MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

# BACHELOR OF COMPUTER APPLICATION (BCA)

**CSB-1112 (Problem Solving Using C)**

## Block 3: Advanced C Programming Concepts

## Unit 10: Functions

| **Structure** | |
|---|---|
| 10.0 | Introduction |
| 10.1 | Objectives |
| 10.2 | Definition of a Function |
| 10.3 | Declaration of a Function |
| 10.4 | Function Prototypes |
| 10.5 | Calling a Function |
| 10.6 | Function Arguments |
| 10.7 | The Return Statement |
| 10.8 | Types of Variables and Storage Classes |
| 10.9 | Types of Function Invoking |
| 10.10 | Recursion |
| 10.11 | Conclusion |
| 10.12 | Questions |
| 10.13 | References |

## 10.0 Introduction

To make programming simple and easy to debug, we break a larger program into smaller subprograms which perform 'well defined tasks'. These subprograms are called functions. So far we have defined a single function main ( ).

After reading this unit you will be able to define many other functions and the main( ) function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for Modular Programming, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an

interesting and important programming technique known as Recursion, in which a function calls within itself.

*This unit will explain to you the functions of language C.*

## 10.1 Objectives

After completing this unit, you will be able to:

- The need of functions in the programming;
- How to define and declare functions in 'c' language;
- Different types of functions and their purpose;
- How the functions are called from other functions;
- How data is transferred through parameter passing, to functions and the return statement; recursive functions; and
- The concept of 'call by value' and its drawbacks.
- Understand the concept and use pointers;
- Address and use of indirection operators;
- Make pointer type declaration, assignment and initialization;
- Use null pointer assignment;
- Use the pointer arithmetic;
- Handle pointers to functions;
- See the underlying unit of arrays and pointers; and
- Understand the concept of dynamic memory allocation

## 10.2 Definition of a Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions. A function can also be referred as a method or a sub-routine or a procedure, etc.

Functions are the C building blocks where every program activity occurs. It is a self contained program segment that carries out some specific, well-defined task. Every C program must have a function. One of the function must be main( ).

The general form of a function definition in C programming language is as follows:

return_type function_name( parameter list )

{ body of the function }

**C functions can be classified into two categories.**

- **Library functions:** Predefined in the standard library of C. Need is just to include the library.
- **User defined functions:** It has to be-developed by the user at the time of program writing.

***Need of user Defined Functions***

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. This approach clearly results in a number of advantages.

- Length of program can be reduced by using function.
- Reusability of function increases.
- It is easy to use.
- Debugging is more suitable (easier) for programs.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

**Example:**

```
/* 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;
}
```

## 10.3 Declaration of a Function

Before defining the function, it is appropriate to declare the function along with its prototype. In function prototype, the return value of function, type, and number of arguments are specified. The declaration of all functions statement should be first statements in main( ) or we can also declare globally for accessing all function within program. The general form of function declaration is

**<return_type> <function_name> ( [<argument_list>] );**

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

**return_type function_name( parameter list );**

For the above defined function max(),the function declaration is as follows:

**int max(int num1, int num2);**

Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration:

**int max(int , int);**

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## 10.4 Function Prototypes

Function prototypes are desirable because they facilitate error checking between calls to a function and corresponding function definition. They also help the compiler to perform automatic type conversions on function parameters. When a function is called, actual arguments are automatically converted to the types in function definition using normal rules of assignment.

The function Definition is, the task assigned to the function, that user declare. The general form of a function definition is:

**return_type function_name (declarations of formal argument list)**

```
{
    local variable declarations;
    executable statement 1;
    executable statement 2;
    ---------------------------
    ---------------------------
    ---------------------------
    executable statement n;
    return (expression);

}
```

Where return_type represents the data type of the value which is returned. The type specification can be omitted if the function returns an integer or a character. The formal argument list is a list of variables separated by commas that receive the values from main program when function is called.

The last statement in the body of function is return (expression). It is used to return the computed result, if any, to the calling program.

## 10.5 Calling a Function

A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas. If a function call does not require any arguments, an empty pair of parenthesis must follow the function name.

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

**Example:**

```
#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;
}
```

**The following result:**

```
Max value is: 200
```

## 10.6 Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function:

| Call Type | Description |
|---|---|
| **Call by value** | This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| **Call by reference** | This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

## 10.7 The Return Statement

Information is returned from the function to the calling portion of the program via return statement. Its uses control to be returned to the point from where the function was accessed. The return statement can take one of the following forms:

**return;**

**or**

**return (expression);**

In the return (expression); statement, the value of the expression is returned to the calling of the program.

A function can have multiple return statements, each containing different expression.

**Example:**

```
/* Program to convert lowercase character to uppercase */
# include
main( )
{
char lower, upper;
char Lower_to_upper (char Lower);
printf("\n Enter the lowercase character:");
scanf("%c", & Lower);
upper = Lower_to_upper (lower);
printf ("\n The upper case Equivalent is % c", upper);
}
char lower_to_upper (char ch);
{
char c2;
c2= (c1 > = 'a' && c1 < = 'z') ? (ch-32) : c1;
return (c2);
}
```

## 10.7.1 Call by Value

Call by value means sending the values of the arguments to functions. When a Single value is passed to a function via-an actual argument, the value of the actual argument is copied into the function in formal argument, and no matter what the function does with that value, the value stored in actual argument remains unchanged. This procedure to pass the value of an argument to a function is known as passing by value or call by value.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

**Example:**

```
/* function definition to swap the values */
#include<stdio.h>
void swap(int x, int y)
{
int temp;
temp = x;
/* save the value of x */
x = y;
/* put y into x */
y = temp;
/* put temp into y */
return;
}
```

Now, let us call the function swap() by passing actual values as in the following

**Example:**

```
#include<stdio.h>
/* function declaration */
void swap(int x, int y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values */
```

```
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

**The following Result:**

```
Before swap, value of a:    100
Before swap, value of b:    200
After swap, value of a:     100
After swap, value of b:     200
```

It shows that there are no changes in the values, though they had been changed inside the function.

## 10.7.2 Call by Reference

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly, you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
#include<stdio.h>
void swap(int *x, int *y)
{
int temp;
temp = *x;              /* save the value at address x */
*x = *y;                /* put y into x */
*y = temp;              /* put temp into y */
return;
}
```

Let us now call the function swap() by passing values by reference as in the following:

**Example:**

```
#include<stdio.h>       /* function declaration */
void swap(int *x, int *y);
int main ()
{
/* local variable definition */
```

```
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values.
* &a indicates pointer to a i.e. address of variable a and
* &b indicates pointer to b i.e. address of variable b. */
 swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

**The following Result:**

```
Before swap, value of a:     100
Before swap, value of b:     200
After swap, value of a:      200
After swap, value of b:      100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

## 10.8 Types of Variables and Storage Classes

In a program consisting of a number of functions a number of different types of variables can be found.

**Global vs. Static variables:** Global variables are recognized through out the program whereas local valuables are recognized only within the function where they are defined.

**Static vs. Dynamic variables:** Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their data type and by their storage class. One way to classify a variable is according to its data type and the other can be through its storage class. Data type refers to the type of value represented by a variable whereas storage class refers to the permanence of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

**Storage Classes**

There are four different storage classes specified in C:

1. Auto (matic)
2. Extern (al)

3. Static
4. Register

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

**A variable storage class tells us**

1. Where the variable would be stored.
2. What will be the initial value of the variable, if the initial value is not specifically assigned (i.e. the default initial value).
3. What is the scope of the variable, i.e., in which functions the value of the variable would be available.
4. What is the life of the variables; i.e., how long would the variable exist.

**Example:**

```
auto int a, b;
static int a, b;
extern float f;
```

## 10.8.1 Automatic Variables

These variables comes into existence whenever and wherever the variable is declared. These variables are also called as local variables, because these are available local to a function. The storage is in the memory and it has a default value, which is a garbage value. It is local to the block in which it has been declared and it life is till the control remains within the block in which the variable is defined. The key word used is 'auto'.

By default..a variable declared inside a function with storage class specification is an automatic.

**Declaration:**

```
int N;
or
auto int N;
```

**Example:**

```
main()
{
auto int i=10;
printf("%d",i);
}
```

**The following Result:**

```
10
```

## 10.8.2 External Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

**Points to remember:**

- These are global and can be accessed by any function within its scope.
- Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
- Usually lies outside or before the function accessing it.
- It allocates storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

**Declaration: extern int N;**

```
int i=10;          // global variable
main()
{
int i=2;
printf("%d",i);
display();
}
display();
{
printf("\n%d",i);
}
```

**The following Result:**

```
2
10
```

In the following example, Variable " i " is a global variable. If the global variable declared outside (before function definition), there is no need to use extern declaration in function that use global variable. Whereas, if global variable declared outside (after function definition), it has to be extern declaration within function that use global variable.

## 10.8.3 Static Variables

The storage is in the memory and default initial value is zero. It is local to the block in which it has been defined. The value of the variable persists between different function calls. The value will not disappear once the function in which it has been declared becomes inactive. It is unavailable only when you come out the program. The key word used is 'static'.

**Declaration:**

static int N;

**Example:**

```
void value()
{
static int a=5;
a=a+2;
printf("\t%d",a);
}
void main()
{ value();
value ();
value();
getch();
}
```

The output of the program is not 7 7 7

but it is 7 9 11

## 10.8.4 Register Variables

The storage of this type of variables is in the CPU registers. It has a garbage value initially. The scope of the variable is it is local to the block in which the variable is defined. Its life is till the control remains in the block in which it is defined. A value stored in a CPU register cal always be accessed faster than the one which is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register. A good example of frequently used variables is loop counters. The key word used is 'register'.

**Declaration:**

register int N;

**Example:**

```
main()
{
register int i=10;
printf(“%d”,i);
```

}

**Output:** 10

## 10.9 Types of Function Invoking

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

**The various types of invoking functions are:**

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value
- With arguments and with return value.

Let us discuss each category with some examples:

**TYPE-1: With no arguments and have no return value**

As the name suggests, any function which has no arguments and does not return any values to the calling function, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

**Example:**

```
/* Program for illustration of the function with no arguments and no return value*/
/* Function with no arguments and no return value*/
#include<stdio.h>
main()
{
void message();
printf("Control is in main\n");
message();                          /* Type 1 Function */
printf("Control is again in main\n");
}
void message()
{
printf("Control is in message function\n");
}                                   /* does not return anything */
```

**OUTPUT:**

```
Control is in main
Control is in message function
```

Control is again in main

**TYPE-2: With no arguments and with return value**

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

**Example:**

Write a program to find the sum of the first ten natural numbers.

```
/* Program to find sum of first ten natural numbers */
#include<stdio.h>
int cal_sum()
{
int i, s=0;
for (i=0; i<=10; i++) s=s + i;
return(s);                    /* function returning sum of first ten natural numbers */ }
main()
{
int sum;
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

**OUTPUT:**

Sum of first ten natural numbers is 55

**TYPE-3: With Arguments and have no return value**

If a function includes arguments but does not return anything, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the type of arguments or parameters here. There are two types of arguments:

- Actual arguments
- Formal arguments

Let us take an example to make this concept clear:

**Example:**

Write a program to calculate sum of any three given numbers.

```
#include<stdio.h>
main()
{
int a1, a2, a3;
```

```
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);                    /* Type 3 function */
}                                  /* function to calculate sum of three numbers */
void sum (int f1, int f2, int f3)
{
int s;
s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

**OUTPUT:**

Enter three numbers: 23 34 45 The sum of the three numbers is 102 Here f1, f2, f3 are formal arguments and a1, a2, a3 are actual arguments.

**TYPE-4: With arguments function and with return value**

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

**Example:**

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/
#include<stdio.h>
main( )
{
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);                    /* function call */
printf ("Sum of the given numbers is : %d\n", result); }   /* Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
{ return(f1+ f2 + f3);                      /* function returns a value */ }
```

**OUTPUT:**

Please enter any 3 numbers: 3 4 5

Sum of the given numbers is: 12

## 10.10 Recursion

Recursion is a process by which function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computation in which each action is stated in terms of previous result.

In order to solve a problem recursively, two conditions must be satisfied:

- Problem must be written in recursive form.
- Problem statement must include a stopping condition.

**Recursive Function Definition:**

Recursive function is a function that contains a call to itself. C supports creating recursive function with ease and efficient.

Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeatedly until the runtime stack overflows. Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily. This is also a well-known computer programming technique.

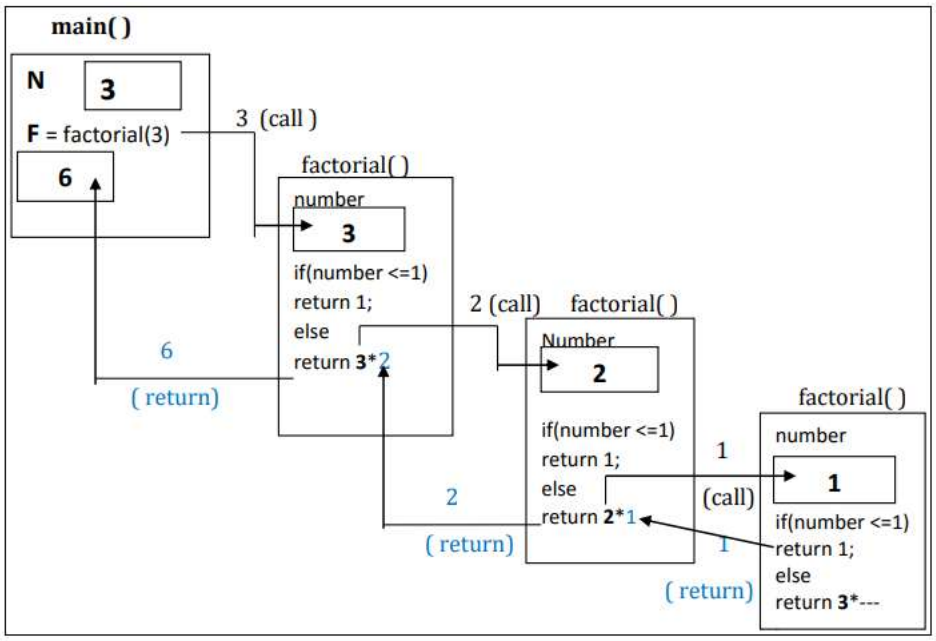
**Example 1:**

```
# include<stdio.h>
long unsigned int factorial( long unsigned int number)
{
if(number <= 1)
return 1;
else
return number * factorial(number - 1);
}
void main()
{
long unsigned int N,F;
clrscr( );
printf("Enter any Number for Calculate Factorial : ");
scanf("%lu",&N); F=factorial(N);
printf("\nFactorial of %lu is : %lu",N,F);
getch( );
}
```

**Output:**

```
Enter any Number for Calculate Factorial : 5
Factorial of 5 is : 120
```

**Process diagram of Program:**




**Example 2:**

/* display numbers from 1 to 10 using recursive function */

```
# include<stdio.h>
void display(int n)
{
 If (number > 10)
return;
else
printf("%d\n",n);
display(n+1);
}
void main()
{
clrscr( );
printf("Numbers from 1 to 10 \n");
display(1);                    //1 is a starting number
getch( );
}
```

## 10.11 Conclusion

In this unit, we learnt about “Functions”: definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as “void”, return type.

A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in ‘C’. Any variable declared in functions are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

## 10.12 Questions

1. **What is a function in C and why are they used?**

**Ans.1** A function in C is a block of code that performs a specific task. Functions are used to break down a large program into smaller, manageable pieces, making the code easier to understand, maintain, and reuse. They also help in reducing redundancy by allowing code reuse.

2. **Explain the difference between a function declaration and a function definition in C.**

**Ans.2** A function declaration (or prototype) specifies the function's name, return type, and parameters, but does not contain the actual body of the function. It informs the compiler about the function. A function definition, on the other hand, includes the actual implementation of the function. Example:

```
// Function declaration
int add(int, int);
// Function definition
int add(int a, int b) {
    return a + b;
}
```

3. **How do you pass arguments to functions in C? Explain with examples of pass-by-value and pass-by-reference.**

**Ans.3** In C, arguments can be passed to functions by value or by reference.

Pass-by-Value: A copy of the actual parameter is passed to the function. Modifications to the parameter within the function do not affect the original variable.

```
void increment(int a) {
    a = a + 1;
}
int main() {
    int x = 10;
    increment(x);
    printf("%d", x); // Output: 10
    return 0;
}
```

Pass-by-Reference: A pointer to the actual parameter is passed to the function. Modifications to the parameter within the function affect the original variable.

```
void increment(int *a) {
    *a = *a + 1;
}
int main() {
    int x = 10;
    increment(&x);
```

```
    printf("%d", x); // Output: 11
    return 0;
}
```

4. **What is a recursive function? Provide an example of a recursive function to calculate the factorial of a number.**

**Ans.4** A recursive function is a function that calls itself to solve a smaller instance of the same problem. Example of a recursive function to calculate the factorial of a number:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
int main() {
    int num = 5;
    printf("Factorial of %d is %d", num, factorial(num)); // Output: 120
    return 0;
}
```

5. **What are inline functions and how do they differ from regular functions?**

**Ans.5** Inline functions are functions that are expanded in line when invoked, meaning the function call is replaced with the function code itself. This can improve performance by avoiding function call overhead. Inline functions are defined using the inline keyword. They are typically used for small, frequently called functions.

```
inline int add(int a, int b) {
    return a + b;
}
int main() {
    int result = add(3, 4); // The call to add is replaced with its code
    printf("%d", result); // Output: 7
    return 0;
}
```

## 10.13 References


- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.


***