

UNIT- 5

FUNCTION IN C

Top-down and bottom-up approach in C

2

In this approach, a large project divides into small programs, and these programs are known as modules.

C programming language supports this approach for developing projects. It is always good idea that decomposing solution into modules in a hierarchal manner.

The basic task of a top-down approach is to divide the problem into tasks and then divide tasks into smaller sub-tasks and so on.

In this approach, first we develop the main module and then the next level modules are developed. This procedure is continued until all the modules are developed.

Advantages of top-down approach:

In this approach, first, we develop and test most important module.

This approach is easy to see the progress of the project by developer or customer.

Using this approach, we can utilize computer resources in a proper manner according to the project.

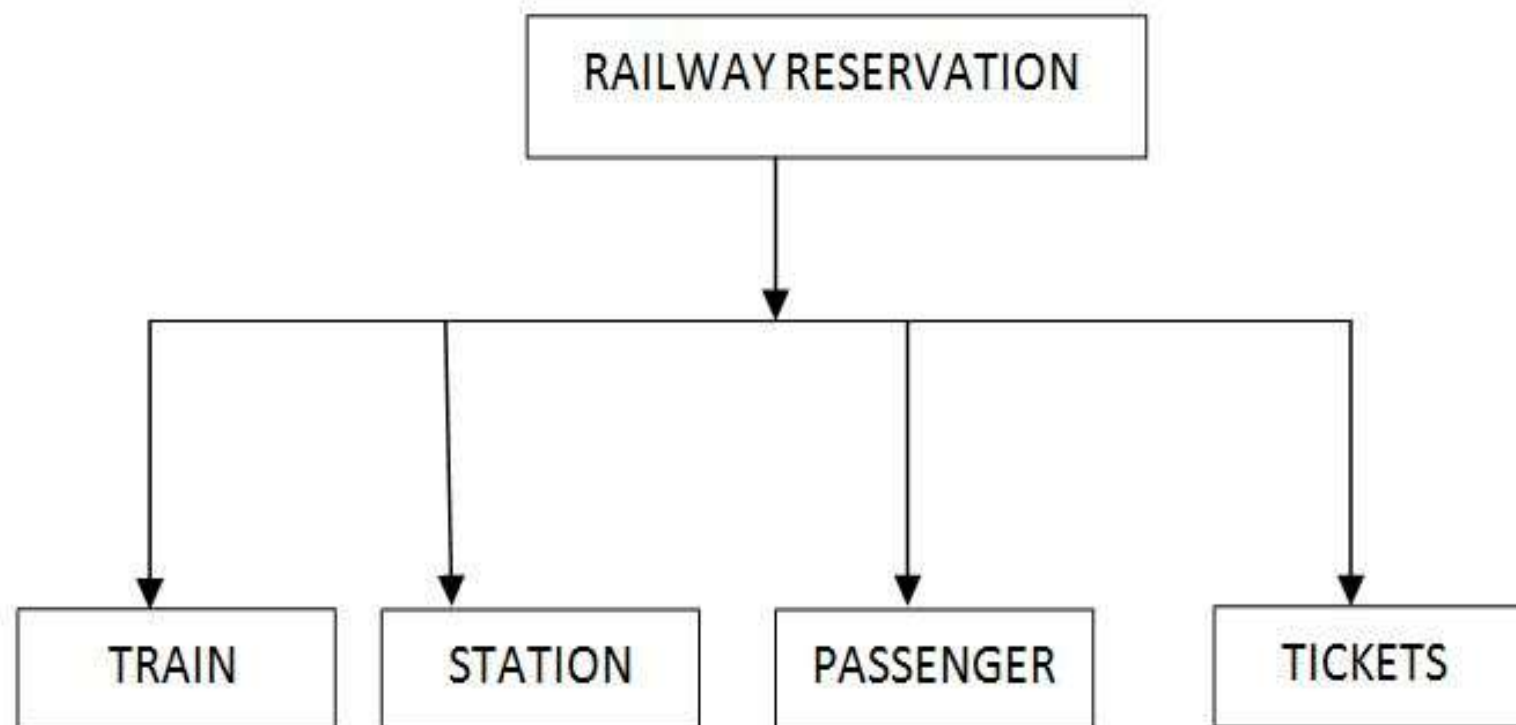
Testing and debugging is easier and efficient.

In this approach, project implementation is smoother and shorter.

This approach is good for detecting and correcting time delays.



TOP



BOTTOM

FUNCTION

- A function is a set of statements that take inputs, do some specific computation and produces output.
- The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.
- A function is a block of code that performs a specific task.
- Dividing a complex problem into smaller chunks makes our program easy to understand and reuse. Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:
 - create a circle function
 - create a color function
- 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.

Defining a Function

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

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Types of functions

1) Predefined standard library functions

Standard library functions are also known as built-in functions. Functions such as puts(), gets(), printf(), scanf() etc are standard library functions. These functions are already defined in header files (files with .h extensions are called header files such as stdio.h), so we just call them whenever there is a need to use them.

For example, printf() function is defined in <stdio.h> header file so in order to use the printf() function, we need to include the <stdio.h> header file in our program using #include <stdio.h>.

2) User Defined functions

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.



How user-defined function works?

7

```
#include <stdio.h>
void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..

    functionName();

    ... ..
    ... ..
}
```

The execution of a C program begins from the main() function.

When the compiler encounters functionName();, control of the program jumps to void functionName()

And, the compiler starts executing the codes inside functionName().

The control of the program jumps back to the main() function once code inside the function definition is executed.

Note, function names are identifiers and should be unique.

How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..
    functionName();
}

... ..
... ..
```

The diagram illustrates the flow of execution in a C program. It shows two function definitions: `void functionName()` and `int main()`. The `main` function calls `functionName()`. An arrow points from the call site in `main` to the start of `functionName`. Another arrow points from the end of `functionName` back to the point in `main` immediately following the call, indicating the return path.

Here is an example to add two integers. To perform this task, we have created an user-defined addNumbers().

```
#include <stdio.h>

int addNumbers(int a, int b);    // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);    // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)    // function definition
{
    int result;
    result = a+b;
    return result;              // return statement
}
```

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the given example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:

name of the function is `addNumbers()`

return type of the function is `int`

two arguments of type `int` are passed to the function

The function prototype is not needed if the user-defined function is defined before the `main()` function.



Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the given example, the function call is made using `addNumbers(n1, n2);` statement inside the `main()` function.

Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)  
{  
    //body of the function  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the given example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

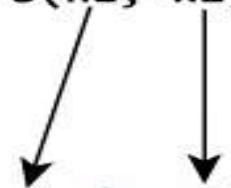
How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    ... ..
}
```

A diagram with two arrows. One arrow starts from the variable 'n1' in the function call 'addNumbers(n1, n2)' and points down to the parameter 'a' in the function definition 'int addNumbers(int a, int b)'. The other arrow starts from the variable 'n2' in the function call and points down to the parameter 'b' in the function definition.

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.

A function can also be called without passing an argument.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function. The sum variable in the main() function is assigned this value.

Syntax of return statement

`return (expression);`

For example,

`return a;`

`return (a+b);`

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```

sum = result

Types of User-defined Functions in C Programming

In the concept of functions, the function call is known as "Calling Function" and the function definition is known as "Called Function".

When we make a function call, the execution control jumps from calling function to called function. After executing the called function, the execution control comes back to calling function from called function. When the control jumps from calling function to called function it may carry one or more data values called "**Parameters**" and while coming back it may carry a single value called "**return value**". That means the data values transferred from calling function to called function are called as **Parameters** and the data value transferred from called function to calling function is called **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 following program...

```
#include<stdio.h>
#include<conio.h>
void main(){
    void addition() ;// function declaration
    clrscr() ;

    addition() ;// function call

    getch() ;
}
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 following program...

```
#include<stdio.h>
#include<conio.h>

void main(){
    int num1, num2 ;
    void addition(int, int) ; // function declaration
    clrscr() ;
    printf("Enter any two integer numbers : ") ;
    scanf("%d%d", &num1, &num2);

    addition(num1, num2) ; // function call
    getch() ;
}

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 following program...

```
#include<stdio.h>
#include<conio.h>

void main(){
    int result ;
    int addition() ;// function declaration
    clrscr() ;

    result = addition() ;// function call
    printf("Sum = %d", result) ;
    getch() ;
}

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 following program...

```
#include<stdio.h>
#include<conio.h>

void 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) ;
    getch() ;
}

int addition(int a, int b) // function definition
{
    return (a+b) ;
}
```

Points to be Remembered

- The parameters specified in calling function are said to be **Actual Parameters**.
- The parameters declared in called function are said to be **Formal Parameters**.
- The value of actual parameters is always copied into formal parameters.

Parameter Passing in C (Passing Arguments to a Function)

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 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 next program...



```
#include<stdio.h>
#include<conio.h>

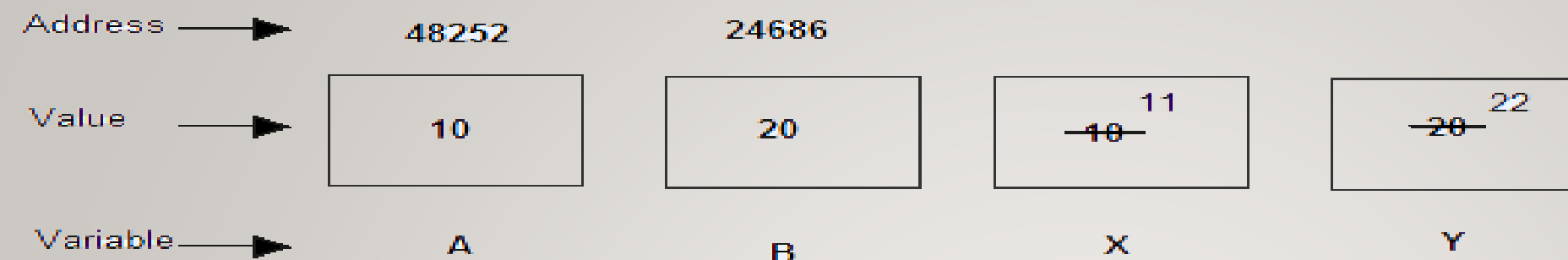
void main(){
    int num1, num2 ;
    void swap(int,int) ;// function declaration
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;

    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;

    swap(num1, num2) ;// calling function

    printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
    getch() ;
}

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



"C:\Users\User\Desktop\New folder\parameter_passing\bin\Debug\parameter_passing.exe"

Before swap: num1 = 10, num2 = 20

After swap: num1 = 10

num2 = 20

Process returned 0 (0x0) execution time : 0.047 s

Press any key to continue.

In the given example program, the variables **num1** and **num2** are called actual parameters and the variables **a** and **b** are called formal parameters. The value of **num1** is copied into **a** and the value of **num2** is copied into **b**. The changes made on variables **a** and **b** does not effect the values of **num1** and **num2**.

Call by Reference

In **Call by Reference** parameter passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function. In this method of parameter passing, the formal parameters must be **pointer** variables.

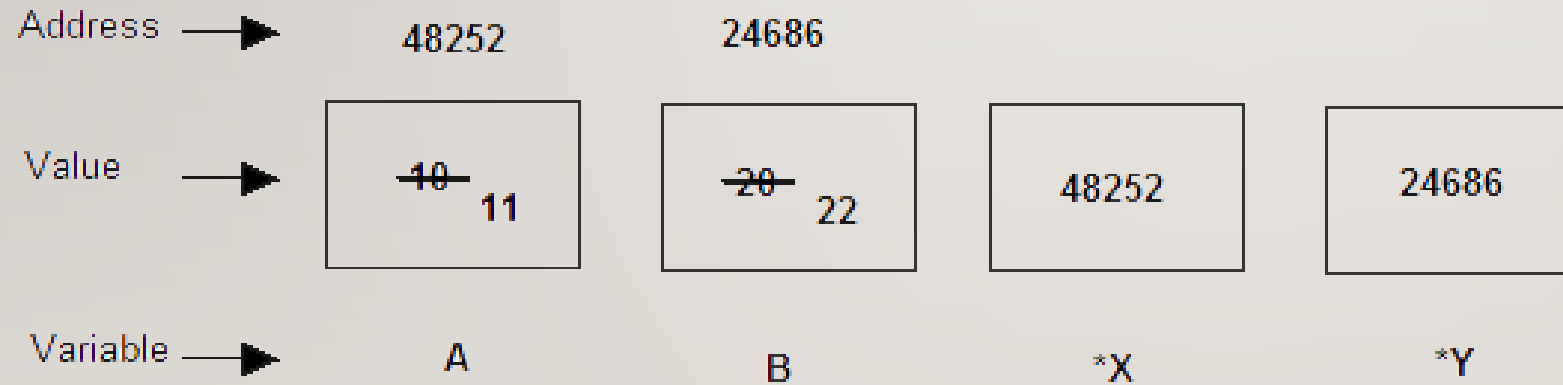
That means in call by reference parameter passing method, the address of the actual parameters is passed to the called function and is received by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So **the changes made on the formal parameters effects the values of actual parameters**. For example consider the given program...

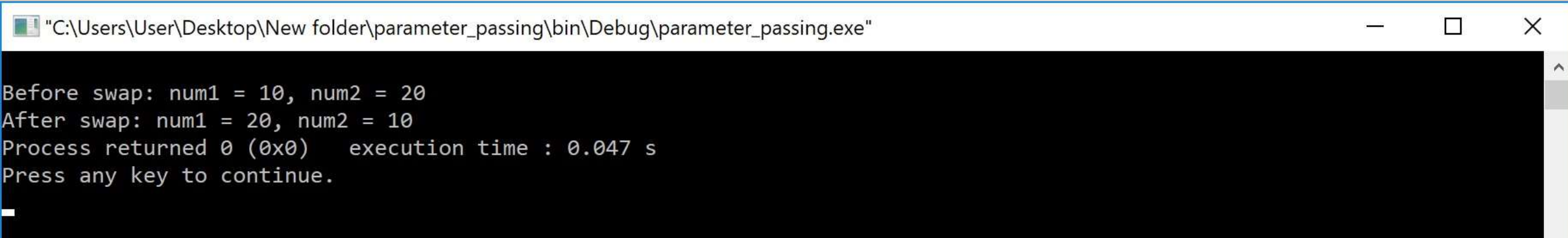
```
#include<stdio.h>
#include<conio.h>

void main(){
    int num1, num2 ;
    void swap(int *,int *) ; // function declaration
    clrscr() ;
    num1 = 10 ;
    num2 = 20 ;

    printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
    swap(&num1, &num2) ; // calling function
```

```
printf("\nAfter swap: num1 = %d, num2 = %d", num1, num2);  
    getch() ;  
}  
void swap(int *a, int *b) // called function  
{  
    int temp ;  
    temp = *a ;  
    *a = *b ;  
    *b = temp ;  
}
```





```
"C:\Users\User\Desktop\New folder\parameter_passing\bin\Debug\parameter_passing.exe"

Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.
_
```

In the given example program, the addresses of variables **num1** and **num2** are copied to pointer variables **a** and **b**. The changes made on the pointer variables **a** and **b** in called function effects the values of actual parameters **num1** and **num2** in calling function.

Difference between Call by Value and Call by Reference.

Call by Value	Call by Reference
The actual arguments can be variable or constant.	The actual arguments can only be variable.
The values of actual argument are sent to formal argument which are normal variables.	The reference of actual argument are sent to formal argument which are pointer variables.
Any changes made by formal arguments will not reflect to actual arguments.	Any changes made by formal arguments will reflect to actual arguments.

Recursive Function in C

27

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {  
    recursion(); /* function calls itself */  
}  
  
int main() {  
    recursion();  
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.



Number Factorial

The following example calculates the factorial of a given number using a recursive function –

```
#include <stdio.h>
```

```
unsigned long long int factorial(unsigned int i) {
```

```
    if(i <= 1) {
```

```
        return 1;
```

```
    }
```

```
    return i * factorial(i - 1);
```

```
}
```

```
int main() {
```

```
    int i = 12;
```

```
    printf("Factorial of %d is %d\n", i, factorial(i));
```

```
    return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result –

Factorial of 12 is 479001600



Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function –

```
#include<stdio.h>
void fibonacciSeries(int range)
{
    int a=0, b=1, c;
    for(i=1; i<=range; i++)
    {
        printf("%d\t", a);
        c = a+b;
        a = b;
        b = c;
    }
}
int main()
{
    int range;
    printf("Enter range: ");
    scanf("%d", &range);
    printf("The fibonacci series is: \n");
    fibonacciSeries(range);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

0
1
1
2
3
5
8
13
21
34

Passing Arrays as Function Arguments in C

3 |

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Way-I

Formal parameters as a pointer –

```
void myFunction(int *param) {  
    .  
    .  
    .  
}
```



Way-2

Formal parameters as a sized array –

```
void myFunction(int param[10]) {  
    .  
    .  
    .  
}
```

Way-3

Formal parameters as an unsized array –

```
void myFunction(int param[]) {  
    .  
    .  
    .  
}
```

Example

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows –



```
double getAverage(int arr[], int size) {
```

```
    int i;
    double avg;
    double sum = 0;
```

```
    for (i = 0; i < size; ++i) {
        sum += arr[i];
    }
```

```
    avg = sum / size;
```

```
    return avg;
}
```

Now, let us call the above function as follows –

```
#include <stdio.h>
```

```
/* function declaration */
double getAverage(int arr[], int size);
```

```
int main () {
```

```
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;
```

```
    /* pass pointer to the array as an argument */
    avg = getAverage( balance, 5 ) ;
```

```
    /* output the returned value */
    printf( "Average value is: %f ", avg );
```

```
    return 0;
}
```

When the above code is compiled together and executed, it produces the following result –

Average value is: 214.400000

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.



UNIT – 5

COMPLETED