UNIT-II

ARRAYS, STRINGS AND FUNCTIONS: Introduction to Arrays: Declaration, Initialization, Single dimensional array, Two-dimensional arrays, Array Manipulations; String operations: length, compare, concatenate, copy; Functions: General form of a function, Function Arguments, Built-in functions, return statement, Recursion.

2.1 IntroductiontoArrays

An array is a collection of data that holds fixed number of values of same type. For example: if you want to store marks of 100 students, you can create an array for it.

float marks[100];

The size and type of arrays cannot be changed after its declaration.

2.1.1 Declaration

How to declare an array in C?

data type array name[array size];

For example,

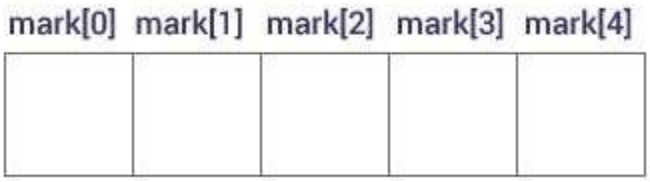
float mark[5];

Here, we declared an array, mark, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and How to access them?

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element ismark [0], second element is mark [1] and so on.



Few key notes:

- Arrays have 0 as the first index not 1. In this example, mark [0]
- If the size of an array is n, to access the last element, (n-1) index is used. In this example, mark [4]
- Suppose the starting address of mark [0] is 2120d. Then, the next address, a [1], will be 2124d, address of a [2] will be 2128d and so on. It is because the size of a float is 4 bytes.

2.1.2 Initialization

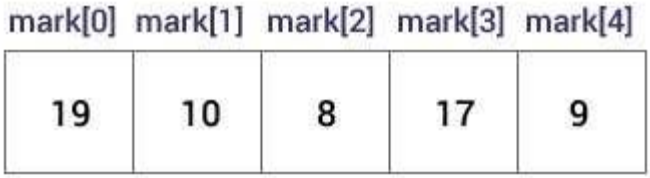
How to initialize an array in C programming?

It is possible to initialize an array during declaration. For example,

int mark [5] = {19, 10, 8, 17, 9};

Another method to initialize array during declaration:

int mark[] = {19, 10, 8, 17, 9};



Here,

mark[0] is equal to 19
mark[1] is equal to 10
mark[2] is equal to 8
mark[3] is equal to 17
mark[4] is equal to 9

How to insert and print array elements?

```
int mark[5] = {19, 10, 8, 17, 9}

// insert different value to third element
mark[3] = 9;

// take input from the user and insert in third element
scanf("%d", &mark[2]);

// take input from the user and insert in (i+1)th element
scanf("%d", &mark[i]);

// print first element of an array
printf("%d", mark[0]);

// print ith element of an array
printf("%d", mark[i-1]);
```

Example: C Arrays

```
// Program to find the average of n (n < 10) numbers using arrays

#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;
    printf("Enter n: ");
    scanf("%d", &n);
    for(i=0; i<n; ++i)
    {
        printf("Enter number%d: ",i+1);
        scanf("%d", &marks[i]);
```

```
        sum += marks[i];
    }
    average = sum/n;

    printf("Average = %d", average);

    return 0;
}
```

Output

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

Important thing to remember when working with C arrays

Suppose you declared an array of 10 elements. Let's say,

```
int testArray[10];
```

You can use the array members from testArray[0] to testArray[9].

If you try to access array elements outside of its bound, let's say testArray[12], the compiler may not show any error. However, this may cause unexpected output (undefined behavior).

2.2Onedimensionalarrayincprogramminglanguage: IDarray

The declaration form of one dimensional array is

Data_type array_name [size];

The following declares an array called ‘numbers’ to hold 5 integers and sets the first and last elements. C arrays are always indexed from 0. So the first integer in ‘numbers’ array is numbers[0] and the last is numbers[4].

```
int  numbers [5];
numbers  [0] = 1;  //   set first element
numbers [4] = 5;       // set last element
```

This array contains 5 elements. Any one of these elements may be referred to by giving the name of the array followed by the position number of the particular element in square brackets ([]). The first element in every array is the zeroth element.Thus, the first element of array ‘numbers’is referred to asnumbers[0], the second element of array ‘numbers’is referred to as numbers[1], the fifth element of array ‘numbers’is referred to as numbers[4], and, in general, the n-th element of array ‘numbers’is referred to as numbers[n – 1].

Example:

```
1  #include<stdio.h>
2  #include<conio.h>
3  int main()
```

```
4  {
5  int i,number[5];
6  clrscr();
7  printf("Enter 5 number \n");
8  for(i=0;i<5;i++)
9  scanf("%d",&numbers[i]);
10 printf("Array elements are \n");
11 for(i=0;i<5;i++)
12 printf("%d\n",numbers[i]);
13 getch();
14 return 0;
15 }
```

Output :

```
4
6
7
3
2
Array elements are
4
6
7
3
2
```

It's a very common error to try to refer to non-existent numbers[5], element. C does not do much hand holding. It is invariably up to the programmer to make sure that programs are free from errors. This is especially true with arrays. C does not complain if you try to write to elements of an array which do not exist!

For example: If you wrote: numbers[5] = 6; C would happily try to write 6 at the location which would have corresponded to the sixth element, had it been declared that way. Unfortunately this would probably be memory taken up by some other variable or perhaps even by the operating system. The result would be either:

- The value in the incorrect memory location would be corrupted with unpredictable consequences.
- The value would corrupt the memory and crash the program completely!

The second of these tends to be the result on operating systems with proper memory protection. Writing over the bounds of an array is a common source of error. Remember that the array limits run from zero to the size of the array minus one.

Out of bounds access :

I would be doing well, that really shows the range of valid indices, please check often.

10, when the number of elements, all the indices of 10 or more is illegal. Also, whether any size of negative indices is also illegal.

These errors are detected at compile time is not. It is an error at run time. However, there is no guarantee that the error reliably. Sometimes works correctly (seems to be moving.) This is the "luck ran" instead of "bad luck worked" so, please, as no doubt.

The best way to see these principles is by use of an example, so load the program and display it on your monitor.

The elements of an array can also be initialized in the array declaration by following the declaration with an equal sign and a comma-separated list (enclosed in braces) of *initializers.*

Example:

```
1  #include <stdio.h>
2  #include <conio.h>
3  int main( )
4  {
5  char name[7]; /* define a string of characters */
6  name[0] = 'A';
7  name[1] = 's';
8  name[2] = 'h';
9  name[3] = 'r';
10 name[4] = 'a';
11 name[5] = 'f';
12 name[6] = '\0'; /* Null character - end of text */
13 name[7] = 'X';
14 clrscr();
15 printf("My name is %s\n",name);
16 printf("First letter is %c\n",name[0]);
17 printf("Fifth letter is %c\n",name[4]);
18 printf("Sixth letter is %c\n",name[5]);
19 printf("Seventh letter is %c\n",name[6]);
20 printf("Eight letter is %c\n",name[7]);
21 getch();
22 return 0;
23 }
```

The following program initializes an integer array with five values and prints the array.

```
1  #include <stdio.h>
2  #include <conio.h>
3  int main()
4  {
5  int numbers[]={1,2,3,4,5};
6  int i;
7  clrscr();
8  printf("Array elements are\n");
9  for(i=0;i<=4;i++)
10 printf("%d\n",numbers[i]);
11 getch();
12 return 0;
13 }
```

If there are fewer initializers than elements in the array, the remaining elements are initialized to zero. For example, the elements of the array n could have been initialized to zero with the declaration int n[10] = {0}; which explicitly initializes the first element to zero and initializes the remaining nine

elements to zero because there are fewer initializers than there are elements in the array. It is important to remember that arrays are not automatically initialized to zero. The programmer must at least initialize the first element to zero for the remaining elements to be automatically zeroed. This method of initializing the array elements to 0 is performed at compile time for static arrays and at run time for automatic arrays.

2.3:Example

Computing mean, median and mode

What is mean ?

Mean is same as average. The mean is found by adding up all of the given data and dividing by the number of elements.

Example: Mean of 1,2,3,4,5 is

(1+2+3+4+5)/5 = 15/3 = 3

What is Median ?

The median is the middle number in an ordered list (ascending or descending). First you arrange the numbers in orders in ascending order, then you find the middle number and save it as median

Example:

1 2 3 4 5

Median is 3

What is Mode ?

Mode is the element which happens most number of time in the list. If no element happens more than once, all elements are considered as mode.

Algorithm

1. Start
2. Declare an array a[20], sum =0,i=0,j=0,z=0
3. Read number of terms n
4. Repeat step 5 & 6 while (i<n)
5. sum=sum+a[i]
6. next i
7. mean=sum/n
8. if(n%2=0)
   median=(a[n/2]+a[n/2-1])/2 else
   median=a[n/2]
9. print mean and median
10. repeat step 11 & 12 when i=0 to n, j=0 to i
11. if a[i]=a[j], then b[i]++
12. if b[i]>z, then z=b[i]
13. For i=0 to n if b[i]=z, print b[i]
14. end

Program:

```
#include <stdio.h>
main()
{ int i,j,x,k=0,n,a[20],z=0,b[20];
float sum=0, t,mean,medn,mod;
printf("\n Enter the no. of elements (Max 20)\n");
```

```
scanf("%d",&n);
printf("Enter the elements\n");
for(i=0;i
{scan ("%d",&a[i]);}
for(i=0;i
{sum=sum+a[i];}
printf("Sum: %f",sum);
mean=sum/n;
}
for(i=0;i
{ for (j=i+1;j
  { if (a[i]>a[j])
     { t=a[i];a[i]=a[j];a[j]=t; }
 }
}
if (n%2==0)
medn=(a[n/2]+a[(n/2)-1])/2; else
medn=a[n/2];
for(i=0;i
{ for ( j=0; j<i; j++ )
  { if (a[i]==a[j])
     b[i]++;
  }
}
for(i=0;i<n;i++)
{ if (b[i]>z)
   z=b[i];
}
printf("Mean :%f \n Median : %f \n Mean : ", mean, medn);
for I i=0; i<n; i++)
 { if (b[i]==z)
    printf("%d\t", a[i]);
}
}
```

2.4. Two-Dimensional Array

- Two-dimensional array are those type of array, which has finite number of rows and finite number of columns. The declaration form of 2-dimensional array is

Data_type Array_name [row size][column size];

- The type may be any valid type supported by C.
- The rule for giving the array name is same as the ordinary variable.
- The row size and column size should be an individual constant.

The following declares a two-dimensional 3 by 3 array of integers and sets the first and last elements to be 10.

```
int matrix [3][3];
matrix[0][0] = 10;
matrix[2][2] = 10;
```

The following figure illustrates a two dimensional array, **matrix**. The array contains three rows and tree columns, so it is said to be a 3-by-3 array. In general, an array with *m* rows and *n* columns is called an *m-by-n array.*

	[0]	[1]	[2]
[0]	10		
[1]			
[2]			10

Every element in array matrix is identified by an element name of the form matrix[i][j]; matrix is the name of the array, and i and j are the subscripts that uniquely identify each element in matrix . Notice that the names of the elements in the first row all have a first subscript of 0; the names of the elements in the third column all have a second subscript of 2.

In the case of Two-dimensional array, during declaration the maximum number of rows and maximum number of column should be specified for processing all array elements. The implementation of the array stores all the elements in a single contiguous block of memory. The other possible implementation would be a combination of several distinct one-dimensional arrays. That's not how C does it. In memory, the array is arranged with the elements of the rightmost index next to each other. In other words, matrix[1][1] comes right before matrix[1][2] in memory.

The following array:

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9

would be stored:

1	2	3	4	5	6	7	8	9

EXAMPLE :

```
1  #include<stdio.h>
2  #include<conio.h>
3  int main()
4  {
5  int matrix [3][3],i,j,r,c;
6  clrscr();
7  printf("Enter the order of matrix\n");
8  scanf("%d%d",&r,&c);
9  printf("Enter elements of %d * %d matrix \n",r,c);
10 for(i=0;i<r;i++)
11 for(j=0;j<c;j++)
12 scanf("%d",&matrix[i][j]);
13 printf("Given matrix:\n");
14 for(i=0;i<r;i++)
15 {
16 for(j=0;j<c;j++)
17 printf("%d\t",matrix[i][j]);
18 printf("\n");
19 }
20 printf("%d\t",matrix[2][2]);
```

```
21 getch();
22 return 0;
23 }
```

Output :

```
Enter the order of matrix
2
2
Enter elements of 2*2 matrix
1
2
3
4
Given matrix :
1       2
3       4
```

2.5 Example

Matrix Operations (Addition, Scaling, Determinant and Transpose)

```
/*  Write a Program to implement the following operations on the matrices :
  1. Addition
  2. Subtraction
  3. Multiplication
  4. Transpose  */
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
  int m,n,a[20][20],b[20][20],i,j,sum[20][20],sub[20][20],opt,tr[20][20],opt1,ch,e,f;
     printf("Note : For Addition or Subtraction , no. of rows and columns should be same and  for transpose of matrices , your first matrices entered should be the desired matrices .\n");
     printf("Enter the no. of rows: ");
     scanf("%d",&m);
     printf("Enter the no. of columns: ");
     scanf("%d",&n);
     printf("Enter the Data Elements of first matrices\n");
   for(i=0;i<m;i++)
    {
      for(j=0;j<n;j++)
       {
         scanf("%d",&a[i][j]);
       }
    } printf("Enter the no. of rows for second matrices: ");
     scanf("%d",&e);
     printf("Enter the no. of columns: ");
     scanf("%d",&f);
     printf("Enter the Data Elements of second matrices\n");
```

```
    for(i=0;i<e;i++)
    {
      for(j=0;j<f;j++)
       {
         scanf("%d",&b[i][j]);
       }
    }
do
     {
     if(m==e&&n==f)
    {
          printf("Enter 1 for addtion or subtraction of matrices\n");
          if(n==e){printf("Enter 2 for multiplication of matrices\n");}
          printf("Enter 3 for transpose of first matrices\n");
    }
  else if(m!=n&&n==e)
  {
         printf("Enter 2 for multiplication of matrices\n");
        printf("Enter 3 for transpose  of matrices\n");

 }
 else
{
printf("Enter 3 for transpose of first matrices\n");
}
   scanf("%d",&ch);
   switch(ch)
    {
       case 1 :
    for(i=0;i<m;i++)
     {
       for(j=0;j<n;j++)
       {
          sum[i][j]=a[i][j]+b[i][j];
          sub[i][j]=a[i][j]-b[i][j];
       }
     }
      printf("Enter 1 for Addition or 2 for Subtraction: ");
      scanf("%d",&opt);
      switch(opt)
   {
       case 1 :
      printf("The resultant matrices is :\n");
    for(i=0;i<m;i++)
     {
       for(j=0;j<n;j++)
       {
          printf("%3d",sum[i][j]);
```

```
    }
  printf("\n");
  }
  break;
     case 2 :
   printf("The resultant matrices is :\n");
 for(i=0;i<m;i++)
  {
    for(j=0;j<n;j++)
    {
      printf("%3d",sub[i][j]);
    }
      printf("\n");
  }
}
  break;
   case 2 :
        printf("The resultant matrices is : \n");
        int k;
        for(i=0;i<m;i++)
        {
          for(j=0;j<f;j++)
          { sum[i][j]=0;
            for(k=0;k<m;k++)
            {
              sum[i][j]+=a[i][k]*b[k][j];
            }
            printf("%d\t",sum[i][j]);
          }
          printf("\n");
        }
   break;
   case 3 :
       for(i=0;i<m;i++)
          {
            for(j=0;j<n;j++)
            {
               tr[j][i]=a[i][j];
            }
          }
        printf("The resultant matrices is :\n");
         for(i=0;i<n;i++)
          {
             for(j=0;j<m;j++)
           {
               printf("%3d",tr[i][j]);
           }
               printf("\n");
          }
```

```
    break;  }}
  while(ch>0);
  getch();
}
```

2.6 StringsOperation

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

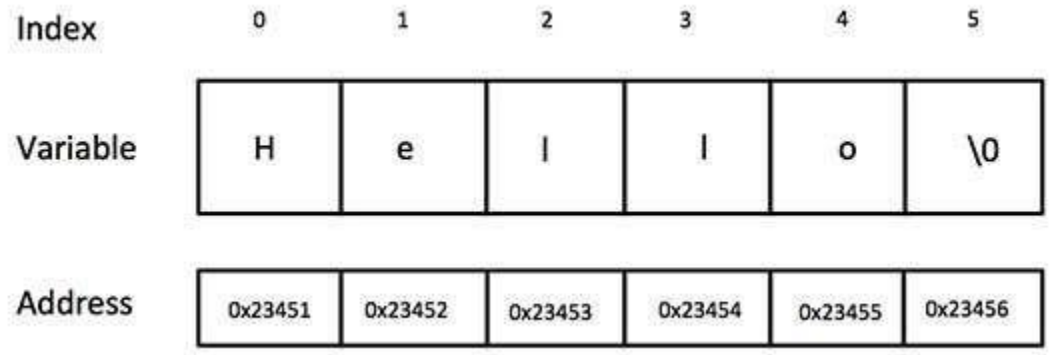
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

```
#include <stdio.h>

int main () {

  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  printf("Greeting message: %s\n", greeting );
  return 0;
}
```

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

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions −

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

int main () {

  char str1[12] = "Hello";
  char str2[12] = "World";
  char str3[12];
  int  len ;

  /* copy str1 into str3 */
  strcpy(str3, str1);
  printf("strcpy( str3, str1) :  %s\n", str3 );

  /* concatenates str1 and str2 */
  strcat( str1, str2);
  printf("strcat( str1, str2):   %s\n", str1 );

  /* total lenghth of str1 after concatenation */
  len = strlen(str1);
  printf("strlen(str1) :  %d\n", len );
```

```
  return 0;
}
```

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

```
strcpy( str3, str1) :  Hello

strcat( str1, str2):   HelloWorld

strlen(str1) :  10
```

2.7 Introduction to functions

A function is a block of code that performs a specific task.

Suppose, a program related to graphics needs to create a circle and color it depending upon the radius and color from the user. You can create two functions to solve this problem:

- create a circle function
- color function

Dividing complex problem into small components makes program easy to understand and use.

Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

Standard library functions

How user-defined function works?

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h"header file. There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

User-defined functions

As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

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

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

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

The execution of a C program begins from the main() function.
When the compiler encounters functionName(); inside the main function, control of the program jumps to

```
void functionName()
```

And, the compiler starts executing the codes inside the user-defined function.
The control of the program jumps to statement next to functionName(); once all the codes inside the function definition are executed.
A function is a block of code that performs a specific task.
C allows you to define functions according to your need. These functions are known as user-defined functions. For example:
Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- createCircle() function
- color() function

Example: User-defined function

Here is a example to add two integers. To perform this task, a user-defined function addNumbers() is defined.

```
#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
}
```

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

1. name of the function is addNumbers()
2. return type of the function is int
3. 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.

2.7.2 Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, 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.

2.7.3 Function Call

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

Syntax of function call

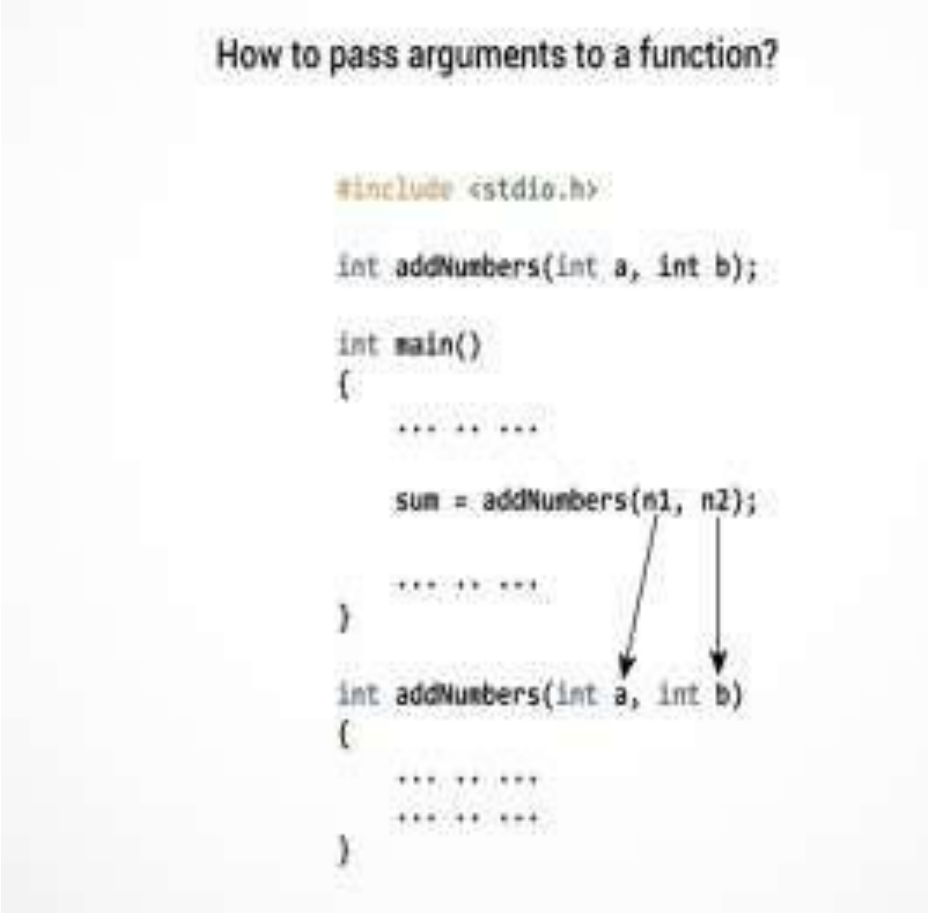
functionName(argument1, argument2, ...);

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

Passing arguments to a function

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

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



The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws 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.

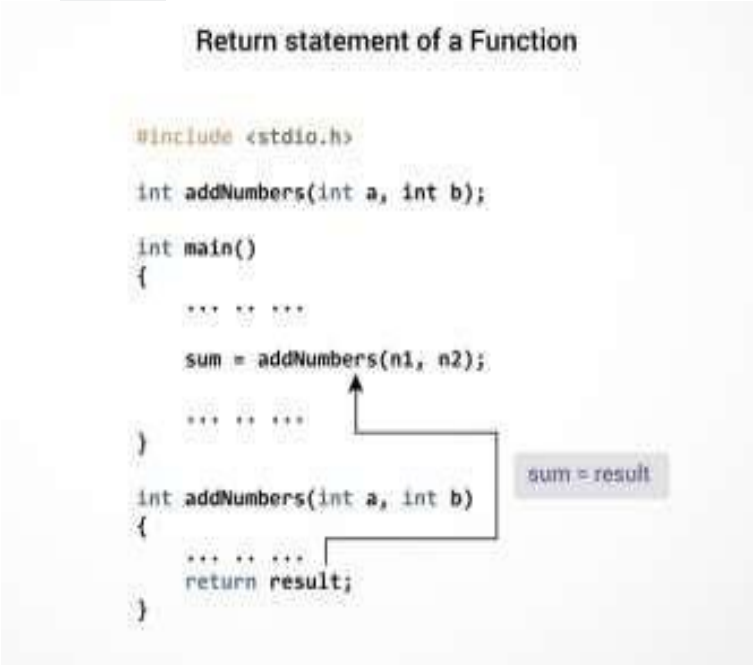
A function can also be called without passing an argument.

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 return statement.

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

.



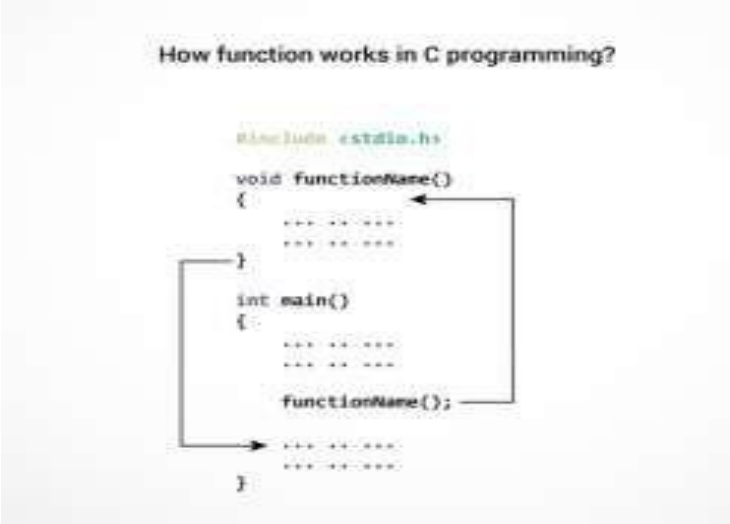
Syntax of return statement

return (expression);

For example,

return a;
return (a+b);

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



Remember, function name is an identifier and should be unique. This is just an overview on user-defined function.

- User-defined Function in C programming
- Types of user-defined Functions

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

For better understanding of arguments and return value from the function, user-defined functions can be categorized as:

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value.

The 4 programs below check whether an integer entered by the user is a prime number or not. And, all these programs generate the same output.

Example #1: No arguments passed and no return Value

```
#include <stdio.h>

void checkPrimeNumber();

int main()
{
   checkPrimeNumber();    // no argument is passed to prime()
   return 0;
}

// return type of the function is void becuase no value is returned from the function 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()
{
    int n, i, flag = 0;
    // no argument is passed to the function
    // the value returned from the function is assigned to n 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;
}
// getInteger() function returns integer entered by the user int
getInteger()
{
    int n;

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

    return n;
}
```

The empty parentheses in 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;
}

// void indicates that no value is returned from the function 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 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

```
#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 value returned from the function is assigned to 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;
}

// integer is returned from the function int
checkPrimeNumber(int n)
{
   /* Integer value is returned from function checkPrimeNumber() */
   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 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 flag variable.
Then, the appropriate message is displayed from the main() function.

Which approach is better?

Well, it depends on the problem you are trying to solve. In case of this problem, the last approach 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, which makes code modular, easy to understand and debug.

2.8 Built-in functions

Some of the "commands" in C are not really "commands" at all but are functions. For example, we have been using **printf** and **scanf** to do input and output, and we have used **rand** to generate random numbers - all three are functions.

There are a great many standard functions that are included with C compilers and while these are not really part of the language, in the sense that you can re-write them if you really want to, most C programmers think of them as fixtures and fittings. Later in the course we will look into the mysteries of how C gains access to these standard functions and how we can extend the range of the standard library. But for now a list of the most common libraries and a brief description of the most useful functions they contain follows:

1. **stdio.h: I/O functions:**
   a. **getchar()** returns the next character typed on the keyboard. b.
   putchar() outputs a single character to the screen.
   c. **printf()** as previously described d.
   scanf() as previously described
2. **string.h: String functions**
   a. **strcat()** concatenates a copy of str2 to str1 b.
   strcmp() compares two strings
   c. **strcpy()** copys contents of str2 to str1
3. **ctype.h: Character functions**
   a. **isdigit()** returns non-0 if arg is digit 0 to 9
   b. **isalpha()** returns non-0 if arg is a letter of the alphabet
   c. **isalnum()** returns non-0 if arg is a letter or digit
   d. **islower()** returns non-0 if arg is lowercase letter
   e. **isupper()** returns non-0 if arg is uppercase letter
4. **math.h: Mathematics functions**
   a. **acos()** returns arc cosine of arg
   b. **asin()** returns arc sine of arg
   c. **atan()** returns arc tangent of arg
   d. **cos()** returns cosine of arg
   e. **exp()** returns natural logarithim e
   f. **fabs()** returns absolute value of num
   g. **sqrt()** returns square root of num
5. **time.h: Time and Date functions**
   a. **time()** returns current calender time of system
   b. **difftime()** returns difference in secs between two times
   c. **clock()** returns number of system clock cycles since program execution
6. **stdlib.h:Miscellaneous functions**
   a. **malloc()** provides dynamic memory allocation, covered in future sections b.
   rand() as already described previously

c. **srand()** used to set the starting point for rand()

List Of Inbuilt C Functions In Math.h File:

"math.h" header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.

List Of Inbuilt C Functions In Math.h File:

"math.h" header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.

Function	Description
floor ( )	This function returns the nearest integer which is less than or equal to the argument passed to this function.
round ( )	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.
ceil ( )	This function returns nearest integer value which is greater than or equal to the argument passed to this function.
sin ( )	This function is used to calculate sine value.
cos ( )	This function is used to calculate cosine.
cosh ( )	This function is used to calculate hyperbolic cosine.
exp ( )	This function is used to calculate the exponential "e" to the x^{th} power.
tan ( )	This function is used to calculate tangent.
tanh ( )	This function is used to calculate hyperbolic tangent.
sinh ( )	This function is used to calculate hyperbolic sine

log ( )	This function is used to calculates natural logarithm
log10 ( )	This function is used to calculates base 10 logarithm
sqrt ( )	This function is used to find square root of the argument passed to this function
pow ( )	This is used to find the power of the given number.
trunc.(.)	This function truncates the decimal value from floating point value and returns integer value

List Of Inbuilt C Functions In string.h File

All C inbuilt functions which are declared in string.h header file are given below. The source code for string.h header file is also given below for your reference.

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns>0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1

strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter
memset()	It is used to initialize a specified number of bytes to null or any other value in the buffer
memcpy()	It is used to copy a specified number of bytes from one memory to another
memmove()	It is used to copy a specified number of bytes from one memory to another or to overlap on same memory.
memcmp()	It is used to compare specified number of characters from two buffers
memicmp()	It is used to compare specified number of characters from two buffers regardless of the case of the characters
memchr()	It is used to locate the first occurrence of the character in the specified string

2.9 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.

```
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.

NumberFactorial

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

```
#include <stdio.h>

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>

int fibonacci(int i) {
```

```
  if(i == 0) {
    return 0;
  }

  if(i == 1) {
    return 1;
  }
  return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

  int i;

  for (i = 0; i < 10; i++) {
    printf("%d\t\n", fibonacci(i));
  }

  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
```