessed only inside the function or block in which it is declared.

```
#include <stdio.h>
main(){
   void addition() ;
   int num1, num2 ;
   num1 = 10 ;
   num2 = 20 ;
   printf("num1 = %d, num2 = %d",
num1, num2) ;
   addition() ;
}
```

```
void addition()
{
   int sumResult ;
   sumResult = num1 + num2 ;
   printf("\naddition = %d", sumResult) ;
}
```

The above example program shows an **error** because, the variables num1 and num2 are declared inside the function main(). So, they can be used only inside main() function and not in addition() function.

In the function definition parameters (Formal Parameters)

- The variables declared in function definition as parameters have local variable scope.
- These variables behave like local variables in the function.
- They can be accessed inside the function but not outside the function.

```
#include <stdio.h>
main(){
   void addition(int, int) ;
   int num1, num2 ;
    num1 = 10 ;
   num2 = 20 ;
   addition(num1, num2) ;
}
void addition(int a, int b)
{
   int sumResult ;
   sumResult = a + b ;
   printf("\naddition = %d", sumResult) ;
}
```

In this example program, the variables a and b are declared in function definition as parameters. So, they can be used only inside the addition() function.

Parameter Passing in C

- When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function.
- When the execution control is transferred from calling function to called function it may carry one or more number of data values. These data values are called as **parameters**.
- **Parameters are the data values that are passed from calling function to called function.**

- In C, there are two types of parameters and they are as follows...
  - **Actual Parameters**
  - **Formal Parameters**

- The **actual parameters** are the parameters that are specified in calling function.

- The **formal parameters** are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

- In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...
  - **Call by Value**
  - **Call by Reference**

Call by Value

- In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function.

- **The changes made on the formal parameters does not effect the values of actual parameters**.

- That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

```
#include <stdio.h>
main()
{
  int num1, num2 ;
  void swap(int,int) ; // function declaration
  num1 = 10 ;
  num2 = 20 ;
  printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
  swap(num1, num2) ; // calling function
  printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}
```

```
void swap (int a, int b)  // called function
{
  int temp ;
  temp = a ;
  a = b ;
  b = temp ;
}
```

Identify Errors?

```
#include<stdio.h>
void message();
int main( )
{
        message() ;
        message();
        return 0;
}
    void message( );
    {
    printf ( "\nPraise worthy and C worthy are synonyms" ) ;
    }
```

Exercise...

Any year is entered through the keyboard. Write a function to determine whether the year is a leap year or not.

https://ideone.com/8cs7bq

Recursive Functions in C

- In C programming language, function calling can be made from main() function, other functions or from same function itself.

- The recursive function is defined as follows...
  - **A function called by itself is called recursive function.**

- The recursive functions should be used very carefully because, when a function called by itself it enters into infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

- When a function is called by itself, the first call remains under execution till the last call gets invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

```
#include <stdio.h>
int  factorial( int ) ;
main( )
{
        int  fact, n ;
        printf("Enter any positive
integer: ") ;
        scanf("%d", &n) ;
        fact = factorial( n ) ;
        printf("Factorial of %d is %d", n,
fact) ;
}
```

```
int  factorial( int  n )
{
        int  temp ;
        if( n == o)
            return  1 ;
        else
            temp = n * factorial( n-1
) ; // recursive function call
        return  temp ;
}
```

In the above example program, the **factorial()** function call is initiated from main() function with the value 3.

Inside the factorial() function, the function calls factorial(2), factorial(1) and factorial(0) are called recursively.

In this program execution process, the function call factorial(3) remains under execution till the execution of function calls factorial(2), factorial(1) and factorial(0) gets completed.

Similarly the function call factorial(2) remains under execution till the execution of function calls factorial(1) and factorial(0) gets completed.

In the same way the function call factorial(1) remains under execution till the execution of function call factorial(0) gets completed.

Output?

```
#include<stdio.h>
void display();
int main( )
{
        printf(“Learn C\n”)
        display();
        return 0;
}
    void display( );
    {
            printf (”Followed by C++ and Java!\n" ) ;
            main();
    }
```