There are two types of **functions in C programming**: Library **Functions**: are the **functions** which are declared in the **C** header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc. User-defined **functions**: are the **functions** which are created by the **C programmer**

Global variables are declared outside any function, and they can be accessed (used) on any function in the program. **Local variables** are declared inside a function, and can be used only inside that function. It is possible to have **local variables** with the same name in different functions. A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```c
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

```c
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

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

```
value of g = 10
```

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example −

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {

   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

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

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

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.

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.

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;
}
```

Function Declarations

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.

Calling a Function

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. For example −

```c
#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;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

```
Max value is : 200
```

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 −

Sr.No.	Call Type & Description
1	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.
2	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.

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.

All C functions can be called either with arguments or without arguments in a **C** program. Also, they may or may not **return** any values. Hence the **function** prototype of a **function** in **C** is as below: ... Similarly when it does not **return** a value, the calling **function** does not receive any data from the called **function**.

Functions in C Programming with examples

BY CHAITANYA SINGH | FILED UNDER: C-PROGRAMMING

A function is a block of statements that performs a specific task. Suppose you are building an application in C language and in one of your program, you need to perform a same task more than once. In such case you have two options –

a) Use the same set of statements every time you want to perform the task
b) Create a function to perform that task, and just call it every time you need to perform that task.

Using option (b) is a good practice and a good programmer always uses functions while writing codes in C.

Types of functions

1) **Predefined standard library functions** – such as `puts()`, `gets()`, `printf()`, `scanf()` etc – These are the functions which already have a definition in header files (.h files like `stdio.h`), so we just call them whenever there is a need to use them.

2) **User Defined functions –** The functions that we create in a program are known as user defined functions.

In this guide, we will learn how to create user defined functions and how to use them in C Programming

Why we need functions in C

Functions are used because of following reasons –
a) To improve the readability of code.
b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Syntax of a function

```
return_type function_name (argument list)
{
    Set of statements – Block of code
}
```

return_type: Return type can be of any data type such as int, double, char, void, short etc. Don't worry you will understand these terms better once you go through the examples below.

function_name: It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.

argument list: Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

Block of code: Set of C statements, which will be executed whenever a call will be made to the function.

Do you find above terms confusing? – Do not worry I'm not gonna end this guide until you learn all of them :)
Lets take an example – Suppose you want to create a function to add two integer variables.

Let's split the problem so that it would be easy to understand –
Function will add the two numbers so it should have some meaningful name like sum, addition, etc. For example lets take the name addition for this function.

```
return_type addition(argument list)
```

This function addition adds two integer variables, which means I need two integer variable as input, lets provide two integer parameters in the function signature. The function signature would be –

```
return_type addition(int num1, int num2)
```

The result of the sum of two integers would be integer only. Hence function should return an integer value – **I got my return type** – It would be integer –

```
int addition(int num1, int num2);
```

So you got your function prototype or signature. Now you can implement the logic in C program like this:

How to call a function in C?

Consider the following C program

Example1: Creating a user defined function addition()

```
#include <stdio.h>
int addition(int num1, int num2)
{
     int sum;
     /* Arguments are used here*/
     sum = num1+num2;

     /* Function return type is integer so we are returning
      * an integer value, the sum of the passed numbers.
      */
     return sum;
}

int main()
{
     int var1, var2;
     printf("Enter number 1: ");
     scanf("%d",&var1);
     printf("Enter number 2: ");
     scanf("%d",&var2);

     /* Calling the function here, the function return type
      * is integer so we need an integer variable to hold the
      * returned value of this function.
      */
     int res = addition(var1, var2);
     printf ("Output: %d", res);

     return 0;
}
```

Output:

```
Enter number 1: 100
Enter number 2: 120
Output: 220
```

Example2: Creating a void user defined function that doesn’t return anything

```
#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
    printf("Hi\n");
    printf("My name is Chaitanya\n");
    printf("How are you?");
    /* There is no return statement inside this function, since its
     * return type is void
     */
}

int main()
{
     /*calling function*/
     introduction();
     return 0;
}
```

Output:

```
Hi
My name is Chaitanya
How are you?
```

Few Points to Note regarding functions in C:

1) `main()` in C program is also a function.
2) Each C program must have at least one function, which is main().
3) There is no limit on number of functions; A C program can have any number of functions.
4) A function can call itself and it is known as "**Recursion**". I have written a separate guide for it.

C Functions Terminologies that you must remember

return type: Data type of returned value. It can be void also, in such case function doesn't return any value.

Note: for example, if function return type is **char,** then function should return a value of char type and while calling this function the main() function should have a variable of char data type to store the returned value.

Structure would look like –

```
char abc(char ch1, char ch2)
{
   char ch3;
   …
   …
   return ch3;
}

int main()
{
   …
   char c1 = abc('a', 'x');
   …
}
```

More Topics on Functions in C

1) **Function – Call by value method** – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

2) **Function – Call by reference method** – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

*Function call by value in C programming

BY CHAITANYA SINGH | FILED UNDER: C-PROGRAMMING

Function call by value is the default way of calling a function in C programming. Before we discuss function call by value, lets understand the terminologies that we will use while explaining this:

Actual parameters: The parameters that appear in function calls.
Formal parameters: The parameters that appear in function declarations.

For example:

```
#include <stdio.h>
int sum(int a, int b)
{
     int c=a+b;
     return c;
}

int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);

    return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

In this guide, we will discuss **function call by value**. If you want to read call by reference method then refer this guide: function call by reference.

Lets get back to the point.
What is Function Call By value?
When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied to the formal parameters. Thus operations performed on the formal parameters don’t reflect in the actual parameters.

Example of Function call by Value

As mentioned above, in the call by value the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn’t affect actual parameters. Lets take an example to understand this:

```
#include <stdio.h>
int increment(int var)
{
    var = var+1;
    return var;
}

int main()
{
   int num1=20;
   int num2 = increment(num1);
   printf("num1 value is: %d", num1);
   printf("\nnum2 value is: %d", num2);

   return 0;
}
```

Output:

```
num1 value is: 20
num2 value is: 21
```

Explanation
We passed the variable num1 while calling the method, but since we are calling the function using call by value method, only the value of num1 is copied to the formal parameter var. Thus change made to the var doesn’t reflect in the num1.

Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
   int tempnum ;
   /*Copying var1 value into temporary variable */
   tempnum = var1 ;

   /* Copying var2 value into var1*/
   var1 = var2 ;

   /*Copying temporary variable value into var2 */
   var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output:

```
Before swapping: 35, 45
After swapping: 35, 45
```

Why variables remain unchanged even after the swap?
The reason is same – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

*Function call by reference in C Programming

BY CHAITANYA SINGH | FILED UNDER: C-PROGRAMMING

Before we discuss function call by reference, lets understand the terminologies that we will use while explaining this:
Actual parameters: The parameters that appear in function calls.
Formal parameters: The parameters that appear in function declarations.
For example: We have a function declaration like this:

```
int sum(int a, int b);
```

The a and b parameters are formal parameters.

We are calling the function like this:

```
int s = sum(10, 20); //Here 10 and 20 are actual parameters
or
int s = sum(n1, n2); //Here n1 and n2 are actual parameters
```

In this guide, we will discuss function call by reference method. If you want to read call by value method then refer this guide: function call by value.

Lets get back to the point.

What is Function Call By Reference?
When we call a function by passing the addresses of actual parameters then this way of calling the function is known as call by reference. In call by reference, the operation performed on formal parameters, affects the value of actual parameters because all the operations performed on the value stored in the address of actual parameters. It may sound confusing first but the following example would clear your doubts.

Example of Function call by Reference

Lets take a simple example. Read the comments in the following program.

```
#include <stdio.h>
void increment(int  *var)
{
    /* Although we are performing the increment on variable
     * var, however the var is a pointer that holds the address
     * of variable num, which means the increment is actually done
     * on the address where value of num is stored.
     */
    *var = *var+1;
}
int main()
{
     int num=20;
     /* This way of calling the function is known as call by
      * reference. Instead of passing the variable num, we are
      * passing the address of variable num
      */
     increment(&num);
     printf("Value of num is: %d", num);
   return 0;
}
```

Output:

```
Value of num is: 21
```

Example 2: Function Call by Reference - Swapping numbers

Here we are swapping the numbers using call by reference. As you can see the values of the variables have been changed after calling the swapnum() function because the swap happened on the addresses of the variables num1 and num2.

```
#include
void swapnum ( int *var1, int *var2 )
{
   int tempnum ;
   tempnum = *var1 ;
   *var1 = *var2 ;
   *var2 = tempnum ;
}
int main( )
{
   int num1 = 35, num2 = 45 ;
   printf("Before swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);

   /*calling swap function*/
   swapnum( &num1, &num2 );

   printf("\nAfter swapping:");
   printf("\nnum1 value is %d", num1);
   printf("\nnum2 value is %d", num2);
   return 0;
}
```

Output:

```
Before swapping:
num1 value is 35
num2 value is 45
After swapping:
num1 value is 45
num2 value is 35
```

Passing array to function in C programming with example

BY CHAITANYA SINGH | FILED UNDER: C-PROGRAMMING

Just like variables, array can also be passed to a function as an argument . In this guide, we will learn how to pass the array to a function using call by value and call by reference methods.

To understand this guide, you should have the knowledge of following C Programming topics:

1. C – Array
2. Function call by value in C
3. Function call by reference in C

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include <stdio.h>
void disp( char ch)
{
   printf("%c ", ch);
}
int main()
{
   char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
   for (int x=0; x<10; x++)
   {
       /* I’m passing each element one by one using subscript*/
       disp (arr[x]);
   }

   return 0;
}
```

Output:

```
a b c d e f g h i j
```

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

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

```
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
     for (int i=0; i<10; i++)
     {
         /* Passing addresses of array elements*/
         disp (&arr[i]);
     }

     return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 0
```

How to pass an entire array to a function as an argument?

In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:

Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that **arr** is equivalent to the **&arr[0]**.

```
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
        /* The pointer var1 is pointing to the first element of
         * the array and the var2 is the size of the array. In the
         * loop we are incrementing pointer so that it points to
         * the next element of the array on each increment.
         *
         */
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}

int main()
{
     int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
     myfuncn(var_arr, 7);
    return 0;
}
```
Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
*
```
What is the use of recursion in C?

C - **Recursion**. **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. The **C** programming language supports **recursion**, i.e., a function to call itself.

***6 Different Types of Recursion in C Explained with Programming Example**

- Primitive **Recursion**.
- Tail **Recursion**.
- Single **Recursion**.
- Multiple **Recursion**.
- Mutual **Recursion** or Indirect **Recursion**)
- General **Recursion**.

Types of Recursion

-
-

SUMMARY TYPES OF RECURSION

There are many ways to categorize a recursive function. Listed below are some of the most common.

Linear Recursive

A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution). The factorial function is a good example of linear recursion.

Another example of a linear recursive function would be one to compute the square root of a number using Newton's method (assume `EPSILON` to be a very small number close to 0):

```
double my_sqrt(double x, double a)
{
        double difference = a*x-x;
        if (difference < 0.0) difference = -difference;
        if (difference < EPSILON) return(a);
        else return(my_sqrt(x,(a+x/a)/2.0));
}
```

Tail recursive

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

```
int gcd(int m, int n)
{
        int r;

        if (m < n) return gcd(n,m);

        r = m%n;
        if (r == 0) return(n);
        else return(gcd(n,r));
}
```

Binary Recursive

Some recursive functions don't just have one call to themself, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

The mathematical combinations operation is a good example of a function that can quickly be implemented as a binary recursive function. The number of combinations, often represented as *nCk* where we are choosing n elements out of a set of k elements, can be implemented as follows:

```
int choose(int n, int k)
{
        if (k == 0 || n == k) return(1);
        else return(choose(n-1,k) + choose(n-1,k-1));
}
```

Exponential recursion

An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set (exponential meaning if there were *n* elements, there would be $O(a^n)$ function calls where a is a positive number).

A good example an exponentially recursive function is a function to compute all the permutations of a data set. Let's write a function to take an array of `n` integers and print out every permutation of it.

```
void print_array(int arr[], int n)
{
        int i;
        for(i=0; i<n; i) printf("%d ", arr[i]);
        printf("\n");
}

void print_permutations(int arr[], int n, int i)
{
        int j, swap;
        print_array(arr, n);
        for(j=i+1; j<n; j) {
                swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
                print_permutations(arr, n, i+1);
                swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
        }
}
```

To run this function on an array `arr` of length `n`, we'd do `print_permutations(arr, n, 0)` where the 0 tells it to start at the beginning of the array.

Nested Recursion

In nested recursion, one of the arguments to the recursive function is the recursive function itself! These functions tend to grow extremely fast. A good example is the classic mathematical function, "Ackerman's function. It grows very quickly (even for small values of x and y, Ackermann(x,y) is extremely large) and it cannot be computed with only definite iteration (a completely defined `for()` loop for example); it requires indefinite iteration (recursion, for example).

```
Ackerman's function
int ackerman(int m, int n)
{
        if (m == 0) return(n+1);
        else if (n == 0) return(ackerman(m-1,1));
        else return(ackerman(m-1,ackerman(m,n-1)));
}
```

Try computing ackerman(4,2) by hand... have fun!

Mutual Recursion

A recursive function doesn't necessarily need to call itself. Some recursive functions work in pairs or even larger groups. For example, function A calls function B which calls function C which in turn calls function A.

A simple example of mutual recursion is a set of function to determine whether an integer is even or odd. How do we know if a number is even? Well, we know 0 is even. And we also know that if a number *n* is even, then *n* - 1 must be odd. How do we know if a number is odd? It's not even!

```
int is_even(unsigned int n)
{
        if (n==0) return 1;
        else return(is_odd(n-1));
}

int is_odd(unsigned int n)
{
        return (!iseven(n));
}
```

I told you recursion was powerful! Of course, this is just an illustration. The above situation isn't the best example of when we'd want to use recursion instead of iteration or a closed form solution. A more efficient set of function to determine whether an integer is even or odd would be the following:

```
int is_even(unsigned int n)
{
        if (n % 2 == 0) return 1;
        else return 0;
}

int is_odd(unsigned int n)
{
        if (n % 2 != 0) return 1;
        else return 0;
}
```

There are two ways to pass parameters in C: Pass by Value, Pass by Reference.

1. **Pass** by Value. **Pass** by Value, means that a copy of the data is made and stored by way of the name of the **parameter**. ...
2. **Pass** by Reference. A reference **parameter** "refers" to the original data in the calling **function**.

A function is a block of codes that performs a specific task and may return value. The main() function is the first user defined function invoked by the compiler. While it is possible to write any code within main function, it leads number of problems. The program may become too large and complex and it is difficult to test, debugg and maintain the complex code. For that reason, We use function to place independent code in separate modules called function or subprogram. In order to make a program using function, we need to perform the followling three steps.

- Function declaration
- Function definition
- Function call

Like variables, all the functions must be declared. Function declaration statement includes function name, what function will take and what function will return.

Syntax :

```
return-type function-name(argument list);
```

return-type : type of value function will return.

function-name : any valid C identifier.

argument list : represents the type and number of value function will take, values are sent by the calling statement.

Example for declaration of function

If we want to return the sum of two integer numbers and function will take two numbers as argument then the function declaration statement will be:

```
int Add(int, int);
```

Function definition includes the actual working or implementation.

Syntax for defining function

```
return-type function-name(argument list)
{
          - - - - - - - - - -
          body of function
          - - - - - - - - - -
}
```

The body of function contains the number of statements to perform specific task.

Example for definition of function

The body of function for calculating sum of two integer numbers.

```
int Add(int x,int y)
{
     int sum;
```

```
        sum = x + y;


       return sum;

}
```

To execute function we must call it. A function can be called or invoked by using function name followed by list of arguments (values) that function definition will recieve to perform task.

Syntax for calling or invoke function

```
var = function-name(val1,val2...n);
```

var can be any variable that will recieve value returning from function definition.

Example for calling or invoke function

Considering the above example, function calling statement should be :

```
int rs;

rs = Add(10,20);        //calling statement

printf("\nThe sum is : %d",rs);
```

Like normal variable, pointer variable can be passed as function argument and it can return from function.

There are two approaches to passing argument to a function:

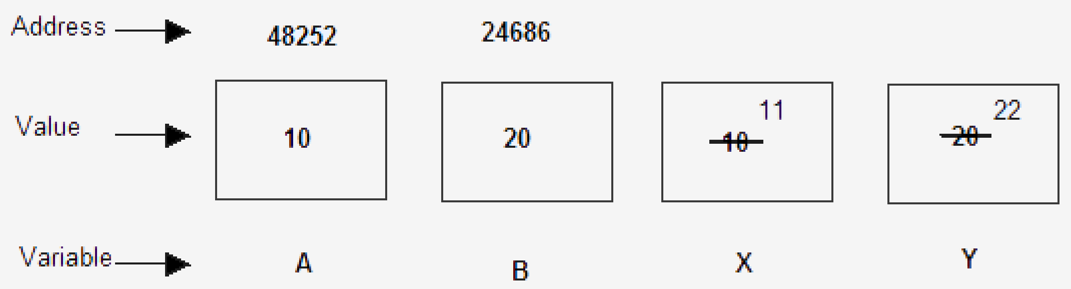
- Call by Value
- Call by Reference/Address

In this approach, the values are passed as function argument to the definition of function.

Example of call by value

```
        #include<stdio.h>

    void main()
    {
        int A=10,B=20;

        printf("\nValues before calling %d, %d",A,B);

        fun(A,B);                    //Statement   1

        printf("\nValues after  calling %d, %d",A,B);

    }

    void fun(int X,int Y)            //Statement   2
    {
        X=11;
        Y=22;
    }

  Output :

        Values before calling 10, 20
        Values after  calling 10, 20
```

In the above example, statement 1 is passing the values of A and B to the calling function fun(). fun() will recieve the value of A and B and put it into X and Y respectively. X and Y are value type variables and are local to fun(). Any changes made by value type variables X and Y will not effect the values of A and B.



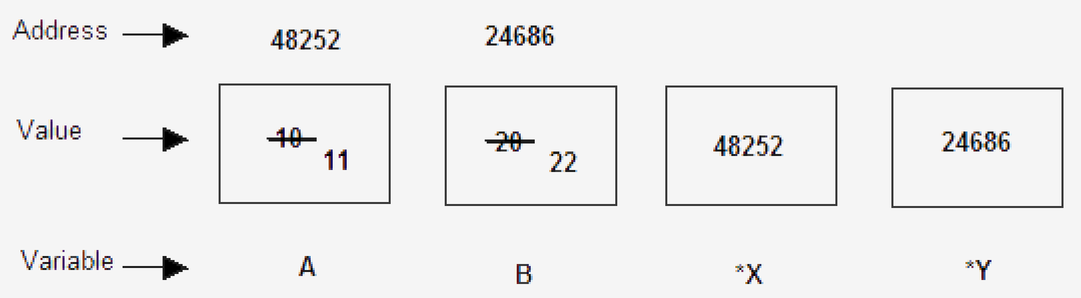
In this approach, the references/addresses are passed as function argument to the definition of function.

Example of call by reference

```
          #include<stdio.h>

     void main()
     {
          int A=10,B=20;

          printf("\nValues before calling %d, %d",A,B);

          fun(&A,&B);                    //Statement    1

          printf("\nValues after  calling %d, %d",A,B);
```

```
    }

    void fun(int *X,int *Y)                    //Statement     2
    {
           *X=11;
           *Y=22;
    }

Output :

           Values before calling 10, 20
           Values after  calling 11, 22
```

In the above example, statement 1 is passing the reference of A and B to the calling function fun(). fun() must have pointer formal arguments to recieve the reference of A and B. In statement 2 *X and *Y is recieving the reference A and B. *X and *Y are reference type variables and are local to fun(). Any changes made by reference type variables *X and *Y will change the values of A and B respectively.



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.

Pass arrays to a function in C

In this tutorial, you'll learn to pass arrays (both one-dimensional and multidimensional arrays) to a function in C programming with the help of examples.

In C programming, you can pass en entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions.

Passing individual array elements

Passing array elements to a function is similar to passing variables to a function.

Example 1: Passing an array

```
#include <stdio.h>
void display(int age1, int age2)
{
    printf("%d\n", age1);
    printf("%d\n", age2);
}

int main()
{
    int ageArray[] = {2, 8, 4, 12};

    // Passing second and third elements to display()
    display(ageArray[1], ageArray[2]);
    return 0;
}
```

Output

```
8

4
```

Example 2: Passing arrays to functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float age[]);

int main() {
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
    // age array is passed to calculateSum()
    result = calculateSum(age);
    printf("Result = %.2f", result);
    return 0;
}

float calculateSum(float age[]) {

  float sum = 0.0;

  for (int i = 0; i < 6; ++i) {
		sum += age[i];
  }

  return sum;
}
```

Output

```
Result = 162.50
```

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result =  calculateSum(age);
```

However, notice the use of `[]` in the function definition.

```
float calculateSum(float age[]) {
... ..
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

Passing Multidimensional Arrays to a Function

To pass multidimensional arrays to a function, only the name of the array is passed to the function(similar to one-dimensional arrays).

Example 3: Passing two-dimensional arrays

```c
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
    int num[2][2];
    printf("Enter 4 numbers:\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to a function
    displayNumbers(num);
    return 0;
}

void displayNumbers(int num[2][2])
{
    printf("Displaying:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
           printf("%d\n", num[i][j]);
        }
    }
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
```

```
2
3
4
5
```

> **Note:** In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

Types of User-defined Functions in C Programming.

These 4 programs below check whether the integer entered by the user is a prime number or not.

The output of all these programs below is the same, and we have created a user-defined function in each example. However, the approach we have taken in each example is different.

Example 1: No arguments passed and no return value

```c
#include <stdio.h>

void checkPrimeNumber();

int main()
{
    checkPrimeNumber();    // argument is not passed
    return 0;
}
```

```
// return type is void meaning doesn't return any value
void checkPrimeNumber()
{
    int n, i, flag = 0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

The `checkPrimeNumber()` function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in `checkPrimeNumber();` statement inside the `main()` function indicates that no argument is passed to the function.

The return type of the function is `void`. Hence, no value is returned from the function.

Example 2: No arguments passed but a return value

```
#include <stdio.h>
int getInteger();

int main()
{
```

```c
    int n, i, flag = 0;

   // no argument is passed
    n = getInteger();

    for(i=2; i<=n/2; ++i)
    {
        if(n%i==0){
            flag = 1;
            break;
        }
    }

    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);

    return 0;
}

// returns integer entered by the user
int getInteger()
{
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    return n;
}
```

The empty parentheses in the `n = getInteger();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to `n`.

Here, the `getInteger()` function takes input from the user and returns it. The code to check whether a number is prime or not is inside the `main()` function.

Example 3: Argument passed but no return value

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main()
{
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n)
{
    int i, flag = 0;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```

The integer value entered by the user is passed to the `checkPrimeAndDisplay()` function.

Here, the `checkPrimeAndDisplay()` function checks whether the argument passed is a prime number or not and displays the appropriate message.

Example 4: Argument passed and a return value

```c
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the returned value is assigned to the flag variable
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// int is returned from the function
int checkPrimeNumber(int n)
{
    int i;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

The input from the user is passed to the `checkPrimeNumber()` function.
The `checkPrimeNumber()` function checks whether the passed argument is prime or not.
If the passed argument is a prime number, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to the `flag` variable.
Depending on whether `flag` is 0 or 1, an appropriate message is printed from the `main()` function.

Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing argument and returning a value from the function (example 4) is better.

A function should perform a specific task. The `checkPrimeNumber()` function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

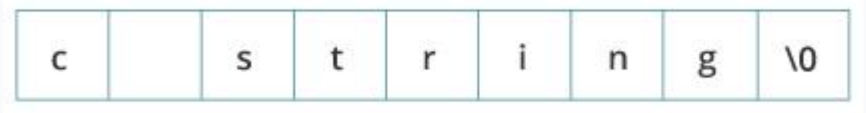
C Programming Strings

In this tutorial, you'll learn about strings in C programming. You'll learn to declare them, initialize them and use them for various I/O operations with the help of examples.

In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

```c
char c[] = "c string";
```

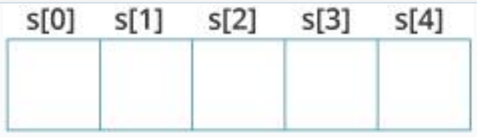
When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.



How to declare a string?

Here's how you can declare strings:

```c
char s[5];
```

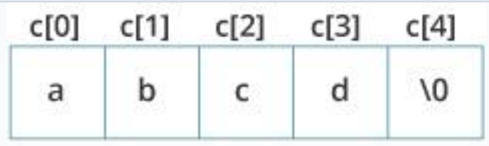


Here, we have declared a string of 5 characters.

How to initialize strings?

You can initialize strings in a number of ways.

```c
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



Let's take another example:

```c
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is `'\0'`) to a `char` array having 5 characters. This is bad and you should never do this.

Read String from the user

You can use the `scanf()` function to read a string.

The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab etc.).

Example 1: scanf() to read a string

```c
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though `Dennis Ritchie` was entered in the above program, only `"Ritchie"` was stored in the `name` string. It's because there was a space after `Dennis`.

How to read a line of text?

You can use the `fgets()` function to read a line of string. And, you can use `puts()` to display the string.

Example 2: fgets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used `fgets()` function to read a string from the user.

`fgets(name, sizeof(name), stdlin); // read string`

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`.

> **Note:** The `gets()` function can also be to take input from the user. However, it is removed from the C standard.
>
> It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

Example 3: Passing string to a Function

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

Example 4: Strings and Pointers

```
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

Commonly Used String Functions

- **strlen()** - calculates the length of a string
- **strcpy()** - copies a string to another
- **strcmp()** - compares two strings
- **strcat()** - concatenates two strings

String Manipulations In C Programming Using Library Functions

You need to often manipulate strings according to the need of a problem. Most, if not all, of the time string manipulation can be done manually but, this makes programming complex and large.
To solve this, C supports a large number of string handling functions in the standard library `"string.h"`.
Few commonly used string handling functions are discussed below:

Function	Work of Function

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

Strings handling functions are defined under `"string.h"` header file.

```
#include <string.h>
```

Note: You have to include the code below to run string handling functions.

gets() and puts()

Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned in the previous chapter.

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);     //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

Note: Though, `gets()` and `puts()` function handle strings, both these functions are defined in `"stdio.h"` header file.