

PROGRAMMING PRINCIPLES AND ALGORITHMS

BCA 103

SELF LEARNING MATERIAL



**DIRECTORATE
OF DISTANCE EDUCATION**

SWAMI VIVEKANAND SUBHARTI UNIVERSITY

MEERUT – 250 005,

UTTAR PRADESH (INDIA)

SLM Module Developed By :

Author:

Reviewed by :

Assessed by:

Study Material Assessment Committee, as per the SVSU ordinance No. VI (2)

Copyright © **Gayatri Sales**

DISCLAIMER

No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior permission from the publisher.

Information contained in this book has been published by Directorate of Distance Education and has been obtained by its authors from sources believed to be reliable and are correct to the best of their knowledge. However, the publisher and its author shall in no event be liable for any errors, omissions or damages arising out of use of this information and specially disclaim and implied warranties or merchantability or fitness for any particular use.

Published by: Gayatri Sales

Typeset at: Micron Computers

Printed at: Gayatri Sales, Meerut.

UNIT-I

Introduction to 'C' Language

History, Structures of 'C' Programming, Function as building blocks.

Language Fundamentals

Character set, C Tokens, Keywords, Identifiers, Variables, Constant, Data Types, Comments.

UNIT-II

Operators

Types of operators, Precedence and Associativity, Expression, Statement and types of statements

Build in Operators and function

Console based I/O and related built in I/O function: printf(), scanf(), getch(), getchar(), putchar(); Concept of header files, Preprocessor directives: #include, #define.

Control structures

Decision making structures: If, If-else, Nested If-else, Switch; Loop Control structures: While, Do-while, for, Nested for loop; Other statements: break, continue, goto, exit.

UNIT-III

Introduction to problem solving

Concept: problem solving, Problem solving techniques (Trail & Error, Brain Storming, Divide & Conquer) Steps in problem solving (Define Problem, Analyze Problem, Explore Solution) Algorithms and Flowcharts (Definitions, Symbols), Characteristics of an algorithm Conditionals in pseudo-code, Loops in pseudo code Time complexity: Big-Oh notation, efficiency Simple Examples: Algorithms and flowcharts (Real Life Examples)

UNIT-IV

Simple Arithmetic Problems

Addition / Multiplication of integers, Determining if a number is +ve / -ve / even / odd, Maximum of 2 numbers, 3 numbers, Sum of first n numbers, given n numbers, Integer division, Digit reversing, Table generation for n , a^b , Factorial, sine series, cosine series, ${}^n C_r$, Pascal Triangle, Prime number, Factors of a number, Other problems such as Perfect number, GCD numbers etc (Write algorithms and draw flowchart), Swapping

UNIT-V

Functions

Basic types of function, Declaration and definition, Function call, Types of function, Parameter passing, Call by value, Call by reference, Scope of variable, Storage classes, Recursion.

UNIT- I

Introduction to 'C' Language

History

What is C programming?

C is a general-purpose programming language that is extremely popular, simple and flexible. It is machine-independent, structured programming language which is used extensively in various applications.

C was the basic language to write everything from operating systems (Windows and many others) to complex programs like the Oracle database, Git, Python interpreter and more.

It is said that 'C' is a god's programming language. One can say, C is a base for the programming. If you know 'C,' you can easily grasp the knowledge of the other programming languages that uses the concept of 'C'

It is essential to have a background in computer memory mechanisms because it is an important aspect when dealing with the C programming language.

History of C language

The base or father of programming languages is 'ALGOL.' It was first introduced in 1960. 'ALGOL' was used on a large basis in European countries. 'ALGOL' introduced the concept of structured programming to the developer community. In 1967, a new computer programming language was announced called as 'BCPL' which stands for Basic Combined Programming Language. BCPL was designed and developed by Martin Richards, especially for writing system software. This was the era of programming languages. Just after three years, in 1970 a new programming language called 'B' was introduced by Ken Thompson that contained multiple features of 'BCPL.' This programming language was created using UNIX operating system at AT&T and Bell Laboratories. Both the 'BCPL' and 'B' were system programming languages.

In 1972, a great computer scientist Dennis Ritchie created a new programming language called 'C' at the Bell Laboratories. It was created from 'ALGOL', 'BCPL' and 'B' programming languages. 'C' programming language contains all the features of these languages and many more additional concepts that make it unique from other languages.

'C' is a powerful programming language which is strongly associated with the UNIX operating system. Even most of the UNIX operating system is coded in 'C'. Initially 'C' programming was limited to the UNIX operating system, but as it started spreading

around the world, it became commercial, and many compilers were released for cross-platform systems. Today 'C' runs under a variety of operating systems and hardware platforms. As it started evolving many different versions of the language were released. At times it became difficult for the developers to keep up with the latest version as the systems were running under the older versions. To assure that 'C' language will remain standard, American National Standards Institute (ANSI) defined a commercial standard for 'C' language in 1989. Later, it was approved by the International Standards Organization (ISO) in 1990. 'C' programming language is also called as 'ANSI C'.

Languages such as C++/Java are developed from 'C'. These languages are widely used in various technologies. Thus, 'C' forms a base for many other languages that are currently in use.

Where is C used? Key Applications

1. 'C' language is widely used in embedded systems.
2. It is used for developing system applications.
3. It is widely used for developing desktop applications.
4. Most of the applications by Adobe are developed using 'C' programming language.
5. It is used for developing browsers and their extensions. Google's Chromium is built using 'C' programming language.
6. It is used to develop databases. MySQL is the most popular database software which is built using 'C'.
7. It is used in developing an operating system. Operating systems such as Apple's OS X, Microsoft's Windows, and Symbian are developed using 'C' language. It is used for developing desktop as well as mobile phone's operating system.
8. It is used for compiler production.
9. It is widely used in IOT applications.

Why learn 'C'?

As we studied earlier, 'C' is a base language for many programming languages. So, learning 'C' as the main language will play an important role while studying other programming languages. It shares the same concepts such as data types, operators, control statements and many more. 'C' can be used widely in various applications. It is a simple language and provides faster execution. There are many jobs available for a 'C' developer in the current market.

'C' is a structured programming language in which program is divided into various modules. Each module can be written separately and together it forms a single 'C' program. This structure makes it easy for testing, maintaining and debugging processes.

'C' contains 32 keywords, various data types and a set of powerful built-in functions that make programming very efficient.

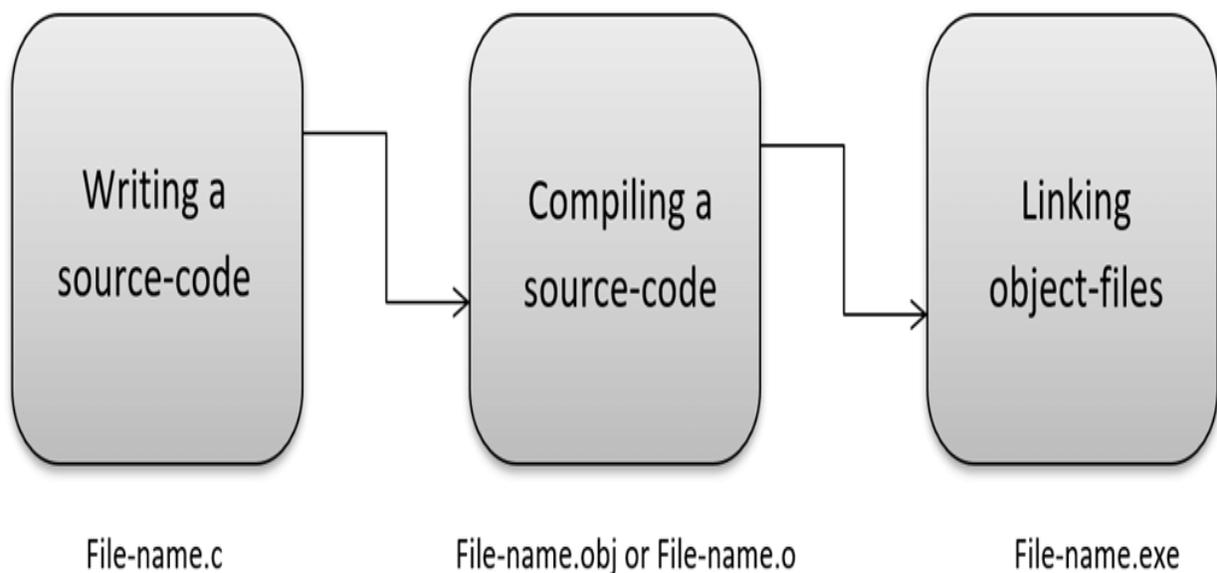
Another feature of 'C' programming is that it can extend itself. A 'C' program contains various functions which are part of a library. We can add our features and functions to the library. We can access and use these functions anytime we want in our program. This feature makes it simple while working with complex programming.

Various compilers are available in the market that can be used for executing programs written in this language.

It is a highly portable language which means programs written in 'C' language can run on other machines. This feature is essential if we wish to use or execute the code on another computer.

How 'C' Works?

C is a compiled language. A compiler is a special tool that compiles the program and converts it into the object file which is machine readable. After the compilation process, the linker will combine different object files and creates a single executable file to run the program. The following diagram shows the execution of a 'C' program



Nowadays, various compilers are available online, and you can use any of those compilers. The functionality will never differ and most of the compilers will provide the features required to execute both 'C' and 'C++' programs.

Following is the list of popular compilers available online:

- Clang compiler
- MinGW compiler (Minimalist GNU for Windows)

- Portable 'C' compiler
- Turbo C

Summary

- 'C' was developed by Dennis Ritchie in 1972.
- It is a robust language.
- It is a low programming level language close to machine language
- It is widely used in the software development field.
- It is a procedure and structure oriented language.
- It has the full support of various operating systems and hardware platforms.
- Many compilers are available for executing programs written in 'C'.
- A compiler compiles the source file and generates an object file.
- A linker links all the object files together and creates one executable file.
- It is highly portable.

Structures of 'C' Programming

Structure is a group of variables of different data types represented by a single name. Lets take an example to understand the need of a structure in C programming.

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

How to create a structure in C Programming

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type**.

```
struct struct_name {  
    DataType member1_name;  
    DataType member2_name;
```

```
DataType member3_name;  
...  
};
```

Here struct_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

First we will see the syntax of creating struct variable, accessing struct members etc and then we will see a complete example.

How to declare variable of a structure?

```
struct struct_name var_name;
```

or

```
struct struct_name {  
    DataType member1_name;  
    DataType member2_name;  
    DataType member3_name;  
    ...  
} var_name;
```

How to access data members of a structure using a struct variable?

```
var_name.member1_name;  
var_name.member2_name;  
...
```

How to assign values to structure members?

There are three ways to do this.

1) Using Dot(.) operator

```
var_name.memeber_name = value;
```

2) All members assigned in one statement

```
struct struct_name var_name =  
{value for memeber1, value for memeber2 ...so on for all the members}
```

3) **Designated initializers** – We will discuss this later at the end of this post.

Example of Structure in C

```
#include <stdio.h>
/* Created a structure here. The name of the structure is
 * StudentData.
 */
struct StudentData{
    char *stu_name;
    int stu_id;
    int stu_age;
};
int main()
{
    /* student is the variable of structure StudentData*/
    struct StudentData student;

    /*Assigning the values of each struct member here*/
    student.stu_name = "Steve";
    student.stu_id = 1234;
    student.stu_age = 30;

    /* Displaying the values of struct members */
    printf("Student Name is: %s", student.stu_name);
    printf("\nStudent Id is: %d", student.stu_id);
    printf("\nStudent Age is: %d", student.stu_age);
    return 0;
}
```

Output:

```
Student Name is: Steve
Student Id is: 1234
Student Age is: 30
```

Nested Structure in C: Struct inside another struct

You can use a structure inside another structure, which is fairly possible. As I explained above that once you declared a structure, the **struct struct_name** acts as a new data type so you can include it in another struct just like the data type of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure in C Programming

Lets say we have two structure like this:

Structure 1: stu_address

```

struct stu_address
{
    int street;
    char *state;
    char *city;
    char *country;
}

```

Structure 2: stu_data

```

struct stu_data
{
    int stu_id;
    int stu_age;
    char *stu_name;
    struct stu_address stuAddress;
}

```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Lets take the example of the two structure that we seen above to understand the logic

```

struct stu_data mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..

```

How to access nested structure members?

Using chain of “.” operator.

Suppose you want to display the city alone from nested struct –

```

printf("%s", mydata.stuAddress.city);

```

Use of typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct.

Code without typedef

```
struct home_address {
    int local_street;
    char *town;
    char *my_city;
    char *my_country;
};
...
struct home_address var;
var.town = "Agra";
```

Code using typedef

```
typedef struct home_address{
    int local_street;
    char *town;
    char *my_city;
    char *my_country;
}addr;
..
..
addr var1;
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined.

Designated initializers to set values of Structure members

We have already learned two ways to set the values of a struct member, there is another way to do the same using designated initializers. This is useful when we are doing assignment of only few members of the structure. In the following example the structure variable s2 has only one member assignment.

```
#include <stdio.h>
struct numbers
{
    int num1, num2;
};
int main()
{
    // Assignment using using designated initialization
```

```

struct numbers s1 = {.num2 = 22, .num1 = 11};
struct numbers s2 = {.num2 = 30};

printf ("num1: %d, num2: %d\n", s1.num1, s1.num2);
printf ("num1: %d", s2.num2);
return 0;
}

```

Output:

```

num1: 11, num2: 22
num1: 30

```

A pointer is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, pointer holds the address of a variable. For example, an integer variable holds (or you can say stores) an integer value, however an integer pointer holds the address of a integer variable. In this guide, we will discuss pointers in C programming with the help of examples.

Before we discuss about pointers in C, lets take a simple example to understand what do we mean by the address of a variable.

A simple example to understand how to access the address of a variable without pointers?

In this program, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory, right? A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

So let's say the address assigned to variable num is 0x7fff5694dc58, which means whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58. See the diagram below.

```

#include <stdio.h>
int main()
{
    int num = 10;
    printf("Value of variable num is: %d", num);
    /* To print the address of a variable we use %p
    * format specifier and ampersand (&) sign just
    * before the variable name like &num.

```

```
*/  
printf("\nAddress of variable num is: %p", &num);  
return 0;  
}
```

Output:

```
Value of variable num is: 10  
Address of variable num is: 0x7fff5694dc58
```

A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

```
#include <stdio.h>  
int main()  
{  
    //Variable declaration  
    int num = 10;  
  
    //Pointer declaration  
    int *p;  
  
    //Assigning address of num to the pointer p  
    p = #  
  
    printf("Address of variable num is: %p", p);  
    return 0;  
}
```

Output:

```
Address of variable num is: 0x7fff5694dc58
```

C Pointers – Operators that are used with Pointers

Lets discuss the operators & and * that are used with Pointers in C.

“Address of”(&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The & operator is also known as “Address of” Operator.

```
printf("Address of var is: %p", &num);
```

Point to note: %p is a format specifier which is used for displaying the address in hex format.

Now that you know how to get the address of a variable but how to store that address in some other variable? That’s where pointers comes into picture. As mentioned in the beginning of this guide, pointers in C programming are used for holding the address of another variables.

Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.

“Value at Address”(*) Operator

The * Operator is also known as Value at address operator.

How to declare a pointer?

```
int *p1 /*Pointer to an integer variable*/  
double *p2 /*Pointer to a variable of data type double*/  
char *p3 /*Pointer to a character variable*/  
float *p4 /*pointer to a float variable*/
```

The above are the few examples of pointer declarations. **If you need a pointer to store the address of integer variable then the data type of the pointer should be int.** Same case is with the other data types.

By using * operator we can access the value of a variable through a pointer. For example:

```
double a = 10;  
double *p;  
p = &a;
```

*p would give us the value of the variable a. The following statement would display 10 as output.

```
printf("%d", *p);
```

Similarly if we assign a value to *pointer like this:

```
*p = 200;
```

It would change the value of variable a. The statement above will change the value of a from 10 to 200.

Example of Pointer demonstrating the use of & and *

```
#include <stdio.h>
int main()
{
    /* Pointer of integer type, this can hold the
     * address of a integer type variable.
     */
    int *p;

    int var = 10;

    /* Assigning the address of variable var to the pointer
     * p. The p can hold the address of var because var is
     * an integer type variable.
     */
    p= &var;

    printf("Value of variable var is: %d", var);
    printf("\nValue of variable var is: %d", *p);
    printf("\nAddress of variable var is: %p", &var);
    printf("\nAddress of variable var is: %p", p);
    printf("\nAddress of pointer p is: %p", &p);
    return 0;
}
```

Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```

C - Pointers

```
int var = 10;
int *p;
p = &var;
```

**P is a pointer that stores the address of variable var.
The data type of pointer p and variable var should match because an integer pointer can only hold the address of integer variable.**

Lets take few more examples to understand it better –
Lets say we have a char variable ch and a pointer ptr that holds the address of ch.

```
char ch='a';
char *ptr;
```

Read the value of ch

```
printf("Value of ch: %c", ch);
or
printf("Value of ch: %c", *ptr);
```

Change the value of ch

```
ch = 'b';
or
*ptr = 'b';
```

The above code would replace the value 'a' with 'b'.

Can you guess the output of following C program?

```
#include <stdio.h>
int main()
```

```

{
int var =10;
int *p;
p= &var;

printf ( "Address of var is: %p", &var);
printf ( "\nAddress of var is: %p", p);

printf ( "\nValue of var is: %d", var);
printf ( "\nValue of var is: %d", *p);
printf ( "\nValue of var is: %d", *( &var));

/* Note I have used %p for p's value as it represents an address*/
printf( "\nValue of pointer p is: %p", p);
printf ( "\nAddress of pointer p is: %p", &p);

return 0;
}

```

Output:

```

Address of var is: 0x7fff5d027c58
Address of var is: 0x7fff5d027c58
Value of var is: 10
Value of var is: 10
Value of var is: 10
Value of pointer p is: 0x7fff5d027c58
Address of pointer p is: 0x7fff5d027c50

```

More Topics on Pointers

1) **Pointer to Pointer** – A pointer can point to another pointer (which means it can store the address of another pointer), such pointers are known as double pointer OR pointer to pointer.

2) **Passing pointers to function** – Pointers can also be passed as an argument to a function, using this feature a function can be called by reference as well as an array can be passed to a function while calling.

3) **Function pointers** – A function pointer is just like another pointer, it is used for storing the address of a function. Function pointer can also be used for calling a function in C program.

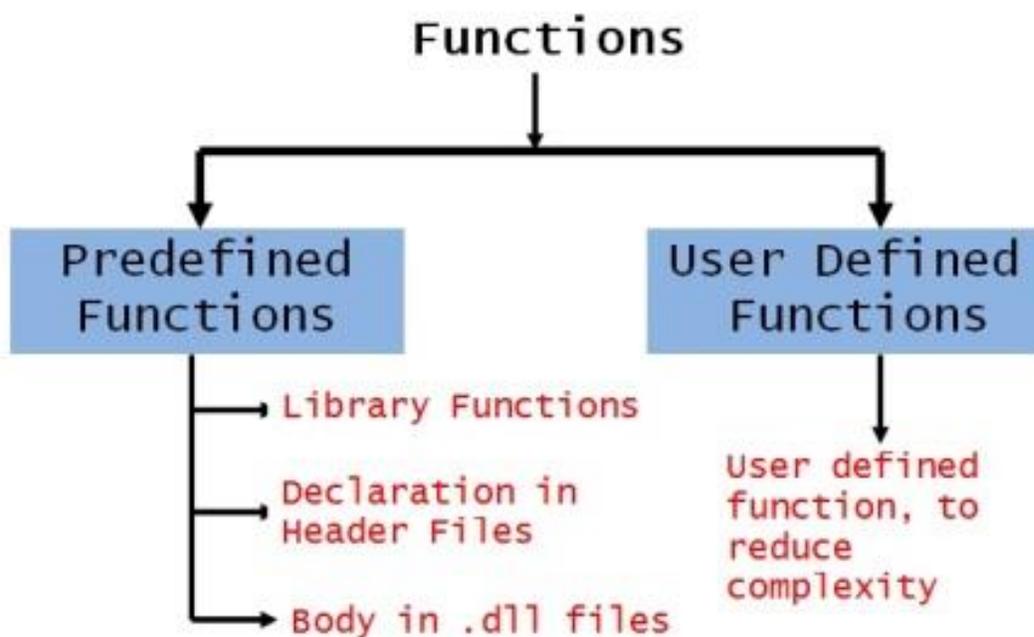
Function as building blocks

Functions in C Programming are building block of a Program. A function is a block of code that performs a specific task. There are times when we need to write particular block of code more than once in our program. This may increase complexity of a program. To make programmers life easy C Programming support functions which allows programmer to declare and define a group of statements once and that can be called and used whenever required.

Introduction of Functions in C

Dividing complex problem into small components makes program easy to understand and use. There are two types of functions Standard Library Functions and User Defined Functions.

Types of Functions in C



www.binaryupdates.com

Library Functions

The functions which are defined by C library example `printf()`, `scanf()`, `strcat()` etc. You

just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

User-defined Functions

The functions which are defined by the user at the time of writing program. Functions are made for code reusability and for saving time and space. Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype gives basic structural information: it tells the compiler what the function will return, when the function will be called, as well as what arguments the function can be passed. When I say that the function returns a value, I mean that the function can be used in the same manner as a variable would be.

Example: Simple Functions

Let's have a look at simple example and then we will break down program to understand.

```
#include <stdio.h>

int add(int x, int y); // function prototype

int main()
{
    int x;
    int y;

    printf("Please input two numbers: ");
    scanf("%d", &x);
    scanf("%d", &y);

    printf("The Addition is: %d\n", add(x, y));
}

int add(int x, int y) // function definition
{
```

```
return x + y;
}
```

Explanation: There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void.

In this example, `int add (int x, int y);` is function prototype. This prototype specifies that the function `add` will accept two arguments, both integers, and that it will return an integer. Do not forget semi-colon while writing function prototype. Without it, the compiler will probably think that you are trying to write the actual definition of the function. Now let's look at actual function definition:

```
int add (int x, int y) // function definition
{
    return x + y;
}
```

The `add` function takes two integer numbers in terms of `x` and `y`. `return` is keyword used to force the function to return an integer value and so we have defined function `add` as type integer. In main routine most of statements are self explanatory but still we'll look at:

```
printf("The Addition is: %d\n", add(x, y));
```

In this statement we have called `add` function by calling it into `printf` statement. The beauty of function is we can use `add` function as many time as we want in the program. So we need not to write logic all over again. This certainly allows us to reuse code.

Why Need Functions?

The function can be used for many reasons, say for example programmer writes a block of code that he wants to use 20-30 times throughout the program. A function to execute that code would save lots of space and time, at same time makes program more readable. As we define function only once, which makes it easy to modify at one place and change reflects all over the places. The functions in C programs also allowed us to break down complex program into logical parts. For example, take calculator program that runs a complex code with variety of different functions like addition, subtraction, multiplication, division and so on. The best way is to break down complex calculator program into smaller and manageable tasks in the form of function. Each task like

addition, subtraction, multiplication and division has been defined in its own function which makes a sense while reading and modifying code.

There are two ways that a C function can be called from a program. They are,

- **Call by Value**
- **Call by Reference**

Call by Value in C Programming

In call by value method, the value of the variable is passed to the function as parameter. The value of the actual parameter can't be modified by formal parameter. Different Memory is allocated for both actual and formal parameters, because value of actual parameter is copied to formal parameter. Note:

- **Actual parameter** – This is the argument which is used in function call
- **Formal parameter** – This is the argument which is used in function definition

Example: Function using call by value

```
#include <stdio.h>

int swap(int a, int b );

int main()
{
    int m=10, n=20;

    printf("Values before swap: m=%d and n=%d", m, n);

    swap(m, n); // calling swap function by value
}

int swap(int a, int b )
{
    int temp;

    temp = a;

    a = b;
```

```
b = temp;

printf("\nValues after swap: m=%d and n=%d\n", a, b);

}
```

Explanation: In this program, the values of the variables “m” and “n” are passed to the function “swap”. These values are copied to formal parameters “a” and “b” in swap function and used.

```
CAUsers\Umesh\Desktop\test\test\bin\Debug\test.exe
Values before swap: m=10 and n=20
Values after swap: m=20 and n=10
Process returned 0 (0x0) execution time : 0.842 s
Press any key to continue.
```

Call by Reference in C Programming

In call by reference method, the address of the variable is passed to the function as parameter. The value of the actual parameter can be modified by formal parameter. Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example: Function using Call by Reference

```
#include <stdio.h>

int swap(int *a, int *b );

int main()

{

    int m=10, n=20;

    printf("Values before swap: m=%d and n=%d", m, n);
```

```

    swap(&m, &n); // calling swap function by passing address
}

int swap(int *a, int *b )
{
    int temp;

    temp = *a;

    *a = *b;

    *b = temp;

    printf("\nValues after swap: m=%d and n=%d\n", *a, *b);
}

```

Explanation: In above program, the address of the variables “m” and “n” are passed to the function “swap”. These values are not copied to formal parameters “a” and “b” in swap function, because they are just holding the address of those variables. This address is used to access and change the values of the variables.

```

C:\Users\Umesh\Desktop\test\test\bin\Debug\test.exe
Values before swap: m=18 and n=28
Values after swap: m=28 and n=18
Process returned 0 (0x0) execution time : 0.811 s
Press any key to continue.

```

Output of Call by Reference Program

This may be difficult to understand for newbie’s if you’re not familiar with pointers in C language. We’ll cover pointers in our future lesson. So don’t worry if you feel difficult to understand for this moment.

Now we know how to create user defined Functions in C Programming. In next post, we'll learn about Recursion in C Programming. Please do write us if you have any suggestion/comment or come across any error on this page. Thanks for reading!

Once you start to write more complex programs, you will quickly find the need to perform some tasks and actions more than once during the course of a program.

This need is addressed by functions, which are similar to methods but are not attached to any particular object. As a programmer, you can create numerous functions in your programs-this helps organize the structure of your applications and makes maintaining and changing your program code easier.

In addition, functions are particularly useful in working with events and event handlers as you will learn in Chapter 5, "Events in JavaScript."

You can also use functions as the basis for creating your own objects to supplement those available to you in JavaScript.

In this chapter we will cover these topics:

- The nature of functions
- Built-in functions versus programmer-created functions
- How to define and use functions
- How to create new objects, properties, and methods
- How to use associative arrays

What Are Functions?

Functions offer the ability for programmers to group together program code that performs a specific task-or function-into a single unit that can be used repeatedly throughout a program.

Like the methods you have seen in earlier chapters, a function is defined by name and is invoked by using its name.

Also, like some of the methods you have seen before (such as `prompt()` and `confirm()`), functions can accept information in the form of arguments and can return results.

JavaScript includes several built-in functions as well as methods of base objects. You have already seen these when you used `alert()`, `document.write()`, `parseInt()`, or any of the other methods and functions you have been working with. The flexibility of JavaScript, though, lies in the ability for programmers to create their own functions to supplement those available in the JavaScript specification.

Using Functions

In order to make use of functions, you need to know how to define them, pass arguments to them, and return the results of their calculations. It is also important to understand the concept of variable scope, which governs whether a variable is available in an entire script or just within a specific function.

Defining Functions

Functions are defined using the function statement. The function statement requires a name for the function, a list of parameters-or arguments-that will be passed to the function, and a command block that defines what the function does:

```
function function_name(parameters, arguments) {  
    command block  
}
```

As you will notice, the naming of functions follows basically the same rules as variables: They are case sensitive, can include underscores (_), and start with a letter. The list of arguments passed to the function appears in parentheses and is separated by commas.

It is important to realize that defining a function does not execute the commands that make up the function. It is only when the function is called by name somewhere else in the script that the function is executed.

Passing Parameters

In the following function, you can see that `printName()` accepts one argument called `name`:

```
function printName(name) {  
    document.write("<HR>Your Name is <B><I>");  
    document.write(name);  
    document.write("</B></I><HR>");  
}
```

Within the function, references to `name` refer to the value passed to the function.

There are several points here to note:

- Both variables and literals can be passed as arguments when calling a function.
- If a variable is passed to the function, changing the value of the parameter within the function does not change the value of the variable passed to the function.
- Parameters exist only for the life of the function-if you call the function several times, the parameters are created afresh each time you call the function, and values they held when the function last ended are not retained.

For example, if you call `printName()` with the command:

```
printName("Bob");
```

then, when `printName()` executes, the value of `name` is "Bob". If you call `printName()` by using a variable for an argument:

```
var user = "John";  
printName(user);
```

then `name` has the value "John". If you were to add a line to `printName()` changing the value of `name`:

```
name = "Mr. " + name;
```

`name` would change, but the variable `user`, which was sent as an argument, would not change.

Note

When passing arguments to a function, two properties that can be useful in working with the arguments are created: `functionname.arguments` and `functionname.arguments.length`. `functionname.arguments` is an array with an entry for each argument and `functionname.arguments.length` is an integer variable indicating the number of variables passed to the function. You can use these properties to produce functions that accept a variable number of arguments.

Variable Scope

This leads to a discussion of variable scope. Variable scope refers to where a variable exists.

For instance, in the example `printName()`, `name` exists only within the function `printName()`-it cannot be referred to or manipulated outside the function. It comes into existence when the function is called and ceases to exist when the function ends. If the function is called again, a new instance of `name` is created.

In addition, any variable declared using the `var` command within the function will have a scope limited to the function.

If a variable is declared outside the body of a function, it is available throughout a script- inside all functions and elsewhere.



Variables declared within a function are known as local variables. Variables declared outside functions and available throughout the script are known as global variables.

If you declare a local variable inside a function that has the same name as an existing global variable, then inside the function, that variable name refers to the new local variable and not the global variable. If you change the value of the variable inside the function, it does not affect the value of the global variable.

Returning Results

As mentioned in the previous section, functions can return results. Results are returned using the return statement. The return statement can be used to return any valid expression that evaluates to a single value. For example, in the function cube(),

```
function cube(number) {  
  var cube = number * number * number;  
  return cube;  
}
```

the return statement will return the value of the variable cube. This function could just as easily have been written like this:

```
function cube(number) {  
  return number * number * number;  
}
```

This works because the expression number * number * number evaluates to a single value.

Functions in the File Header

As was mentioned in Chapter 3, "Working with Data and Information," there are compelling reasons to include function definitions inside the HEAD tags of the HTML file.

This ensures that all functions have been parsed before it is possible for user events to invoke a function. This is especially relevant once you begin working with event handlers where incorrect placement of a function definition can mean an event can lead to a function call when the function has not been evaluated and Navigator doesn't know it exists. When this happens, it causes an error message to be displayed.

The term <code>parsed</code> refers to the process by which the JavaScript interpreter evaluates each line of script code and converts it into a pseudo-compiled Byte Code (much like Java), before attempting to execute it. At this time, syntax errors and other programming mistakes that would prevent the script from running may be caught and reported to the user or programmer.

The type of Operator

JavaScript offers an operator that we didn't discuss in the last chapter when we looked at variables, expressions, and operators. The `typeof` operator is used to identify the type of an element in JavaScript. Given an unevaluated operand (such as a variable name or a function name), the `typeof` operator returns a string identifying the type of the operand.

For instance, suppose you have the following JavaScript code:

```
var question="What is 10 x 10?";
var answer=10;
var correct=false;
function showResult(results) {
    document.write(results);
}
```

Then, `typeof question` returns `string`; `typeof answer` returns `number`; `typeof correct` returns `boolean`; and `typeof showResult` returns `function`. Other possible results returned by the `typeof` operator include `undefined` and `object`.

Note
The <code>typeof</code> operator can be useful in determining if a function has been loaded and is ready to be called. This is especially useful in multi-frame pages where a script loaded in one frame might be trying to call a function located in another frame that hasn't completed loading.

Frames are covered in more detail in Chapter 8.

Putting Functions to Work

To demonstrate the use of functions, you are going to rewrite the simple test question example you used in Listing 3.3. In order to do this, you are going to create a function that receives a question as an argument, poses the question, checks the answer, and returns an output string based on the accuracy of the user's response as shown in Listing 4.1

In order to do this, you need to learn the `eval()` method, which evaluates a string to a numeric value; for instance,

```
eval("10*10")
```

returns a numeric value of 100.

Listing 4.1. Evaluating an expression with the `eval()` function.

```
<HTML>

<HEAD>
<TITLE>Example 4.1</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//DEFINE FUNCTION testQuestion()
function testQuestion(question) {
  //DEFINE LOCAL VARIABLES FOR THE FUNCTION
  var answer=eval(question);
  var output="What is " + question + "?";
  var correct='<IMG SRC="correct.gif">';
  var incorrect='<IMG SRC="incorrect.gif">';

  //ASK THE QUESTION
  var response=prompt(output,"0");

  //chECK THE RESULT
  return (response == answer) ? correct : incorrect;
}

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>
```

```
</HEAD<

<BODY>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//ASK QUESTION AND OUTPUT RESULTS
var result=testQuestion("10 + 10");
document.write(result);

//STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

</BODY>

</HTML>
```

At first glance, this script may seem a little more complicated than the version used in Listing 3.3. In reality, though, it simply separates the work into logical blocks and moves most of the work into the function testQuestion().

To understand the function, let's analyze the key lines.

```
function testQuestion(question) {
```

In this line, you define the function testQuestion() and indicate that it receives one argument, which is referred to as question within the function. In the case of this function, it is expected that question will be a string containing an arithmetic expression.

```
var answer=eval(question);
```

The first thing you do after entering the function is to declare the variable answer and assign to it the numeric value of the arithmetic expression contained in the string question. This is achieved using the eval() function.

```
var output="What is " + question + "?";  
var correct='<IMG SRC="correct.gif">';  
var incorrect='<IMG SRC="incorrect.gif">';
```

In these lines you declare several more variables. The variable output contains the actual question to display, which is created using the concatenation operator.

```
var response=prompt(output,"0");
```

Here you ask the question and assign the user's response to the variable response.

```
return (response == answer) ? correct : incorrect;
```

In this line you use the conditional operator to check the user's response. The resulting value is returned by the return command.

Now that you understand the function, it should be clear how you are invoking it later in the body of the HTML file. The line

```
var result=testQuestion("10 + 10");
```

calls testQuestion() and passes a string to it containing an arithmetic expression. The function returns a result, which is stored in the variable result. Then you are able to output the result using document.write().

These two lines could be condensed into a single line:

```
document.write(testQuestion("10 + 10"));
```

Recursive Functions

Now that you have seen an example of how functions work, let's take a look at an application of functions called recursion.

Recursion refers to situations in which functions call themselves. These types of functions are known as recursive functions.

For instance, the following is an example of a recursive function that calculates a factorial:

Note

A factorial is a mathematical function. For example, factorial 5 (written 5!) is equal to $5 \times 4 \times 3 \times 2 \times 1$ and $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.

```
function factorial(number) {  
  if (number > 1) {  
    return number * factorial(number - 1);  
  } else {  
    return number;  
  }  
}
```

At first glance, this function may seem strange. This function relies on the fact that the factorial of a number is equal to the number multiplied by the factorial of one less than the number. Expressed mathematically, this could be written:

$$x! = x * (x-1)!$$

In order to apply this formula, you have created a recursive function called factorial(). The function receives a number as an argument. Using the following if-else construct:

```
if (number > 1) {  
  return number * factorial(number - 1);  
} else {  
  return number;  
}
```

The function either returns a value of 1 if the argument is equal to 1 or applies the formula and returns the number multiplied by the factorial of one less than the number.

In order to do this, it must call the function factorial() from within the function factorial(). This is where the concept of variable scope becomes extremely important. It is important to realize that when the function calls factorial(), a new instance of the function is being invoked, which means that a new instance of number is created. This continues to occur until the expression number-1 has a value of 1.

Tip

Recursive functions are powerful, but they can be dangerous if you don't watch out for infinite recursion. Infinite recursion occurs when

the function is designed in such a way as to call itself forever without stopping.

At a practical level, in JavaScript, infinite recursion isn't likely to happen because of the way in which JavaScript handles some of its memory allocation. This means that deep recursions, even if they aren't infinite, may cause Navigator to crash.

It is important to note that the function factorial() prevents infinite recursion because the if-else construct ensures that eventually the function will stop calling itself once the number passed to it is equal to one. In addition, if the function is initially called with a value less than two, the function will immediately return without any recursion.

Using recursive functions, it is possible to extend the program used in Listing 4.1 so that it continues to ask the question until the user provides the correct answer, as shown in Listing 4.2.

Listing 4.2. Using a recursive function to repeat input.

```
<HTML>

<HEAD>
<TITLE>Example 4.2</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//DEFINE FUNCTION testQuestion()
function testQuestion(question) {
  //DEFINE LOCAL VARIABLES FOR THE FUNCTION
  var answer=eval(question);
  var output="What is " + question + "?";
  var correct='<IMG SRC="correct.gif">';
  var incorrect='<IMG SRC="incorrect.gif">';

  //ASK THE QUESTION
  var response=prompt(output,"0");
```

```
//CHECK THE RESULT
return (response == answer) ? correct : testQuestion(question);
}
```

```
// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>
```

```
</HEAD<
```

```
<BODY>
```

```
<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS
```

```
//ASK QUESTION AND OUTPUT RESULTS
var result=testQuestion("10 + 10");
document.write(result);
```

```
//STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>
```

```
</BODY>
```

```
</HTML>
```

Notice that you have made only a single change to the conditional expression:

```
return (response == answer) ? correct : testQuestion(question);
```

Where you originally returned the value of the variable incorrect when the user provided an incorrect response, you are now returning the result of asking the question again (by calling testQuestion() again).

It is important to realize that this example could cause JavaScript to crash because of its memory handling problems if the user never provides the correct answer. This can be remedied by adding a counter to keep track of the number of chances the user has to provide a correct answer:

```

<HTML>

<HEAD>
<TITLE>Example 4.2</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//DEFINE FUNCTION testQuestion()
function testQuestion(question,chances) {
  //DEFINE LOCAL VARIABLES FOR THE FUNCTION
  var answer=eval(question);
  var output="What is " + question + "?";
  var correct='<IMG SRC="correct.gif">';
  var incorrect='<IMG SRC="incorrect.gif">';

  //ASK THE QUESTION
  var response=prompt(output,"0");

  //CHECK THE RESULT
  if (chances > 1) {
    return (response == answer) ? correct : testQuestion(question,chances-1);
  } else {
    return (response == answer) ? correct : incorrect;
  }
}

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

</HEAD>

<BODY>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//ASK QUESTION AND OUTPUT RESULTS
var result=testQuestion("10 + 10",3);
document.write(result);

```

```
//STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

</BODY>

</HTML>
```

By adding the if-else construct when you check the user's answer, you are ensuring that you cannot enter an infinite recursion. The if-else construct could be replaced by a conditional expression:

```
return (response == answer) ? correct : ((chances > 1) ?
testQuestion(question,chances-1) : incorrect);
```

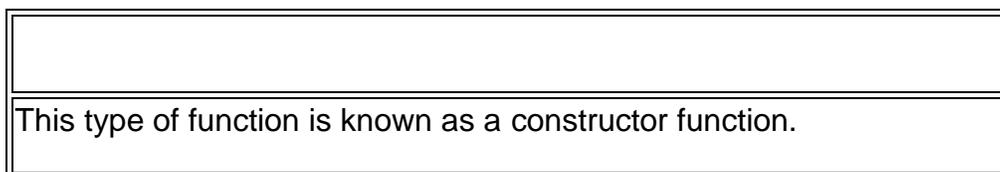
What this expression says is, if the user's response is correct (response==answer evaluates to true), then return the value of correct. Otherwise, if there are chances left (chances > 1 evaluates to true), ask the question again and return the result. If there are no chances left and the answer is incorrect, return the value of the variable incorrect.

Building Objects in JavaScript

As you learned earlier, it is possible to use functions to build custom objects in JavaScript. In order to do this, you must be able to define an object's properties, to create new instances of objects, and to add methods to objects.

Defining an Object's Properties

Before creating a new object, it is necessary to define that object by outlining its properties. This is done by using a function that defines the name and properties of the function.



If you want to create an object type for students in a class, you could create an object named student with properties for name, age, and grade. This could be done with the function:

```
function student(name,age, grade) {
  this.name = name;
```

```
this.age = age;
this.grade = grade;
}
```

Note

Notice the use of the special keyword `this`. `this` plays a special role in JavaScript and refers to the current object. You will learn more about this in Chapter 5, when we begin discussing event handlers.

Using this function, it is now possible to create an object using the `new` statement:

```
student1 = new student("Bob",10,75);
```

This line of JavaScript code creates an object called `student1` with three properties: `student1.name`, `student1.age`, and `student1.grade`. This is known as an instance of the object `student`. By creating a new object `student2` using the `new` statement,

```
student2 = new student("Jane",9,82);
```

you would be creating a new instance of the object that is independent from `student1`.

It is also possible to add properties to objects once they are created simply by assigning values to a new property. For instance, if you want to add a property containing Bob's mother's name, you could use the structure

```
student1.mother = "Susan";
```

This would add the property to `student1` but would have no effect on `student2` or future instances of the `student` object. To add the property `mother` to all instances of `student`, it would be necessary to add the property to the object definition before creating instances of the object:

```
function student(name, age, grade, mother) {
  this.name = name;
  this.age = age;
  this.grade = grade;
  this.mother = mother;
}
```

Objects as Properties of Objects

You can also use objects as properties of other objects. For instance, if you were to create an object called grade

```
function grade (math, english, science) {  
  this.math = math;  
  this.english = english;  
  this.science = science;  
}
```

you could then create two instances of the grade object for the two students:

```
bobGrade = new grade(75,80,77);  
janeGrade = new grade(82,88,75);
```

Note
The order of arguments is important in JavaScript. In the preceding example, if Jane hasn't taken English, you would need to pass a place-holder to the function, such as zero or a string value, such as "N/A" or the empty string. The function would then need to be written to handle this eventuality.

Using these objects, you could then create the student objects like this:

```
student1 = new student("Bob",10,bobGrade);  
student2 = new student("Jane",9,janeGrade);
```

You could then refer to Bob's math grade as `student1.grade.math` or Jane's science grade as `student2.grade.science`.

Adding Methods to Objects

In addition to adding properties to object definitions, you can also add a method to an object definition. Because methods are essentially functions associated with an object, first you need to create a function that defines the method you want to add to your object definition.

For instance, if you want to add a method to the student object to print the student's name, age, and grades to the document window, you could create a function called `displayProfile()`:

```

function displayProfile() {
  document.write("Name: " + this.name + "<BR>");
  document.write("Age: " + this.age + "<BR>");
  document.write("Mother's Name: " + this.mother + "<BR>");
  document.write("Math Grade: " + this.grade.math + "<BR>");
  document.write("English Grade: " + this.grade.english + "<BR>");
  document.write("Science Grade: " + this.grade.science + "<BR>");
}

```

Note
Here again, you use this to refer to the object that is invoking the method. If you call a method as object1.method, then this refers to object1.

Having defined the method, you now need to change the object definition to include the method:

```

function student(name,age, grade) {
  this.name = name;
  this.age = age;
  this.grade = grade;
  this.mother = mother;
  this.displayProfile = displayProfile;
}

```

Then, you could output Bob's student profile by using the command:

```
student1.displayProfile();
```

This would produce results similar to those in Figure 4.1.

Figure 4.1 : The display. Profile() method displays the profile for any instance of the student object.

Extending Objects Dynamically

Starting with Navigator 3.0, it is possible to extend objects after they have been created with a new statement.

Properties can be added to object definitions by setting the value of objectName.prototype.propertyName. objectName refers to the name of the

constructor function, and `propertyName` is the name of the property or method being added to the function.

For instance, if you want to add an additional method called `updateProfile()` to the student object definition you created earlier, you could use the command:

```
student.prototype.updateProfile = updateInfo;
```

where you have already created a function called `updateInfo()`:

```
function updateInfo() {  
    this.age = prompt("Enter the correct age for " + this.name,this.age);  
    this.mother = prompt("Enter the mother's name for " + this.name,this.mother);  
}
```

Then, all instances of `student` that had previously been created using the new statement would be able to use the new method.

In the example we used above, you could update the age and mother's name for Bob by using this command:

```
student1.updateProfile();
```

Defining Your Own Objects

To further demonstrate the application of objects and defining your own objects, Listing 4.3 is a program that asks the user for personnel information of an employee and then formats it for display on the screen.

In order to do this, you need to define an employee object, as well as a method for displaying the employee information.

Listing 4.3. Creating an employee profile.

```
<HTML>
```

```
<HEAD>
```

```
<TITLE>Example 4.3</TITLE>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
<!-- HIDE FROM OTHER BROWSERS
```

```
//DEFINE METHOD
```

```
function displayInfo() {
```

```

document.write("<H1>Employee Profile: " + this.name + "</H1><HR><PRE>");
document.writeln("Employee Number: " + this.number);
document.writeln("Social Security Number: " + this.socsec);
document.writeln("Annual Salary: " + this.salary);
document.write("</PRE>");
}

//DEFINE OBJECT
function employee() {
  this.name=prompt("Enter Employee's Name","Name");
  this.number=prompt("Enter Employee Number for " + this.name,"000-000");
  this.socsec=prompt("Enter Social Security Number for " +
this.name,"000-00-0000");
  this.salary=prompt("Enter Annual Salary for " + this.name,"$00,000");
  this.displayInfo=displayInfo;
}

newEmployee=new employee();

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

</HEAD>

<BODY>
<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

newEmployee.displayInfo();

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

</BODY>

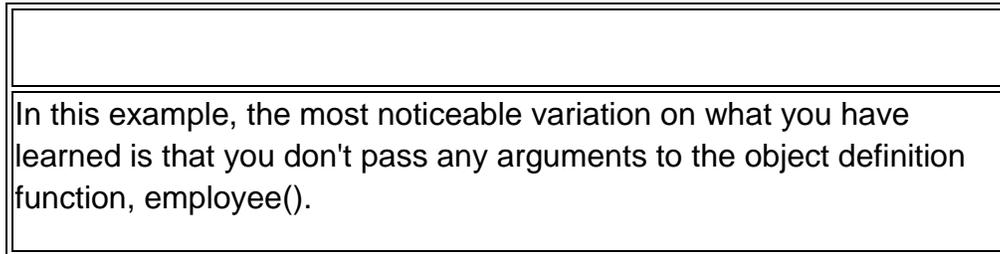
</HTML>

```

This script produces results similar to those in Figure 4.2 and 4.3.

Figure 4.2 :The program prompts the user for the employee information.

Figure 4.3 : The method you defined displays the formatted data.



Instead, this object definition is more of a dynamic object definition in that, when a new instance of the object is created, the user is prompted for all the relevant data for the properties.

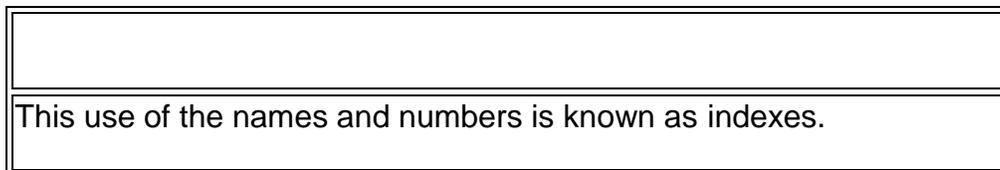
Properties as Indexes

In Navigator 2.0, it was possible to refer to object properties numerically in two ways other than `objectName.propertyName`. That is, in the previous example of the student object, the properties of `student2` could have been referred to as:

```
student2["name"]  
student2["age"]
```

and so on, or by numbers starting at zero, where

```
student2[0] == "Jane"  
student2[1] == 9
```



In Navigator 3.0, it is no longer possible to refer to a property both by numeric index and by name. Instead, properties can either be created as numeric indexes or names but then all subsequent references have to be the same. Numbers and names cannot be used inter-changeably.

Arrays

Anyone who has programmed in other structured languages has probably encountered arrays of one sort or another and will be wondering where JavaScript's arrays are.

Arrays are ordered collections of values referred to by a single variable name. For instance, if you have an array named student, you might have the following ordered values:

```
student[0] = "Bob"  
student[1] = 10  
student[2] = 75
```

Array elements are referred to by their indexes-the numbers in brackets. In JavaScript, arrays start with index zero.

Arrays in JavaScript are created using the Array() constructor object. You can create an array of undefined length by using the new keyword:

```
arrayName = new Array();
```

The length of the array changes dynamically as you assign values to the array. The length of the array is defined by the largest index number used.

For instance

```
var sampleArray = new Array();  
sampleArray[49] = "50th Element";
```

creates an array with 50 elements. (Remember, indexes start at zero.)

It is also possible to define an initial length of an array by passing the length to the Array() object as an argument:

```
var sampleArray = new Array(100);
```

In addition, you can create an array and assign values to all its elements at the time it is defined. This is done by passing the value for all elements as arguments to the Array() object. For instance,

```
var sampleArray = new Array("1st Element", 2, "3rd Element");
```

creates a three-element array with the values

```
sampleArray[0] == "1st Element"  
sampleArray[1] == 2  
sampleArray[2] == "3rd Element"
```

As objects, arrays have several methods, including

- `join()` returns all elements of the array joined together as a single string. This takes one argument: a string used as the separator between each element of the array in the final string. If the argument is omitted, `join()` uses a comma-space as the separator.
- `reverse()` reverses the order of elements in the array.

To demonstrate how arrays can be useful, Listing 4.4 builds on the personnel information example in the Listing 4.3.

In this example, you do not have the user enter the personnel information for the new employee in the same way. You present a list of information you want. The users select a number for the information they want to enter. When they are done, they select "0."

After a user finishes entering the information, the script displays the formatted employee profile.

Listing 4.4. Creating a user menu.

```
<HTML>

<HEAD>
<TITLE>Listing 4.4</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

//DEFINE METHOD
function displayInfo() {
    document.write("<H1>Employee Profile: " + this.data[0] + "</H1><HR><PRE>");
    document.writeln("Employee Number: " + this.data[1]);
    document.writeln("Social Security Number: " + this.data[2]);
    document.writeln("Annual Salary: " + this.data[3]);
    document.write("</PRE>");
}

//DEFINE METHOD TO GET EMPLOYEE INFORMATION
function getInfo() {
    var menu="0-Exit/1-Name/2-Emp. #/3-Soc. Sec. #/4-Salary";
    var choice=prompt(menu,"0");
    if (choice != null) {
        if ((choice < 0) || (choice > 4)) {
            alert ("Invalid choice");
            this.getInfo();
        }
    }
}
```

```

    } else {
      if (choice != "0") {
        this.data[choice-1]=prompt("Enter information","");
        this.getInfo();
      }
    }
  }
}

```

```

//DEFINE OBJECT
function employee() {
  this.data = new Array(4);
  this.displayInfo=displayInfo;
  this.getInfo=getInfo;
}

```

```

newEmployee=new employee();

```

```

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

```

```

</HEAD>

```

```

<BODY>

```

```

<SCRIPT LANGUAGE="JavaScript">
<!-- HIDE FROM OTHER BROWSERS

```

```

newEmployee.getInfo();
newEmployee.displayInfo();

```

```

// STOP HIDING FROM OTHER BROWSERS -->
</SCRIPT>

```

```

</BODY>

```

```

</HTML>

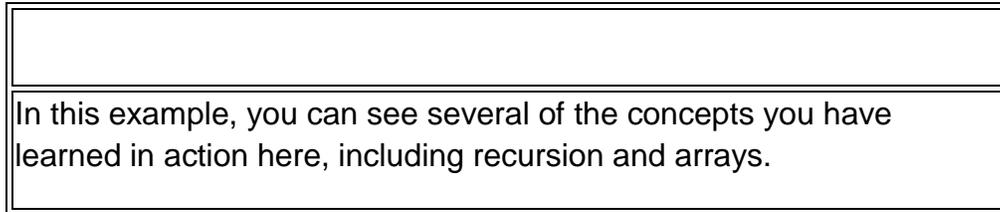
```

This script produces a series of results similar to those in Figures 4.4, 4.5, and 4.6.

Figure 4.4 : A menu in a prompt box.

Figure 4.5 : Prompting for input

Figure 4.6 : The final result.



The method `getInfo()` needs some explanation:

```
var menu="0-Exit/1-Name/2-Emp. #/3-Soc. Sec. #/4-Salary";
```

The menu variable contains the string that presents the choices to the user. Notice the use of the `\n` special character to create a multiline menu in a single text string.

```
var choice = prompt(menu,"0");
```

Here you present the menu to the user and ask for a choice, which is stored in the variable `choice`.

```
if (choice != null) {  
    if ((choice < 0) || (choice > 4)) {  
        alert ("Invalid choice");  
        this.getInfo();  
    } else {  
        if (choice != "0") {  
            this.data[choice-1]=prompt("Enter information","");  
            this.getInfo();  
        }  
    }  
}
```

This set of if statements is where the real work of the `getInfo()` method is done. The first if statement checks whether the user has selected Cancel. If not, then the user's choice is checked to make sure it is in range (from zero to four). If the choice is out of range, then the user is alerted, and the `getInfo()` method is called again. If the user's choice is in range, it is checked to see if the user has chosen 0 for exit. If the user doesn't select 0, the user is prompted to enter the data he has indicated. Then the `getInfo()` method is called again to present the menu again.

You will notice the use of the `this` keyword to refer to the current object and the use of `this.data [choice-1]` to refer to the array element (or property) selected by the user.

You use choice-1 because the menu presents choices from 1 to 4 to the user, but array indexes start from 0 and, in this case, the index goes up to 3.

Functions as Objects

Functions in JavaScript can be created as instances of the special Function object using the new keyword:

```
functionName = new Function(arglist, functionDefinition);
```

This provides an alternate way to create functions. In the example above, arglist is a comma-separated list of argument names, and functionDefinition is the JavaScript code to be executed each time the function is called. Each item in the argument list should be a string literal.

Functions defined using the Function object can be used in the same way as functions defined using the function keyword (including in event handlers) and can be called directly in expressions.

Instances of the Function object have two properties:

- arguments An array corresponding to the arguments of the function in the order they are defined.
- prototype As you learned earlier in this chapter, prototype can be used to add properties and methods to objects.

According to the Netscape documentation, creating functions using the Function object is the less efficient of the two methods of creating objects.

Summary

Functions provide a means to define segments of code that can be used more than once in a program. Like methods, which are part of objects, functions are defined by names, can be passed arguments, and can return results.

Variable scope, whether a variable exists locally to a function or globally for the entire program, is an important concept in dealing with functions.

Recursive functions are functions that call themselves one or more times. Recursion is a powerful tool, but it needs to be used carefully to avoid infinite recursion, which occurs when a function repeatedly calls itself without ever ending. With the current implementation of JavaScript, infinite recursion can't really happen because memory handling shortcomings mean that Navigator will crash when recursion gets too deep.

Functions are also used to define the properties and objects that make up user-defined methods. Using the new keyword, it is possible to create multiple instances of an object which all exist independently.

Arrays provide a mechanism to group together data into ordered collections. Index numbers provide a means to access individual elements in an array.

If you have made it to the end of this chapter, you are making progress because recursion, functions, and objects are advanced topics.

In Chapter 5, you begin to work with events and event handlers which will allow you to design programs that interact with the user in a sophisticated way.

Commands and Extensions Review

Command/Extension	Type	Description
function	JavaScript keyword	Declares a function
new	JavaScript keyword	Creates a new instance of an object
eval()	JavaScript method	Evaluates a string to a numeric value
this	JavaScript keyword	Refers to the current object
type of	JavaScript operator	Returns a string indicating the type of the operand

Exercises

1. Write the object definition for an object called car with four properties: model, make, year, and price.
2. If you have an object defined as follows

```
function house(address,rooms,owner) {  
  this.address = address;  
  this.rooms = rooms;  
  this.owner = owner;  
}
```

and you create two instances of the house object

```
house1 = new house("10 Maple St.",10,"John");  
house2 = new house("15 Sugar Rd.",12,"Mary");
```

then, what would be the value of the following:

- a. house1.rooms
 - b. house2.owner
 - c. house1["address"]
3. Create a function that calculates the value of x to the power of y. For instance, if you have a function called power() and you issue the command

```
value = power(10,4);
```

then power() should return the value of 10 to the power of 4, or $10 * 10 * 10 * 10$.

Note

If the notation x^y refers to x to the power of y, then it will be helpful in writing this function to realize that $x^y = x * x^{(y-1)}$.

Answers

1. The object definition would look like this:

```
function car(model, make, year, price) {  
  this.model = model;  
  this.make = make;  
  this.year = year;  
  this.price = price;  
}
```

2. The values are as follows:

- a. 10
- b. "Mary"
- c. "10 Maple St."

3. The power() function could be written using recursion:

```
function power(number, exponent) {  
  if (exponent > 1) {  
    return number * power(number, exponent - 1);  
  }
```

```
} else {  
    return 1;  
}  
}
```

This function makes use of a similar principle as the factorial example earlier in the chapter. This function uses the fact that x to the power of y equals x multiplied by x to the power of $y-1$.

While this function works, it is important to note that it isn't perfect. Although negative exponents are mathematically legal, this function will not calculate the result correctly for this type of exponent.

Remembering that x to the power of $-y$ is the same as 1 divided by x to the power of y , you could fix the problem with negative exponents by making the following change to the function:

```
function power(number, exponent) {  
  
    // CHECK IF WE HAVE A NEGATIVE EXPONENT  
    var negative = (exponent < 0) ? true : false;  
  
    // DECLARE WORKING VARIABLE  
    var value=0;  
  
    // CALCULATE number TO THE POWER OF exponent  
    if (exponent > 1) {  
        value = number * power(number, exponent - 1);  
    } else {  
        value = 1;  
    }  
  
    // IF THE EXPONENT WAS NEGATIVE, TAKE THE RECIPROCAL  
    if (negative)  
        value = 1 / value;  
  
    return value;  
}
```

Note

JavaScript includes a method that performs the same operation as your <code>power()</code> function. The <code>Math.pow()</code> method is part of the <code>Math</code> object and is discussed in Chapter 10, "Strings, Math, and the History List."
--

Language Fundamentals

Character set

The basic elements used to construct a simple C program are: the C character set, identifiers and keywords, data types, constants, arrays, declarations, expressions and statements. Let us see how these elements can be combined to form more comprehensive program components-

The C Character Set:

C uses letters A to Z in lowercase and uppercase, the digits 0 to 9, certain special characters, and white spaces to form basic program elements (e.g variables, constants, expressions etc.)

The special characters are:

+ - * / = % & # ! ? ^ " ' / | < > () [] { } ; : , ~ @ !

The white spaces used in C programs are: blank space, horizontal tab, carriage return, new line and form feed.

Identifiers and Keywords:

Identifiers are names given to various program elements such as variables, functions, and arrays. Identifiers consist of letters and digits, in any order, except that the first character must be a letter. Both uppercase and lowercase letters are permitted and the underscore may also be used, as it is also regarded as a letter. Uppercase and lowercase letters are not equivalent, thus not interchangeable. This is why it is said that C is case sensitive. An identifier can be arbitrarily long.

The same identifier may denote different entities in the same program, for example, a variable and an array may be denoted by the same identifier, example below.

```
int sum, average, A[10]; // sum, average and the array name A are all identifiers.
```

The `__func__` predefined identifier:-

The predefined identifier `__func__` makes a function name available for use within the function. Immediately following the opening brace of each function definition, `__func__` is implicitly declared by the compiler in the following way:

```
static const char __func__[ ] = "function-name";
```

where function-name is the name of the function.

consider the following example

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

```
void func1(void) {  
    printf("%sn", __func__);  
    return;  
}
```

```
int main() {  
    myfunc();  
}
```

The output would be

func1

Keywords

Keywords are reserved words that have standard predefined meanings. These keywords can only be used for their intended purpose; they cannot be used as programmer defined identifiers. Examples of some keywords are: int, main, void, if.

Data Types

Data values passed in a program may be of different types. Each of these data types are represented differently within the computer's memory and have different memory requirements. These data types can be augmented by the use of data type qualifiers/modifiers.

The data types supported in C are described below:

int:

It is used to store an integer quantity. An ordinary int can store a range of values from INT_MIN to INT_MAX as defined by in header file <limits.h>. The **type modifiers** for the int data type are: signed, unsigned, short, long and long long.

- A short int occupies 2 bytes of space and a long int occupies 4 bytes.
- A short unsigned int occupies 2 bytes of space but it can store only positive values in the range of 0 to 65535.

- An unsigned int has the same memory requirements as a short unsigned int. However, in case of an ordinary int, the leftmost bit is reserved for the sign.
- A long unsigned int occupies 4 bytes of memory and stores positive integers in the range of 0 to 4294967295.
- By default the int data type is signed.
- A long long int occupies 64 bits of memory. It may be signed or unsigned. The signed long long int stores values from $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$ and the unsigned long long ranges from 0 to $18,446,744,073,709,551,615$.

char:

It stores a single character of data belonging to the C character set. It occupies 1 byte of memory, and stores any value from the C character set. The type modifiers for char are signed and unsigned.

Both signed and unsigned char occupy 1 byte of memory but the range of values differ. An unsigned char can store values from 0 to 255 and a signed char can store values from -128 to +127. Each char type has an equivalent integer interpretation, so that a char is really a special kind of short integer. By default, char is unsigned.

float:

It is used to store real numbers with single precision i.e. a precision of 6 digits after decimal point. It occupies 4 bytes of memory. The type modifier for float is long. It has the same memory requirements as double.

double:

It is used to store real numbers with double precision. It occupies 8 bytes of memory. The type modifier for double is long. A long double occupies 10 bytes of memory.

void:

It is used to specify an empty set containing no values. Hence, it occupies 0 bytes of memory.

_Bool:

A boolean data type, which is an unsigned integer type, that can store only two values, 0 and 1. Include the file `<stdbool.h>` when using `_Bool`.

_Complex:

It is used to store complex numbers. There are three complex types: `float _Complex`, `double _Complex`, and `long double _Complex` It is found in the `<complex.h>` file.

Arrays:

An array is an identifier that refers to a collection of data items of that have the same name. They must also have the same data type (i.e. all characters, all integers etc.). Each data item is represented by its array element. The individual array elements are distinguished from one another by their subscripts.

Syntax for array declaration:

Suppose `arr` is a 5 element integer array which stores the numbers 5, 4, 2, 7, 3 in that order. The first element is referred to as `arr[0]` which stores the data value 5, the second element is `arr[1]` which stores the value 4 and so on. The last element is `arr[4]` which stores the value 3. The subscript associated with each element is shown in square braces. For an n-element array the subscripts will range from 0 to n-1.

In case of a character array of size n, the array will be able to store only n-1 elements as a null character is automatically stored at the end of the string to terminate it. Therefore, a character array `letter` that stores 5 elements must be declared of size 6. The fifth element would be stored at `letter[4]` and `letter[5]` will store the null character.

Constants:

A constant is an identifier whose value remains unchanged throughout the program. To declare any constant the syntax is:
`const datatype varname = value;` where `const` is a keyword that declares the variable to be a fixed value entity.

There are four basic types of constants in C. They are integer constants, floating point constants, character constants and string constants. Integer and floating point constants cannot contain commas or blank spaces; but they can be prefixed by a minus sign to indicate a negative quantity.

Integer Constants:

An integer constant is an integer valued number. It consists of a sequence of digits. Integer constants can be written in the following three number systems:

Decimal(base 10): A decimal constant can consist of any combination of digits from 0 to 9. If it contains two or more digits, the first digit must be something other than 0, for example: `const int size =50;`

Octal(base 8): An octal constant can consist of any combination of digits from 0 to 7. The first digit must be a 0 to identify the constant as an octal number, for example: `const int a= 074;` `const int b= 0;`

Hexadecimal constant(base 16): A hexadecimal constant can consist of any combination of digits from 0 to 9 and a to f (either uppercase or lowercase). It must begin with 0x or 0X to identify the constant as a hexadecimal number, for example: `const int c= 0x7FF;`

Integer constants can also be prefixed by the type modifiers `unsigned` and `long`. Unsigned constants must end with `u` or `U`, long integer constants must end with `l` or `L` and unsigned long integer constants must end with `ul` or `UL`. Long long integer constants end with `LL` or `ll`. Unsigned long long end with `ULL` or `ull`

Floating Point Constant:

Its a base 10 or a base 16 number that contains a decimal point or an exponent or both. In case of a decimal floating point constant the exponent the base 10 is replaced by `e` or `E`. Thus, $1.4 * 10^{-3}$ would be written as `1.4E-3` or `1.4e-3`.

In case of a hexadecimal character constant, the exponent is in binary and is replaced by `p` or `P`. For example:

```
const float a= 5000. ;
```

```
const float b= .1212e12;
```

```
const float c= 827.54;
```

Floating point constants are generally double precision quantities that occupy 8 bytes. In some versions of C, the constant is appended by `F` to indicate single precision and by `L` to indicate a long floating point constant.

Character Constants:

A character constant is a sequence of one or more characters enclosed in apostrophes. Each character constant has an equivalent integer value that is determined by the computer's character set. It may also contain escape sequences. A character literal may be prefixed with the letter `L`, `u` or `U`, for example `L'c'`.

A character literal without the `L` prefix is an ordinary character constant or a narrow character constant. A character literal with the `L` prefix is a wide character constant. The type of a narrow character constant and a multicharacter constant is `int`. The type of a wide character constant with prefix `L` is `wchar_t` defined in the header file `<stddef.h>`. A wide character constant with prefix `u` or `U` is of type `char16_t` or `char32_t`. These are unsigned character types defined in `<uchar.h>`.

An ordinary character literal that contains more than one character or escape sequence is a multicharacter constant, for example: `const char p= 'A';`

Escape Sequences are also character constants that are used to express certain non printing characters such as the tab or the carriage return. An escape sequence always begins with a backward slash and is followed by one or more special characters. for eg. b will represent the bell, n will represent the line feed.

String Literals:

A string literal consists of a sequence of multibyte characters enclosed in double quotation marks. They are of two types, wide string literal and UTF-8 string literal. A UTF-8 string literal is prefixed by `u8` and a wide string literal by `L`, `u` or `U`.

The compiler recognizes and supports the additional characters (the extended character set) which you can meaningfully use in string literals and character constants. The support for extended characters includes the multibyte character sets. A multibyte character is a character whose bit representation fits into one or more bytes.

Symbolic Constants:

A symbolic constant is a name that substitutes for a numeric constant, a character constant or a string constant throughout the program. When the program is compiled each occurrence of a symbolic constant is replaced by its actual value.

A symbolic constant is defined at the beginning of a program using the **# define** feature. The **# define** feature is called a preprocessor directive, more about the C preprocessor in a later article.

A symbolic constant definition never ends with a semi colon as it is not a C statement rather it is a directive, for example:

```
#define PI 3.1415 //PI is the constant that will represent value 3.1415
#define True 1
#define name "Alice"
```

These were the some of the basic elements of C. Elements like scope of an identifier, declarations, statements, expressions, etc., will be explained in future articles with the help of practical examples. The next article will discuss the operators provided in the C language.

C Tokens, Keywords

Like every other language, 'C' also has its own character set. A program is a set of instructions that, when executed, generate an output. The data that is processed by a program consists of various characters and symbols. The output generated is also a combination of characters and symbols.

A character set in 'C' is divided into,

- Letters
- Numbers
- Special characters
- White spaces (blank spaces)

A compiler always ignores the use of characters, but it is widely used for formatting the data. Following is the character set in 'C' programming:

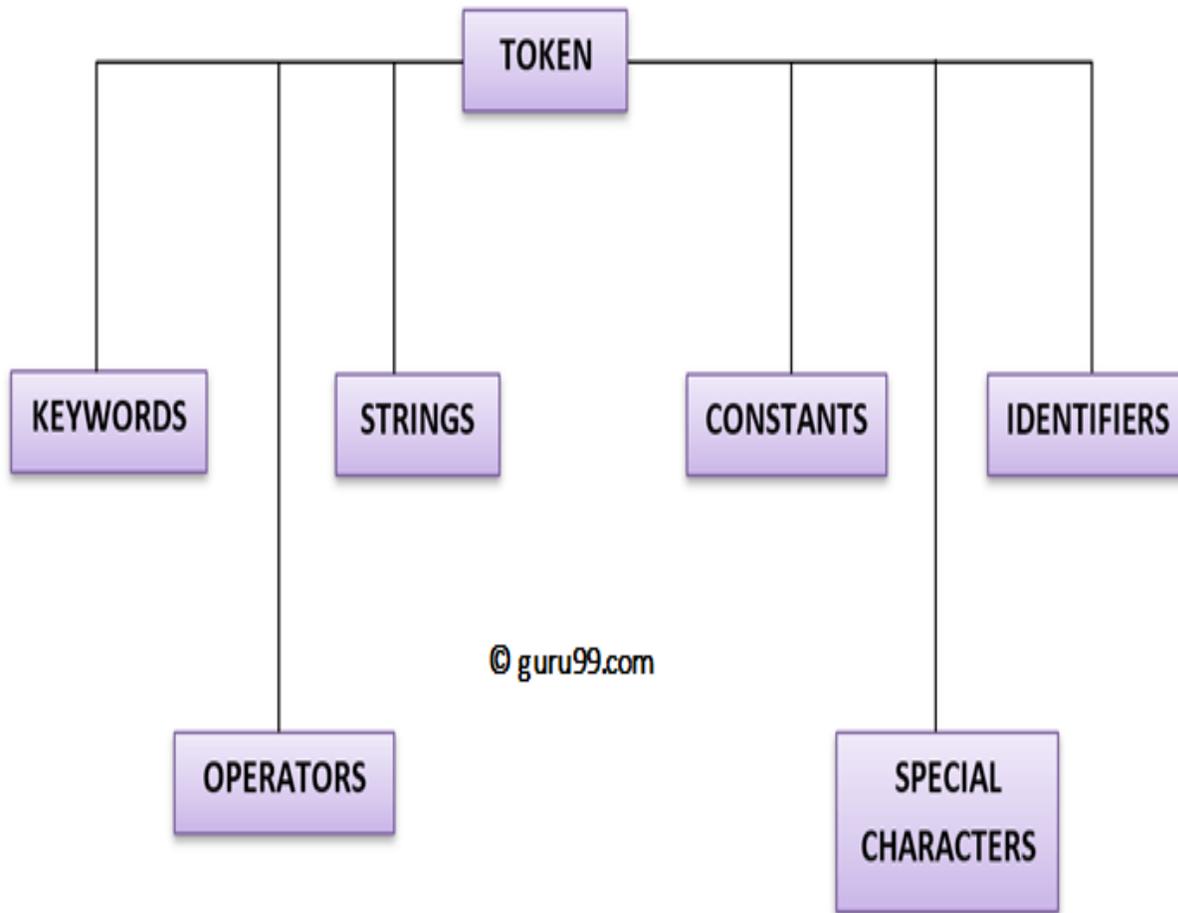
1. Letters
 - Uppercase characters (A-Z)
 - Lowercase characters (a-z)
2. Numbers
 - All the digits from 0 to 9
3. White spaces
 - Blank space
 - New line
 - Carriage return
 - Horizontal tab
4. Special characters
 - Special characters in 'C' are shown in the given table,

, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
; (semi-colon)	[(left bracket)
:] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)

(vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
\$ (dollar sign)	< (less than or opening angle bracket)
% (percentage sign)	# (hash sign)

What is Token in C?

TOKEN is the smallest unit in a 'C' program. It is each and every word and punctuation that you come across in your C program. The compiler breaks a program into the smallest possible units (tokens) and proceeds to the various stages of the compilation. A token is divided into six different types, viz, Keywords, Operators, Strings, Constants, Special Characters, and Identifiers.



Tokens in C

Keywords and Identifiers

In 'C' every word can be either a keyword or an identifier.

Keywords have fixed meanings, and the meaning cannot be changed. They act as a building block of a 'C' program. There are a total of 32 keywords in 'C'. Keywords are written in lowercase letters.

Following table represents the keywords in 'C'-

auto	double	int	struct
------	--------	-----	--------

break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	short	float	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc. Identifiers are the user-defined names consisting of 'C' standard character set. As the name says, identifiers are used to identify a particular element in a program. Each identifier must have a unique name. Following rules must be followed for identifiers:

1. The first character must always be an alphabet or an underscore.
2. It should be formed using only letters, numbers, or underscore.
3. A keyword cannot be used as an identifier.
4. It should not contain any whitespace character.
5. The name must be meaningful.

Summary

- A token is the smallest unit in a program.
- A keyword is reserved words by language.
- There are total of 32 keywords.
- An identifier is used to identify elements of a program.

What is a Variable?

A variable is an identifier which is used to store some value. Constants can never change at the time of execution. Variables can change during the execution of a program and update the value stored inside it.

A single variable can be used at multiple locations in a program. A variable name must be meaningful. It should represent the purpose of the variable.

Example: Height, age, are the meaningful variables that represent the purpose it is being used for. Height variable can be used to store a height value. Age variable can be used to store the age of a person

A variable must be declared first before it is used somewhere inside the program. A variable name is formed using characters, digits and an underscore.

Following are the rules that must be followed while creating a variable:

1. A variable name should consist of only characters, digits and an underscore.
2. A variable name should not begin with a number.
3. A variable name should not consist of whitespace.
4. A variable name should not consist of a keyword.
5. 'C' is a case sensitive language that means a variable named 'age' and 'AGE' are different.

Following are the examples of valid variable names in a 'C' program:

```
height or HEIGHT  
_height  
_height1  
My_name
```

Following are the examples of invalid variable names in a 'C' program:

```
1height  
Hei$ght  
My name
```

For example, we declare an integer variable **my_variable** and assign it the value 48:

```
int my_variable;  
my_variable = 48;
```

By the way, we can both declare and initialize (assign an initial value) a variable in a single statement:

```
int my_variable = 48;
```

Data types

'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application. Following are the three data types:

1. Primitive data types
2. Derived data types
3. User-defined data types

There are five primary fundamental data types,

1. int for integer data
2. char for character data
3. float for floating point numbers
4. double for double precision floating point numbers
5. void

Array, functions, pointers, structures are derived data types. 'C' language provides more extended versions of the above mentioned primary data types. Each data type differs from one another in size and range. Following table displays the size and range of each data type.

Data type	Size in bytes	Range
Char or signed char	1	-128 to 127
Unsigned char	1	0 to 255
int or signed int	2	-32768 to 32767
Unsigned int	2	0 to 65535
Short int or Unsigned short int	2	0 to 255
Signed short int	2	-128 to 127

Long int or Signed long int	4	-2147483648 to 2147483647
Unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
Long double	10	3.4E-4932 to 1.1E+4932

Note: In C, there is no Boolean data type.

Integer data type

Integer is nothing but a whole number. The range for an integer data type varies from machine to machine. The standard range for an integer data type is -32768 to 32767.

An integer typically is of 2 bytes which means it consumes a total of 16 bits in memory. A single integer value takes 2 bytes of memory. An integer data type is further divided into other data types such as short int, int, and long int.

Each data type differs in range even though it belongs to the integer data type family. The size may not change for each data type of integer family.

The short int is mostly used for storing small numbers, int is used for storing averagely sized integer values, and long int is used for storing large integer values.

Whenever we want to use an integer data type, we have place int before the identifier such as,

```
int age;
```

Here, age is a variable of an integer data type which can be used to store integer values.

Floating point data type

Like integers, in 'C' program we can also make use of floating point data types. The 'float' keyword is used to represent the floating point data type. It can hold a floating

point value which means a number is having a fraction and a decimal part. A floating point value is a real number that contains a decimal point. Integer data type doesn't store the decimal part hence we can use floats to store decimal part of a value.

Generally, a float can hold up to 6 precision values. If the float is not sufficient, then we can make use of other data types that can hold large floating point values. The data type double and long double are used to store real numbers with precision up to 14 and 80 bits respectively.

While using a floating point number a keyword float/double/long double must be placed before an identifier. The valid examples are,

```
float division;  
double BankBalance;
```

Character data type

Character data types are used to store a single character value enclosed in single quotes.

A character data type takes up-to 1 byte of memory space.

Example,

```
Char letter;
```

Void data type

A void data type doesn't contain or return any value. It is mostly used for defining functions in 'C'.

Example,

```
void displayData()
```

Type declaration of a variable

```
int main() {  
int x, y;  
float salary = 13.48;  
char letter = 'K';  
x = 25;  
y = 34;  
int z = x+y;  
printf("%d \n", z);  
printf("%f \n", salary);
```

```
printf("%c \n", letter);  
return 0;}
```

Output:

```
59  
13.480000  
K
```

We can declare multiple variables with the same data type on a single line by separating them with a comma. Also, notice the use of format specifiers in **printf** output function float (%f) and char (%c) and int (%d).

Constants

Constants are the fixed values that never change during the execution of a program. Following are the various types of constants:

Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

1. Decimal constant contains digits from 0-9 such as,

```
Example, 111, 1234
```

Above are the valid decimal constants.

2. Octal constant contains digits from 0-7, and these types of constants are always preceded by 0.

```
Example, 012, 065
```

Above are the valid decimal constants.

3. Hexadecimal constant contains a digit from 0-9 as well as characters from A-F. Hexadecimal constants are always preceded by 0X.

```
Example, 0X2, 0Xbcd
```

Above are the valid hexadecimal constants.

The octal and hexadecimal integer constants are very rarely used in programming with 'C'.

Character constants

A character constant contains only a single character enclosed within a single quote ("). We can also represent character constant by providing ASCII value of it.

```
Example, 'A', '9'
```

Above are the examples of valid character constants.

String constants

A string constant contains a sequence of characters enclosed within double quotes (").

```
Example, "Hello", "Programming"
```

These are the examples of valid string constants.

Real Constants

Like integer constants that always contains an integer value. 'C' also provides real constants that contain a decimal point or a fraction value. The real constants are also called as floating point constants. The real constant contains a decimal point and a fractional value.

```
Example, 202.15, 300.00
```

These are the valid real constants in 'C'.

A real constant can also be written as,

```
Mantissa e Exponent
```

For example, to declare a value that does not change like the classic circle constant PI, there are two ways to declare this constant

1. By using the **const** keyword in a variable declaration which will reserve a storage memory

```
#include <stdio.h>
int main() {
const double PI = 3.14;
printf("%f", PI);
```

```
//PI++; // This will generate an error as constants cannot be changed
return 0;}
```

2. By using the **#define** pre-processor directive which doesn't use memory for storage and without putting a semicolon character at the end of that statement

```
#include <stdio.h>
#define PI 3.14
int main() {
printf("%f", PI);
return 0;}
```

Summary

- A constant is a value that doesn't change throughout the execution of a program.
- A variable is an identifier which is used to store a value.
- There are four commonly used data types such as int, float, char and a void.
- Each data type differs in size and range from one another.

Keywords

C programs are constructed from a set of reserved words which provide control and from libraries which perform special functions. The basic instructions are built up using a reserved set of words, such as **main, for, if, while, default, double, extern, for, and int**, etc., C demands that they are used only for giving commands or making statements. You cannot use **default**, for example, as the name of a variable. An attempt to do so will result in a compilation error.

Keywords have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as programmer-defined identifiers. Keywords are an essential part of a language definition. They implement specific features of the language. Every C word is classified as either a keyword or an identifier. A keyword is a sequence of characters that the C compiler readily accepts and recognizes while being used in a program. Note that the keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier.

- The keywords are also called 'Reserved words'.
- Keywords are the words whose meaning has already been explained to the C compiler and their meanings cannot be changed.
- Keywords serve as basic building blocks for program statements.

- Keywords can be used only for their intended purpose.
- Keywords cannot be used as user-defined variables.
- All keywords must be written in lowercase.
- 32 keywords available in C.

Data types	Qualifiers	User-defined	Storage Classes	Loop	Others
int	signed	typedef	auto	for	const
char	unsigned	enum	extern	while	volatile
float	short	register	do	sizeof	
double	long		static		
Decision	Jump	Derived	function		
if	goto	struct	void		
else	continue	union	return		
switch	break				
case					
default					

Restrictions apply to keywords

- Keywords are the words whose meaning has already been explained to the C compiler and their meanings cannot be changed.
- Keywords can be used only for their intended purpose.
- Keywords cannot be used as user-defined variables.
- All keywords must be written in lowercase.

Data type Keywords

int	Specifies the integer type of value a variable will hold
char	Specifies the character type of value a variable will hold
float	Specifies the single-precision floating-point of value a variable will hold
double	Specifies the double-precision floating-point type of value a variable will

Qualifier Keywords

signed	Specifies a variable can hold positive and negative integer type of data
unsigned	Specifies a variable can hold only the positive integer type of data
short	Specifies a variable can hold fairly small integer type of data
long	Specifies a variable can hold fairly large integer type of data

Loop Control Structure Keywords

For	Loop is used when the number of passes is known in advance
While	Loop is used when the number of passes is not known in advance
Do	Loop is used to handle menu-driven programs

User-defined type Keywords

typedef	Used to define a new name for an existing data type
Enum	Gives an opportunity to invent own data type and define what values the variable of this data type can take

Jumping Control Keywords

Break	Used to force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop
continue	Used to take the control to the beginning of the loop bypassing the statements inside the loop
Goto	Used to take the control to required place in the program

Storage Class Keywords

Storage Classes	Storage	Default initial value	Scope	Life
auto	Memory	An unpredictable value	Local	Till the control remains within the block
register	CPU registers	Garbage value	Local	Till the control remains within the block
static	Memory	Zero	Local	Value of the variable persists between different function calls
extern	Memory	Zero	Global	Till the program's execution doesn't come to an end

Review Questions

Part - A

1. Define Keyword.
2. List any four key words of C language.
3. What are keywords? Give examples.
4. What are the restrictions apply for keywords?
5. List any four keywords (Reserved words) of C language

Part - B

1. Write short notes on: keywords (Reserved words) in C
2. State the meaning of the following keywords in C: auto, double, int, long

Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumeric characters that represent the identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

1. total, sum, average, _m_, sum_1, etc.

Example of invalid identifiers

1. 2sum (starts with a numerical digit)
2. **int** (reserved word)
3. **char** (reserved word)
4. m+n (special character, i.e., '+')

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

```
1. int main()  
2. {  
3.   int a=10;  
4.   int A=20;  
5.   printf("Value of a is : %d",a);  
6.   printf("\nValue of A is :%d",A);
```

- 7. `return 0;`
- 8. `}`

Output

```
Value of a is : 10  
Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, `playerScore` is a variable of `int` type. Here, the variable is assigned an integer value `95`.

The value of a variable can be changed, hence the name variable.

```
char ch = 'a';  
// some code  
ch = 'l';
```

Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Note: You should always try to give meaningful names to variables. For example: `firstName` is a better variable name than `fn`.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;    // integer variable
number = 5.5;    // error
double number;    // error
```

Here, the type of `number` variable is `int`. You cannot assign a floating-point (decimal) value `5.5` to this variable. Also, you cannot redefine the data type of the variable to `double`. By the way, to store the decimal values in C, you need to declare its type to either `double` or `float`.

Visit this page to learn more about different types of data a variable can store.

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.

Here, `1`, `2.5` and `'c'` are literals. Why? You cannot assign different values to these terms.

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

In C programming, octal starts with a `0`, and hexadecimal starts with a `0x`.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: `E-5 = 10-5`

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: `'a'`, `'m'`, `'F'`, `'2'`, `'\'` etc.

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences

Escape Sequences

Character

`\b`

Backspace

`\f`

Form feed

`\n`

Newline

`\r`

Return

`\t`

Horizontal tab

`\v`

Vertical tab

`\\`

Backslash

`\'`

Single quotation mark

`\"`

Double quotation mark

`\?`

Question mark

`\0`

Null character

For example: `\n` is used for a newline. The backslash `\` causes escape from the normal way the characters are handled by the compiler.

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"           //string constant
""              //null string constant
"  "           //string constant of six white space
"x"            //string constant having a single character.
"Earth is round\n" //prints string with a newline
```

Constants

If you want to define a variable whose value cannot be changed, you can use the `const` keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword `const`.

Here, `PI` is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;
PI = 2.9; //Error
```

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
<code>int</code>	at least 2, usually 4	<code>%d</code> , <code>%i</code>
<code>char</code>	1	<code>%c</code>
<code>float</code>	4	<code>%f</code>
<code>double</code>	8	<code>%lf</code>
<code>short int</code>	2 usually	<code>%hd</code>
<code>unsigned int</code>	at least 2, usually 4	<code>%u</code>
<code>long int</code>	at least 4, usually 8	<code>%ld</code> , <code>%li</code>
<code>long long int</code>	at least 8	<code>%lld</code> , <code>%lli</code>
<code>unsigned long int</code>	at least 4	<code>%lu</code>
<code>unsigned long long int</code>	at least 8	<code>%llu</code>

Type	Size (bytes)	Format Specifier
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10

We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from -2147483648 to 2147483647.

float and double

`float` and `double` are used to hold real numbers.

```
float salary;
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types –

- **Numbers**, eg. 123, 120.50 etc.
- **Strings** of text e.g. "This text string" etc.
- **Boolean** e.g. true or false.

JavaScript also defines two trivial data types, **null** and **undefined**, each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as **object**. We will cover objects in detail in a separate chapter.

Note – JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows.

```
<script type = "text/javascript">
  <!--
    var money;
    var name;
  //-->
</script>
```

You can also declare multiple variables with the same **var** keyword as follows –

```
<script type = "text/javascript">
  <!--
    var money, name;
  //-->
```

```
</script>
```

Storing a value in a variable is called **variable initialization**. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named **money** and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type = "text/javascript">
  <!--
    var name = "Ali";
    var money;
    money = 2000.50;
  //-->
</script>
```

Note – Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables** – A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables** – A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
<html>
  <body onload = checkscope();>
    <script type = "text/javascript">
      <!--
        var myVar = "global";    // Declare a global variable
        function checkscope( ) {
```

```

    var myVar = "local"; // Declare a local variable
    document.write(myVar);
  }
  //-->
</script>
</body>
</html>

```

This produces the following result –

local

JavaScript Variable Names

While naming your variables in JavaScript, keep the following rules in mind.

- You should not use any of the JavaScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, **break** or **boolean** variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, **123test** is an invalid variable name but **_123test** is a valid one.
- JavaScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws

catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

What is an Operator?

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

Lets have a look on all operators one by one.

Arithmetic Operators

JavaScript supports the following arithmetic operators –

Assume variable A holds 10 and variable B holds 20, then –

Sr.No.	Operator & Description
1	+ (Addition) Adds two operands Ex: A + B will give 30
2	- (Subtraction) Subtracts the second operand from the first Ex: A - B will give -10
3	* (Multiplication) Multiply both operands Ex: A * B will give 200
4	/ (Division) Divide the numerator by the denominator Ex: B / A will give 2
5	% (Modulus) Outputs the remainder of an integer division Ex: B % A will give 0
6	++ (Increment) Increases an integer value by one Ex: A++ will give 11
7	-- (Decrement)

Decreases an integer value by one

Ex: A-- will give 9

Note – Addition operator (+) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example

The following code shows how to use arithmetic operators in JavaScript.

```
<html>
<body>

  <script type = "text/javascript">
    <!--
      var a = 33;
      var b = 10;
      var c = "Test";
      var linebreak = "<br />";

      document.write("a + b = ");
      result = a + b;
      document.write(result);
      document.write(linebreak);

      document.write("a - b = ");
      result = a - b;
      document.write(result);
      document.write(linebreak);

      document.write("a / b = ");
      result = a / b;
      document.write(result);
      document.write(linebreak);

      document.write("a % b = ");
      result = a % b;
      document.write(result);
      document.write(linebreak);

      document.write("a + b + c = ");
      result = a + b + c;
      document.write(result);
      document.write(linebreak);
```

```

a = ++a;
document.write(++a = ");
result = ++a;
document.write(result);
document.write(linebreak);

b = --b;
document.write("--b = ");
result = --b;
document.write(result);
document.write(linebreak);
//-->
</script>

```

Set the variables to different values and then try...

```

</body>
</html>

```

Output

```

a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 35
--b = 8

```

Set the variables to different values and then try...

Comparison Operators

JavaScript supports the following comparison operators –

Assume variable A holds 10 and variable B holds 20, then –

Sr.No.	Operator & Description
1	<p>== (Equal)</p> <p>Checks if the value of two operands are equal or not, if yes, then the condition becomes true.</p> <p>Ex: (A == B) is not true.</p>

2	<p>!= (Not Equal)</p> <p>Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true.</p> <p>Ex: (A != B) is true.</p>
3	<p>> (Greater than)</p> <p>Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true.</p> <p>Ex: (A > B) is not true.</p>
4	<p>< (Less than)</p> <p>Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true.</p> <p>Ex: (A < B) is true.</p>
5	<p>>= (Greater than or Equal to)</p> <p>Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true.</p> <p>Ex: (A >= B) is not true.</p>
6	<p><= (Less than or Equal to)</p> <p>Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true.</p> <p>Ex: (A <= B) is true.</p>

Example

The following code shows how to use comparison operators in JavaScript.

```

<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = 20;
        var linebreak = "<br />";

```

```
document.write("(a == b) => ");
result = (a == b);
document.write(result);
document.write(linebreak);

document.write("(a < b) => ");
result = (a < b);
document.write(result);
document.write(linebreak);

document.write("(a > b) => ");
result = (a > b);
document.write(result);
document.write(linebreak);

document.write("(a != b) => ");
result = (a != b);
document.write(result);
document.write(linebreak);

document.write("(a >= b) => ");
result = (a >= b);
document.write(result);
document.write(linebreak);

document.write("(a <= b) => ");
result = (a <= b);
document.write(result);
document.write(linebreak);
//-->
</script>
Set the variables to different values and different operators and then try...
</body>
</html>
```

Output

(a == b) => false
(a < b) => true
(a > b) => false
(a != b) => true
(a >= b) => false
a <= b) => true

Set the variables to different values and different operators and then try...

Logical Operators

JavaScript supports the following logical operators –

Assume variable A holds 10 and variable B holds 20, then –

Sr.No.	Operator & Description
1	&& (Logical AND) If both the operands are non-zero, then the condition becomes true. Ex: (A && B) is true.
2	 (Logical OR) If any of the two operands are non-zero, then the condition becomes true. Ex: (A B) is true.
3	! (Logical NOT) Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. Ex: ! (A && B) is false.

Example

Try the following code to learn how to implement Logical Operators in JavaScript.

```
<html>
<body>
  <script type = "text/javascript">
    <!--
      var a = true;
      var b = false;
      var linebreak = "<br />";

      document.write("(a && b) => ");
      result = (a && b);
      document.write(result);
      document.write(linebreak);

      document.write("(a || b) => ");
      result = (a || b);
```

```

document.write(result);
document.write(linebreak);

document.write("!(a && b) => ");
result = (!(a && b));
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
</body>
</html>

```

Output

(a && b) => false

(a || b) => true

!(a && b) => true

Set the variables to different values and different operators and then try...

Bitwise Operators

JavaScript supports the following bitwise operators –

Assume variable A holds 2 and variable B holds 3, then –

Sr.No.	Operator & Description
1	<p>& (Bitwise AND)</p> <p>It performs a Boolean AND operation on each bit of its integer arguments.</p> <p>Ex: (A & B) is 2.</p>
2	<p> (BitWise OR)</p> <p>It performs a Boolean OR operation on each bit of its integer arguments.</p> <p>Ex: (A B) is 3.</p>
3	<p>^ (Bitwise XOR)</p> <p>It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.</p>

	Ex: (A ^ B) is 1.
4	<p>~ (Bitwise Not)</p> <p>It is a unary operator and operates by reversing all the bits in the operand.</p> <p>Ex: (~B) is -4.</p>
5	<p><< (Left Shift)</p> <p>It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on.</p> <p>Ex: (A << 1) is 4.</p>
6	<p>>> (Right Shift)</p> <p>Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.</p> <p>Ex: (A >> 1) is 1.</p>
7	<p>>>> (Right shift with Zero)</p> <p>This operator is just like the >> operator, except that the bits shifted in on the left are always zero.</p> <p>Ex: (A >>> 1) is 1.</p>

Example

Try the following code to implement Bitwise operator in JavaScript.

```
<html>
<body>
  <script type = "text/javascript">
    <!--
      var a = 2; // Bit presentation 10
      var b = 3; // Bit presentation 11
      var linebreak = "<br />";

      document.write("(a & b) => ");
      result = (a & b);
```

```

document.write(result);
document.write(linebreak);

document.write("(a | b) => ");
result = (a | b);
document.write(result);
document.write(linebreak);

document.write("(a ^ b) => ");
result = (a ^ b);
document.write(result);
document.write(linebreak);

document.write("(~b) => ");
result = (~b);
document.write(result);
document.write(linebreak);

document.write("(a << b) => ");
result = (a << b);
document.write(result);
document.write(linebreak);

document.write("(a >> b) => ");
result = (a >> b);
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
</body>
</html>

```

(a & b) => 2

(a | b) => 3

(a ^ b) => 1

(~b) => -4

(a << b) => 16

(a >> b) => 0

Set the variables to different values and different operators and then try...

Assignment Operators

JavaScript supports the following assignment operators –

Sr.No.	Operator & Description
1	<p>= (Simple Assignment)</p> <p>Assigns values from the right side operand to the left side operand</p> <p>Ex: $C = A + B$ will assign the value of $A + B$ into C</p>
2	<p>+= (Add and Assignment)</p> <p>It adds the right operand to the left operand and assigns the result to the left operand.</p> <p>Ex: $C += A$ is equivalent to $C = C + A$</p>
3	<p>-= (Subtract and Assignment)</p> <p>It subtracts the right operand from the left operand and assigns the result to the left operand.</p> <p>Ex: $C -= A$ is equivalent to $C = C - A$</p>
4	<p>*= (Multiply and Assignment)</p> <p>It multiplies the right operand with the left operand and assigns the result to the left operand.</p> <p>Ex: $C *= A$ is equivalent to $C = C * A$</p>
5	<p>/= (Divide and Assignment)</p> <p>It divides the left operand with the right operand and assigns the result to the left operand.</p> <p>Ex: $C /= A$ is equivalent to $C = C / A$</p>
6	<p>%= (Modules and Assignment)</p> <p>It takes modulus using two operands and assigns the result to the left operand.</p> <p>Ex: $C %= A$ is equivalent to $C = C \% A$</p>

Note – Same logic applies to Bitwise operators so they will become like $\ll=$, $\gg=$, $\gt;>=$, $\&=$, $|=$ and $\wedge=$.

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
<body>
  <script type = "text/javascript">
    <!--
      var a = 33;
      var b = 10;
      var linebreak = "<br />";

      document.write("Value of a => (a = b) => ");
      result = (a = b);
      document.write(result);
      document.write(linebreak);

      document.write("Value of a => (a += b) => ");
      result = (a += b);
      document.write(result);
      document.write(linebreak);

      document.write("Value of a => (a -= b) => ");
      result = (a -= b);
      document.write(result);
      document.write(linebreak);

      document.write("Value of a => (a *= b) => ");
      result = (a *= b);
      document.write(result);
      document.write(linebreak);

      document.write("Value of a => (a /= b) => ");
      result = (a /= b);
      document.write(result);
      document.write(linebreak);

      document.write("Value of a => (a %= b) => ");
      result = (a %= b);
      document.write(result);
      document.write(linebreak);
    //-->
  </script>
  <p>Set the variables to different values and different operators and then try...</p>
</body>
```

```
</html>
```

Output

Value of a => (a = b) => 10

Value of a => (a += b) => 20

Value of a => (a -= b) => 10

Value of a => (a *= b) => 100

Value of a => (a /= b) => 10

Value of a => (a %= b) => 0

Set the variables to different values and different operators and then try...

Miscellaneous Operator

We will discuss two operators here that are quite useful in JavaScript: the **conditional operator** (?:) and the **typeof operator**.

Conditional Operator (?:)

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

Sr.No.	Operator and Description
1	? : (Conditional) If Condition is true? Then value X : Otherwise value Y

Example

Try the following code to understand how the Conditional Operator works in JavaScript.

```
<html>
<body>
  <script type = "text/javascript">
    <!--
      var a = 10;
      var b = 20;
      var linebreak = "<br />";

      document.write ("((a > b) ? 100 : 200) => ");
      result = (a > b) ? 100 : 200;
      document.write(result);
      document.write(linebreak);
```

```
document.write ("((a < b) ? 100 : 200) ==> ");
result = (a < b) ? 100 : 200;
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
</body>
</html>
```

Output

((a > b) ? 100 : 200) ==> 200
((a < b) ? 100 : 200) ==> 100
Set the variables to different values and different operators and then try...

typeof Operator

The **typeof** operator is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The typeof operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.

Here is a list of the return values for the **typeof** Operator.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"

Null	"object"
------	----------

Example

The following code shows how to implement **typeof** operator.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = "String";
        var linebreak = "<br />";

        result = (typeof b == "string" ? "B is String" : "B is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);

        result = (typeof a == "string" ? "A is String" : "A is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

Result => B is String

Result => A is Numeric

Set the variables to different values and different operators and then try...

Constant

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals –

```
212      /* Legal */
215u     /* Legal */
0xFeeL   /* Legal */
078      /* Illegal: 8 is not an octal digit */
032UU    /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals –

```
85       /* decimal */
0213     /* octal */
0x4b     /* hexadecimal */
30       /* int */
30u      /* unsigned int */
30l      /* long */
30ul     /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals –

```
3.14159  /* Legal */
314159E-5L /* Legal */
```


String literals or constants are enclosed in double quotes `""`. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
```

```
"hello, \  
dear"
```

```
"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C to define constants –

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

Given below is the form to use #define preprocessor to define a constant –

```
#define identifier value
```

The following example explains it in detail –

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'

int main() {
    int area;

    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –
value of area : 50

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows –
const type variable = value;

The following example explains it in detail –

```
#include <stdio.h>

int main() {
    const int LENGTH = 10;
    const int WIDTH = 5;
    const char NEWLINE = '\n';
    int area;

    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –
value of area : 50

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program –

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
    int mount;
    auto int month;
```

```
}
```

The example above defines two variables within the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{  
    register int miles;  
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>  
  
/* function declaration */  
void func(void);  
  
static int count = 5; /* global variable */  
  
main() {  
  
    while(count-->0) {  
        func();  
    }  
}
```

```

    return 0;
}

/* function definition */
void func( void ) {

    static int i = 5; /* local static variable */
    i++;

    printf("i is %d and count is %d\n", i, count);
}

```

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

```

i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0

```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```

#include <stdio.h>

int count ;
extern void write_extern();

main() {
    count = 5;
    write_extern();
}

```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
    printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows –

```
$gcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it produces the following result –

```
count is 5
```

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

We will, in this chapter, look into the way each operator works.

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

Operator	Description	Example
+	Adds two operands.	A + B = 30
-	Subtracts second operand from the first.	A - B = -10

*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then –

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	$(A > B)$ is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	$(A < B)$ is true.
>=	Checks if the value of left operand is greater than or equal to the	$(A >= B)$ is not

	value of right operand. If yes, then the condition becomes true.	true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1

1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows –

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of	(~A) = ~(60),

	'flipping' bits.	i.e., - 0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language –

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A

*= 	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	Bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

Misc Operators → sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

Show Examples

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

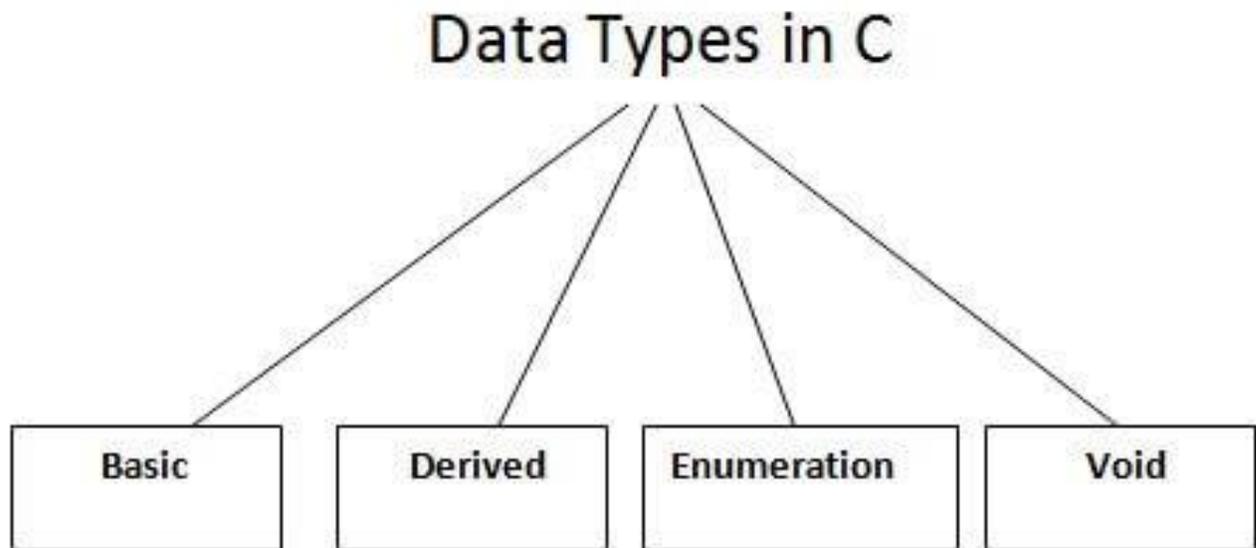
Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right

Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Data Types

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	

double	8 byte	
long double	10 byte	

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	break	case	char	const	continue	default
double	else	enum	extern	float	for	goto
int	long	register	return	short	signed	sizeof
struct	switch	typedef	union	unsigned	void	volatile

We will learn about all the C language keywords later.

C Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumeric characters that represent the identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.

- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

1. total, sum, average, _m _, sum_1, etc.

Example of invalid identifiers

1. 2sum (starts with a numerical digit)
2. **int** (reserved word)
3. **char** (reserved word)
4. m+n (special character, i.e., '+')

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier

Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

```

1. int main()
2. {
3.     int a=10;
4.     int A=20;
5.     printf("Value of a is : %d",a);
6.     printf("\nValue of A is :%d",A);
7.     return 0;
8. }
```

Output

```

Value of a is : 10
Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

Data Types in C with Examples

There are 4 Data types in C:

- Basic
- Derived

- Void
- Enumeration

Most of the time, for small programs, we use the basic fundamental data types in C – int, char, float, and double.

For more complex and huge amounts of data, we use derived types – array, structure, union, and pointer.

Enumeration and void consist of enum and void, respectively. We will discuss these later in the article.

Basic Data Types

These are also termed as primary or fundamental data types. All the names mean the same thing. Suppose we have to store student details like name, id, group, avg_marks, interest_on_fees.

We can use basic data types to store each of these data:

```
char name[25];
```

```
int id;
```

```
char group;
```

```
float marks[5];
```

```
double interest;
```

int Data Type

Integer types can be signed (with negative values) or unsigned values (only positive). Int values are always signed unless specifically mentioned.

Integer types are further classified as –

Data type	Range
int	
signed int	-32,768 to 32,767

unsigned int	0 to 65,535
short int	
signed short int	-2,147,483,648 to 2,147,483,647 (4 bytes)
unsigned short int	0 to 4,294,967,295 (4 bytes)
long int	
signed long int	-2,147,483,648 to 2,147,483,647 (4 bytes)
unsigned long int	0 to 4,294,967,295 (4 bytes)

Some examples:

```
int number = 456;
```

```
long prime = 12230234029;
```

How to print integer variables? Here is a small program that you can try and tweak to get different results and understand the range of short, int, and long.

```
#include

int main(void) {

    short int num1 = 10000;

    int number = 121113991;

    long prime = 49929929991;

    long notprime = 2300909090909933322;

    long long sum = prime + notprime;

    printf("num1 is %hd, number is %d, prime is %ld, notprime is %ld, sum is %lld", num1,
    number, prime, notprime, sum);
```

```
return 0;
}
```

We have used %hd for short, %d for int, and so on for printing each data type.

Note that we have used 'long long' for sum, which is 8 bytes, whereas long is 4 bytes. Though in practical situations, we may not use numbers that are this big, it is good to know the range and what data type we should use for programs with exponential calculations. We can use %u in place of %d for unsigned int but even %d works. Let us say the value of long notprime = -2300909090909933322; has a minus, but we print it as notprime is %lu, the correct value will not be printed. This is why it is safe to use %ld, unless you want the values to be always unsigned.

If we add more digits to short int num1 = 10000, it will be out of range and will print wrong value. 'short int' can be used to limit the size of the integer data type.

Float

The floating point data type allows the user to type decimal values. For example, average marks can be 97.665. if we use int data type, it will strip off the decimal part and print only 97. To print the exact value, we need 'float' data type.

Float is 4 bytes, and we can print the value using %f.

The float can contain int values too.

```
float average = 97.665;

float mark = 67;

printf("average is %f", average);

printf(" mark is %f", mark);
```

However, you will get the result of the mark as 67.00000, which may not be a pleasant sight with a lot of redundant zeroes. If you try to print the value of mark as %d after declaring it as float, you will not get 67. Try to run this program and see what value you get.

Double

You can think of float, double and long double similar to short int, int, and long int. Double is 8 bytes, which means you can have more precision than float. This is useful in scientific programs that require precision. Float is just a single-precision data type;

double is the double-precision data type. Long Double is treated the same as double by most compilers; however, it was made for quadruple data precision.

```
double average = 679999999.454;

float score = 679999999.454;

printf("average is %lf", average);

printf(", score is %f", score);
```

The outputs are –
the average is 679999999.454000, the score is 680000000.000000
Note the difference in outputs – while double prints the exact value, float value is rounded off to the nearest number.

char

char stores a single character. Char consists of a single byte.

For example,

```
char group = 'B';
```

To print a name or a full string, we need to define char array.

```
char group = 'B';
```

```
char name[30] = "Student1";
```

```
printf("group is %c, name is %s", group, name);
```

Note that for a single character, we use single quotes, but for String (character array), we use double-quotes. Since its an array, we have to specify the length (30 in this case).

Just like the int data type, char can be signed (range from -128 to +127) or unsigned (0 to 255). C stores the binary equivalent of the Unicode/ASCII value of any character that we type. In our above example, the char group will be stored as a value '066'.

You can think of char also as an int value, as char takes int values too. The importance of signed and unsigned comes when you store an int between the specified range in a char.

Here is an example to help understand signed and unsigned chars better –

```
signed char char1 = -127;

unsigned char char2 = -127;

printf("char1 is %d, char2 is %d", char1, char2);
```

Note that since we are taking in int values, we will print as %d and not %c. Since char1 is signed, the printf will give value as -127. However, char2 is unsigned, which means the range is from 0 to 255, -127 is out of range. So, it will print 129. Same way, if you assign char2 as -1, you will get a value of 255.

Derived Data Types

Array, pointers, struct, and union are the derived data types in C.

Array

Same as any other language, Array in C stores multiple values of the same data type. That means we can have an array of integers, chars, floats, doubles, etc

```
int numbers[] = ;

double marks[7];

float interest[5] = ;
```

The array needs to be either initialized, or the size needs to be specified during the declaration.

To understand one-dimensional Array operations, let us go through the following simple code –

```
#include

int main(void) {

    // declare array with maximum 5 values

    int marks[5];

    // get the size of the array

    int noOfSubjects = sizeof(marks)/sizeof(int);
```

```

// let us get the inputs from user
for(int i=0; i<noOfSubjects; i++)
{
printf("\nEnter marks ");
scanf("%d", &marks[i]);
}

double average;

double sum = 0;

// fetch individual array elements
for(int i=0; i<noOfSubjects; i++)

// let us print the average of marks

average = sum/noOfSubjects;

printf("\nAverage marks = %lf", average);

return 0;

}

```

Few points to note here:

- If we don't enter any value for marks, marks[i] will have defaulted to zero.
- If the sum is an int, sum/noOfSubjects will round off the average to nearest value and print only the value before decimal (even though average is of double data type). We can also do type casting to avoid this.
- Each element in the array is filled by using marks[i], where I correspond to the respective element. Same way, to fetch the data, we again loop through the array using marks[i] to get individual elements.
- sum += marks[i]; is same as writing sum = sum + marks[i];

In C, arrays can be multi-dimensional. For simplicity, we will restrict to a two-dimensional array.

```
dataType arrayName [rows][columns];
```

For example,

```
int matrix1[3][5] = {  
    , //first row with index 0  
    , // second row with index 1  
    // third row with index 2  
};
```

The index starts with 0 for both rows and columns. For example –

```
matrix1[0][0] will be 1.  
matrix1[1][1] will be 12.  
matrix1[2][2] will be 23.  
matrix1[2][4] will be 25.
```

If you have to access these values through a program, you will need two loop counters, the outer one for the rows, and the inner one for the columns.

Pointers

Pointers are considered by many to be complex in C, but that is not the case. Simply put, a pointer is just a variable that stores the address of another variable. A pointer can store the address of variables of any data types. This allows for dynamic memory allocation in C. Pointers also help in passing variables by reference.

The pointer is defined by using a ‘**’ operator. For example –

```
int *ptr;
```

This indicates ptr stores an address and not a value. To get the address of the variable, we use the dereference operator '&.' The size of a pointer is 2 bytes. Pointers cannot be added, multiplied, or divided. However, we can subtract them. This will help us know the number of elements present between the two subtracted pointers. Here is a simple program that illustrates pointer –

```
#include

int main(void) {

    int *ptr1;

    int *ptr2;

    int a = 5;

    int b = 10;

    /* address of a is assigned to ptr1*/

    ptr1 = &a;

    /* address of b is assigned to ptr2*/

    ptr2 = &b;

    /* display value of a and b using pointer variables */

    printf("%d", *ptr1); //prints 5

    printf("\n%d", *ptr2); //prints 10

    //print address of a and b

    printf("\n%d", ptr1); // prints address like -599163656

    printf("\n%d", ptr2); // prints address like -599163652

    // pointer subtraction

    int minus = ptr2 - ptr1;

    printf("\n%d", minus); // prints the difference (in this case 1)
```

```
return 0;
}
```

Structs

A struct is a composite structure that can contain variables of different data types. For example, all the student data that we declared earlier in basic data types can be put under one structure. Instead of having the information scattered, when we give it a structure, it is easier to store information about more students.

```
typedef struct{
    char name[25];
    int id;
    char group;
    float marks[5];
    double interest;
}Student;
```

A structure can be created outside the main method as well as inside, just before creating the variable to use it.

```
struct student1, student[20];
```

Structure members can be accessed **using the dot(.) operator**. For example,

```
printf("Student id is %d - ", student1.id);
```

Elements in structure can be accessed using pointers too. There is no toString() method in C (like Java has), so to print struct values, we need to fetch them individually and print.

Here is a small program that shows the same (for simplicity, I have hard-coded the data, you can do a for loop and get the data from user too and store it the same as in an array).

```

int main(void) {

    // Store values in structures

    Student st1 = {"student1", 1, 'a', , 4.5};

    Student st2 = {"student2", 2, 'b', , 9.5};

    // Send structure values to the printing method

    print_student_details(&st1);

    print_student_details(&st2);

    return 0;

}

// get the address of structure data and print

void print_student_details(Student *st) {

    printf("\nStudent details for %s are:\n", st->name);

    printf("id: %d\n",st->id);

    printf("group %c\n", st->group);

    // since marks is an array, loop through to get the data

    for(int i=0;i<5;i++)

        printf("marks %f\n", st->marks[i]);

    printf("interest %lf", st->interest);

}

```

- Using the * operator, we are passing the value of student struct by reference, so that the correct values are retained.
- Instead of the dot operator, we are using '->' operator to fetch the values.

Structs are simple to use and combine data in a neat way.

Union

With a union, you can store different data types in the same memory location. The union can have many members, but only one member can have a value at one time. Union, is thus, a special kind of data type in C.

The union is defined in the same way as a structure but with the keyword union.

```
union Student{  
  
    char name[25];  
  
    int id;  
  
    char group;  
  
    float marks[5];  
  
    double interest;  
  
}st1, st2;
```

When we assign values to union data, union allocates enough memory to accommodate the largest data type defined. For example, since the name takes the biggest space in the memory out of all the other data types, the union will allocate the space taken by name.

Let's say we assign and print multiple values in the union at the same time.

```
st1.id = 1;  
  
st1.group = 'a';  
  
strcpy(st1.name, "student1");  
  
printf( "ID : %d\n", st1.id);  
  
printf( "Group : %c\n", st1.group);  
  
printf( "Name : %s\n", st1.name);
```

Unlike struct, this will fetch output as –

```
ID : 1685419123
```

```
Group : s
```

```
Name : student1
```

Only the value of the member name is correct; other values have been corrupted. However, if we assign and print the values one by one, we will get all the values correctly.

```
st1.id = 1;

printf( "ID : %d\n", st1.id);

st1.group = 'a';

printf( "Group : %c\n", st1.group);

strcpy(st1.name, "student1");

printf( "Name : %s\n", st1.name);
```

Now, we get the output as –

```
ID : 1
```

```
Group : a
```

```
Name : student1
```

Read this [blog](#) to know more differences between structures and unions.

Enumeration

Enumeration data types enhance the readability of the code. If you have integer constants in the code that can be reused or clubbed together, we can use enums to define the constants. The most common example of this is the days of the week.

```
enum weekdays;
```

```
enum weekend;
```

Internally, C will store MON as 0, TUE as one, and so on. We can assign values to the enum as well.

```
enum weekdays;
```

If we **print each** of the enum **values**, the output will be –

```
1, 2, 6, 7, 8
```

Enums are very useful and can be used as flags. They provide flexibility and efficiency in the code.

Void

The void is just an empty data type used as a return type for functions. The absence of any other data type is void. When you declare a function as void, it doesn't have to return anything. For example –

```
void swapNumbers(int a, int b){  
  
    //multiple lines of code here  
  
}
```

Same way, if a function does not have any parameters, that can be indicated with the void.

```
int getNumbers(void){  
  
    // some code  
  
}
```

We can declare a void pointer so that it can take a variable of any data type. A pointer declared as void becomes a general-purpose pointer –

```
char *ptr;
```

```
int value;
```

```
ptr = &value; //this will give error because we cannot point a char pointer to an int value
```

However,

```
void *ptr;
```

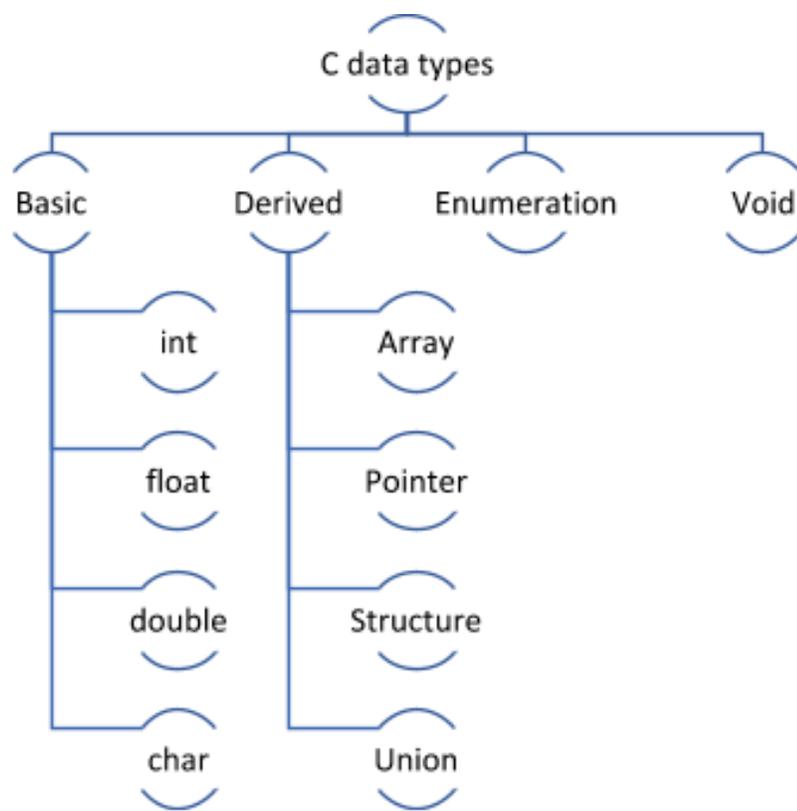
will solve **this** problem and now we can write

```
ptr = &value;
```

without any compilation errors. You can assign any data type to the void pointer.

Conclusion

In this blog, we have discussed all the data types in C in detail i.e., basic, derived, enumeration, and void. All the data types are useful in their own ways and make C the robust language it is. Check out C tutorials and best C books to further learn the language and clear your concepts. For a quick reference, use this diagram to remember all the data types in one go:



Comments

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash `//`. Let's see an example of a single line comment in C.

1. `#include<stdio.h>`
2. `int main(){`
3. `//printing information`
4. `printf("Hello C");`
5. `return 0;`
6. `}`

Output:

Hello C

Even you can place the comment after the statement. For example:

1. `printf("Hello C");//printing information`

Multi-Line Comments

Multi-Line comments are represented by slash asterisk `/* ... */`. It can occupy many lines of code, but it can't be nested. Syntax:

1. `/*`
2. code
3. to be commented
4. `*/`

Let's see an example of a multi-Line comment in C.

1. #include<stdio.h>
2. **int** main(){
3. /*printing information
4. Multi-Line Comment*/
5. printf("Hello C");
6. **return** 0;
7. }

Output:

Hello C

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.

<code>%e/%E</code>	It is used for scientific notation. It is also known as Mantissa or Exponent.
<code>%g</code>	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
<code>%p</code>	It is used to print the address in a hexadecimal form.
<code>%c</code>	It is used to print the unsigned character.
<code>%s</code>	It is used to print the strings.
<code>%ld</code>	It is used to print the long-signed integer value.

Let's understand the format specifiers in detail through an example.

- **%d**

```

1. int main()
2. {
3.   int b=6;
4.   int c=8;
5.   printf("Value of b is:%d", b);
6.   printf("\nValue of c is:%d",c);
7.
8.   return 0;
9. }
```

In the above code, we are printing the integer value of b and c by using the %d specifier.

Output

- **%u**

```

1. int main()
2. {
3.   int b=10;
4.   int c= -10;
5.   printf("Value of b is:%u", b);
```

```
6. printf("\nValue of c is:%u",c);
7.
8.   return 0;
9. }
```

In the above program, we are displaying the value of b and c by using an unsigned format specifier, i.e., %u. The value of b is positive, so %u specifier prints the exact value of b, but it does not print the value of c as c contains the negative value.

Output

- **%o**

```
1. int main()
2. {
3.   int a=0100;
4.   printf("Octal value of a is: %o", a);
5.   printf("\nInteger value of a is: %d",a);
6.   return 0;
7. }
```

In the above code, we are displaying the octal value and integer value of a.

Output

- **%x and %X**

```
1. int main()
2. {
3.   int y=0xA;
4.   printf("Hexadecimal value of y is: %x", y);
5.   printf("\nHexadecimal value of y is: %X",y);
6.   printf("\nInteger value of y is: %d",y);
7.   return 0;
8. }
```

In the above code, y contains the hexadecimal value 'A'. We display the hexadecimal value of y in two formats. We use %x and %X to print the hexadecimal value where %x

displays the value in small letters, i.e., 'a' and %X displays the value in a capital letter, i.e., 'A'.

Output

- %f

```
1. int main()
2. {
3.     float y=3.4;
4.     printf("Floating point value of y is: %f", y);
5.     return 0;
6. }
```

The above code prints the floating value of y.

Output

- %e

```
1. int main()
2. {
3.     float y=3;
4.     printf("Exponential value of y is: %e", y);
5.     return 0;
6. }
```

Output

- %E

```
1. int main()
2. {
3.     float y=3;
4.     printf("Exponential value of y is: %E", y);
5.     return 0;
6. }
```

Output

- **%g**

```
1. int main()
2. {
3.     float y=3.8;
4.     printf("Float value of y is: %g", y);
5.     return 0;
6. }
```

In the above code, we are displaying the floating value of y by using %g specifier. The %g specifier displays the output same as the input with a same precision.

Output

- **%p**

```
1. int main()
2. {
3.     int y=5;
4.     printf("Address value of y in hexadecimal form is: %p", &y);
5.     return 0;
6. }
```

Output

- **%c**

```
1. int main()
2. {
3.     char a='c';
4.     printf("Value of a is: %c", a);
5.     return 0;
6. }
```

Output

- **%s**

```
1. int main()
2. {
3.   printf("%s", "javaTpoint");
4.   return 0;
5. }
```

Output

Minimum Field Width Specifier

Suppose we want to display an output that occupies a minimum number of spaces on the screen. You can achieve this by displaying an integer number after the percent sign of the format specifier.

```
1. int main()
2. {
3.   int x=900;
4.   printf("%8d", x);
5.   printf("\n%-8d",x);
6.   return 0;
7. }
```

In the above program, %8d specifier displays the value after 8 spaces while %-8d specifier will make a value left-aligned.

Output

Now we will see how to fill the empty spaces. It is shown in the below code:

```
1. int main()
2. {
3.   int x=12;
4.   printf("%08d", x);
5.   return 0;
```

6. }

In the above program, %08d means that the empty space is filled with zeroes.

Output

Specifying Precision

We can specify the precision by using '.' (Dot) operator which is followed by integer and format specifier.

```
1. int main()
2. {
3.     float x=12.2;
4.     printf("%.2f", x);
5.     return 0;
6. }
```

Output

UNIT-II

Operators

Types of operators

D provides the ability to access and manipulate a variety of data objects: variables and data structures can be created and modified, data objects defined in the operating system kernel and user processes can be accessed, and integer, floating-point, and string constants can be declared. D provides a superset of the ANSI-C operators that are used to manipulate objects and create complex expressions. This chapter describes the detailed set of rules for types, operators, and expressions.

2.1. Identifier Names and Keywords

D identifier names are composed of upper case and lower case letters, digits, and underbars where the first character must be a letter or underbar. All identifier names beginning with an underbar (`_`) are reserved for use by the D system libraries. You should avoid using such names in your D programs. By convention, D programmers typically use mixed-case names for variables and all upper case names for constants.

D language keywords are special identifiers reserved for use in the programming language syntax itself. These names are always specified in lower case and may not be used for the names of D variables.

D Keywords

auto [*]	goto [*]	sizeof
break [*]	if [*]	static [*]
case [*]	import ^{*+}	string ⁺
char	inline	stringof ⁺
const	int	struct
continue [*]	long	switch [*]
counter ^{*+}	offsetof ⁺	this ⁺
default [*]	probe ^{*+}	translator ⁺

do [*]	provider ⁺⁺	typedef
double	register [*]	union
else [*]	restrict [*]	unsigned
enum	return [*]	void
extern	self ⁺	volatile
float	short	while [*]
for [*]	signed	xlate ⁺

D reserves for use as keywords a superset of the ANSI-C keywords. The keywords reserved for future use by the D language are marked with “^{*}”. The D compiler will produce a syntax error if you attempt to use a keyword that is reserved for future use. The keywords defined by D but not defined by ANSI-C are marked with “⁺⁺”. D provides the complete set of types and operators found in ANSI-C. The major difference in D programming is the absence of control-flow constructs. Keywords associated with control-flow in ANSI-C are reserved for future use in D.

2.2. Data Types and Sizes

D provides fundamental data types for integers and floating-point constants. Arithmetic may only be performed on integers in D programs. Floating-point constants may be used to initialize data structures, but floating-point arithmetic is not permitted in D. D provides a 32-bit and 64-bit data model for use in writing programs. The data model used when executing your program is the native data model associated with the active operating system kernel. You can determine the native data model for your system using `isainfo -b`.

The names of the integer types and their sizes in each of the two data models are shown in the following table. Integers are always represented in twos-complement form in the native byte-encoding order of your system.

D Integer Data Types

Type Name	32-bit Size	64-bit Size
char	1 byte	1 byte

short	2 bytes	2 bytes
int	4 bytes	4 bytes
long	4 bytes	8 bytes
long long	8 bytes	8 bytes

Integer types may be prefixed with the signed or unsigned qualifier. If no sign qualifier is present, the type is assumed to be signed. The D compiler also provides the type aliases listed in the following table:

D Integer Type Aliases

Type Name	Description
int8_t	1 byte signed integer
int16_t	2 byte signed integer
int32_t	4 byte signed integer
int64_t	8 byte signed integer
intptr_t	Signed integer of size equal to a pointer
uint8_t	1 byte unsigned integer
uint16_t	2 byte unsigned integer
uint32_t	4 byte unsigned integer
uint64_t	8 byte unsigned integer
uintptr_t	Unsigned integer of size equal to a pointer

These type aliases are equivalent to using the name of the corresponding base type in the previous table and are appropriately defined for each data model. For example, the type name `uint8_t` is an alias for the type `unsigned char`. See **Type and Constant Definitions** for information on how to define your own type aliases for use in your D programs.

D provides floating-point types for compatibility with ANSI-C declarations and types. Floating-point operators are not supported in D, but floating-point data objects can be traced and formatted using the `printf` function. The floating-point types listed in the following table may be used:

D Floating-Point Data Types

Type Name	32-bit Size	64-bit Size
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	16 bytes	16 bytes

D also provides the special type `string` to represent ASCII strings. Strings are discussed in more detail in **Strings**.

2.3. Constants

Integer constants can be written in decimal (12345), octal (012345), or hexadecimal (0x12345). Octal (base 8) constants must be prefixed with a leading zero. Hexadecimal (base 16) constants must be prefixed with either 0x or 0X. Integer constants are assigned the smallest type among `int`, `long`, and `long long` that can represent their value. If the value is negative, the signed version of the type is used. If the value is positive and too large to fit in the signed type representation, the unsigned type representation is used. You can apply one of the following suffixes to any integer constant to explicitly specify its D type:

u or U	unsigned version of the type selected by the compiler
l or L	long
ul or UL	unsigned long
ll or LL	long long
ull or ULL	unsigned long long

Floating-point constants are always written in decimal and must contain either a decimal point (12.345) or an exponent (123e45) or both (123.34e-5). Floating-point constants are assigned the type `double` by default. You can apply one of the following suffixes to any floating-point constant to explicitly specify its `D` type:

f or F	float
l or L	long double

Character constants are written as a single character or escape sequence enclosed in a pair of single quotes ('a'). Character constants are assigned the type `int` and are equivalent to an integer constant whose value is determined by that character's value in the ASCII character set. You can refer to **ascii(5)** for a list of characters and their values. You can also use any of the special escape sequences shown in the following table in your character constants. `D` supports the same escape sequences found in ANSI-C.

D Character Escape Sequences

<code>\a</code>	alert	<code>\\</code>	backslash
<code>\b</code>	backspace	<code>\?</code>	question mark
<code>\f</code>	formfeed	<code>\'</code>	single quote
<code>\n</code>	newline	<code>\"</code>	double quote
<code>\r</code>	carriage return	<code>\0oo</code>	octal value 0oo
<code>\t</code>	horizontal tab	<code>\xhh</code>	hexadecimal value 0xhh
<code>\v</code>	vertical tab	<code>\0</code>	null character

You can include more than one character specifier inside single quotes to create integers whose individual bytes are initialized according to the corresponding character specifiers. The bytes are read left-to-right from your character constant and assigned to the resulting integer in the order corresponding to the native endianness of your operating environment. Up to eight character specifiers can be included in a single character constant.

Strings constants of any length can be composed by enclosing them in a pair of double quotes ("hello"). A string constant may not contain a literal newline character. To create

strings containing newlines, use the `\n` escape sequence instead of a literal newline. String constants may contain any of the special character escape sequences shown for character constants above. Similar to ANSI-C, strings are represented as arrays of characters terminated by a null character (`\0`) that is implicitly added to each string constant that you declare. String constants are assigned the special D type `string`. The D compiler provides a set of special features for comparing and tracing character arrays that are declared as strings, as described in `Strings`.

2.4. Arithmetic Operators

D provides the binary arithmetic operators shown in the following table for use in your programs. These operators all have the same meaning for integers as they do in ANSI-C.

D Binary Arithmetic Operators

<code>+</code>	integer addition
<code>-</code>	integer subtraction
<code>*</code>	integer multiplication
<code>/</code>	integer division
<code>%</code>	integer modulus

Arithmetic in D may only be performed on integer operands, or on pointers, as discussed in `Pointers and Arrays`. Arithmetic may not be performed on floating-point operands in D programs. The DTrace execution environment does not take any action on integer overflow or underflow. You must check for these conditions yourself in situations where overflow and underflow can occur.

The DTrace execution environment does automatically check for and report division by zero errors resulting from improper use of the `/` and `%` operators. If a D program executes an invalid division operation, DTrace will automatically disable the affected instrumentation and report the error. Errors detected by DTrace have no effect on other DTrace users or on the operating system kernel, so you don't need to worry about causing any damage if your D program inadvertently contains one of these errors.

In addition to these binary operators, the `+` and `-` operators may also be used as unary operators as well; these operators have higher precedence than any of the binary arithmetic operators. The order of precedence and associativity properties for all the D operators is presented in **D Operator Precedence and Associativity**. You can control precedence by grouping expressions in parentheses (`()`).

2.5. Relational Operators

D provides the binary relational operators shown in the following table for use in your programs. These operators all have the same meaning as they do in ANSI-C.

D Relational Operators

<	left-hand operand is less than right-operand
<=	left-hand operand is less than or equal to right-hand operand
>	left-hand operand is greater than right-hand operand
>=	left-hand operand is greater than or equal to right-hand operand
==	left-hand operand is equal to right-hand operand
!=	left-hand operand is not equal to right-hand operand

Relational operators are most frequently used to write D predicates. Each operator evaluates to a value of type `int` which is equal to one if the condition is true, or zero if it is false.

Relational operators may be applied to pairs of integers, pointers, or strings. If pointers are compared, the result is equivalent to an integer comparison of the two pointers interpreted as unsigned integers. If strings are compared, the result is determined as if by performing a **`strcmp(3C)`** on the two operands. Here are some example D string comparisons and their results:

"coffee" < "espresso"	... returns 1 (true)
"coffee" == "coffee"	... returns 1 (true)
"coffee" >= "mocha"	... returns 0 (false)

Relational operators may also be used to compare a data object associated with an enumeration type with any of the enumerator tags defined by the enumeration. Enumerations are a facility for creating named integer constants and are described in more detail in **Type and Constant Definitions**.

2.6. Logical Operators

D provides the following binary logical operators for use in your programs. The first two operators are equivalent to the corresponding ANSI-C operators.

D Logical Operators

&&	logical AND: true if both operands are true
	logical OR: true if one or both operands are true
^^	logical XOR: true if exactly one operand is true

Logical operators are most frequently used in writing D predicates. The logical AND operator performs short-circuit evaluation: if the left-hand operand is false, the right-hand expression is not evaluated. The logical OR operator also performs short-circuit evaluation: if the left-hand operand is true, the right-hand expression is not evaluated. The logical XOR operator does not short-circuit: both expression operands are always evaluated.

In addition to the binary logical operators, the unary ! operator may be used to perform a logical negation of a single operand: it converts a zero operand into a one, and a non-zero operand into a zero. By convention, D programmers use ! when working with integers that are meant to represent boolean values, and == 0 when working with non-boolean integers, although both expressions are equivalent in meaning.

The logical operators may be applied to operands of integer or pointer types. The logical operators interpret pointer operands as unsigned integer values. As with all logical and relational operators in D, operands are true if they have a non-zero integer value and false if they have a zero integer value.

2.7. Bitwise Operators

D provides the following binary operators for manipulating individual bits inside of integer operands. These operators all have the same meaning as in ANSI-C.

D Bitwise Operators

&	bitwise AND
	bitwise OR
^	bitwise XOR
<<	shift the left-hand operand left by the number of bits specified by the right-hand

	operand
>>	shift the left-hand operand right by the number of bits specified by the right-hand operand

The binary & operator is used to clear bits from an integer operand. The binary | operator is used to set bits in an integer operand. The binary ^ operator returns one in each bit position where exactly one of the corresponding operand bits is set.

The shift operators are used to move bits left or right in a given integer operand. Shifting left fills empty bit positions on the right-hand side of the result with zeroes. Shifting right using an unsigned integer operand fills empty bit positions on the left-hand side of the result with zeroes. Shifting right using a signed integer operand fills empty bit positions on the left-hand side with the value of the sign bit, also known as an arithmetic shift operation.

Shifting an integer value by a negative number of bits or by a number of bits larger than the number of bits in the left-hand operand itself produces an undefined result. The D compiler will produce an error message if the compiler can detect this condition when you compile your D program.

In addition to the binary logical operators, the unary ~ operator may be used to perform a bitwise negation of a single operand: it converts each zero bit in the operand into a one bit, and each one bit in the operand into a zero bit.

2.8. Assignment Operators

D provides the following binary assignment operators for modifying D variables. You can only modify D variables and arrays. Kernel data objects and constants may not be modified using the D assignment operators. The assignment operators have the same meaning as they do in ANSI-C.

D Assignment Operators

=	set the left-hand operand equal to the right-hand expression value
+=	increment the left-hand operand by the right-hand expression value
-=	decrement the left-hand operand by the right-hand expression value
*=	multiply the left-hand operand by the right-hand expression value
/=	divide the left-hand operand by the right-hand expression value

<code>%=</code>	modulo the left-hand operand by the right-hand expression value
<code> =</code>	bitwise OR the left-hand operand with the right-hand expression value
<code>&=</code>	bitwise AND the left-hand operand with the right-hand expression value
<code>^=</code>	bitwise XOR the left-hand operand with the right-hand expression value
<code><<=</code>	shift the left-hand operand left by the number of bits specified by the right-hand expression value
<code>>>=</code>	shift the left-hand operand right by the number of bits specified by the right-hand expression value

Aside from the assignment operator `=`, the other assignment operators are provided as shorthand for using the `=` operator with one of the other operators described earlier. For example, the expression `x = x + 1` is equivalent to the expression `x += 1`, except that the expression `x` is evaluated once. These assignment operators obey the same rules for operand types as the binary forms described earlier.

The result of any assignment operator is an expression equal to the new value of the left-hand expression. You can use the assignment operators or any of the operators described so far in combination to form expressions of arbitrary complexity. You can use parentheses `()` to group terms in complex expressions.

2.9. Increment and Decrement Operators

D provides the special unary `++` and `--` operators for incrementing and decrementing pointers and integers. These operators have the same meaning as in ANSI-C. These operators can only be applied to variables, and may be applied either before or after the variable name. If the operator appears before the variable name, the variable is first modified and then the resulting expression is equal to the new value of the variable. For example, the following two expressions produce identical results:

<code>x += 1;</code>	<code>y = ++x;</code>
<code>y = x;</code>	

If the operator appears after the variable name, then the variable is modified after its current value is returned for use in the expression. For example, the following two expressions produce identical results:

<code>y = x;</code>	<code>y = x--;</code>
<code>x -= 1;</code>	

You can use the increment and decrement operators to create new variables without declaring them. If a variable declaration is omitted and the increment or decrement operator is applied to a variable, the variable is implicitly declared to be of type `int64_t`.

The increment and decrement operators can be applied to integer or pointer variables. When applied to integer variables, the operators increment or decrement the corresponding value by one. When applied to pointer variables, the operators increment or decrement the pointer address by the size of the data type referenced by the pointer. Pointers and pointer arithmetic in D are discussed in **Pointers and Arrays**.

2.10. Conditional Expressions

Although D does not provide support for if-then-else constructs, it does provide support for simple conditional expressions using the `?` and `:` operators. These operators enable a triplet of expressions to be associated where the first expression is used to conditionally evaluate one of the other two. For example, the following D statement could be used to set a variable `x` to one of two strings depending on the value of `i`:

```
x = i == 0 ? "zero" : "non-zero";
```

In this example, the expression `i == 0` is first evaluated to determine whether it is true or false. If the first expression is true, the second expression is evaluated and the `?:` expression returns its value. If the first expression is false, the third expression is evaluated and the `?:` expression return its value.

As with any D operator, you can use multiple `?:` operators in a single expression to create more complex expressions. For example, the following expression would take a char variable `c` containing one of the characters 0-9, a-z, or A-Z and return the value of this character when interpreted as a digit in a hexadecimal (base 16) integer:

```
hexval = (c >= '0' && c <= '9') ? c - '0' :
(c >= 'a' && c <= 'z') ? c + 10 - 'a' : c + 10 - 'A';
```

The first expression used with `?:` must be a pointer or integer in order to be evaluated for its truth value. The second and third expressions may be of any compatible types. You may not construct a conditional expression where, for example, one path returns a string and another path returns an integer. The second and third expressions also may not invoke a tracing function such as `trace` or `printf`. If you want to conditionally trace data, use a predicate instead, as discussed in **Introduction**.

2.11. Type Conversions

When expressions are constructed using operands of different but compatible types, type conversions are performed in order to determine the type of the resulting expression. The D rules for type conversions are the same as the arithmetic conversion rules for integers in ANSI-C. These rules are sometimes referred to as the usual arithmetic conversions.

A simple way to describe the conversion rules is as follows: each integer type is ranked in the order char, short, int, long, long long, with the corresponding unsigned types assigned a rank above its signed equivalent but below the next integer type. When you construct an expression using two integer operands such as $x + y$ and the operands are of different integer types, the operand type with the highest rank is used as the result type.

If a conversion is required, the operand of lower rank is first promoted to the type of higher rank. Promotion does not actually change the value of the operand: it simply extends the value to a larger container according to its sign. If an unsigned operand is promoted, the unused high-order bits of the resulting integer are filled with zeroes. If a signed operand is promoted, the unused high-order bits are filled by performing sign extension. If a signed type is converted to an unsigned type, the signed type is first sign-extended and then assigned the new unsigned type determined by the conversion.

Integers and other types can also be explicitly cast from one type to another. In D, pointers and integers can be cast to any integer or pointer types, but not to other types. Rules for casting and promoting strings and character arrays are discussed in **Strings**. An integer or pointer cast is formed using an expression such as:

```
y = (int)x;
```

where the destination type is enclosed in parentheses and used to prefix the source expression. Integers are cast to types of higher rank by performing promotion. Integers are cast to types of lower rank by zeroing the excess high-order bits of the integer.

Because D does not permit floating-point arithmetic, no floating-point operand conversion or casting is permitted and no rules for implicit floating-point conversion are defined.

2.12. Precedence

The D rules for operator precedence and associativity are described in the following table. These rules are somewhat complex, but are necessary to provide precise compatibility with the ANSI-C operator precedence rules. The table entries are in order from highest precedence to lowest precedence.

D Operator Precedence and Associativity

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof sizeof offsetof xlate	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
^^	left to right
	left to right
?:	right to left
= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right

There are several operators in the table that we have not yet discussed; these will be covered in subsequent chapters:

sizeof	Computes the size of an object (Structs and Unions)
--------	--

offsetof	Computes the offset of a type member (Structs and Unions)
stringof	Converts the operand to a string (Strings)
xlate	Translates a data type (Translators)
unary &	Computes the address of an object (Pointers and Arrays)
unary *	Dereferences a pointer to an object (Pointers and Arrays)
-> and .	Accesses a member of a structure or union type (Structs and Unions)

The comma (,) operator listed in the table is for compatibility with the ANSI-C comma operator, which can be used to evaluate a set of expressions in left-to-right order and return the value of the rightmost expression. This operator is provided strictly for compatibility with C and should generally not be used.

The () entry in the table of operator precedence represents a function call; examples of calls to functions such as printf and trace are presented in **Introduction**. A comma is also used in D to list arguments to functions and to form lists of associative array keys. This comma is not the same as the comma operator and does not guarantee left-to-right evaluation. The D compiler provides no guarantee as to the order of evaluation of arguments to a function or keys to an associative array. You should be careful of using expressions with interacting side-effects, such as the pair of expressions i and i++, in these contexts.

The [] entry in the table of operator precedence represents an array or associative array reference. Examples of associative arrays are presented in Introduction. A special kind of associative array called an aggregation is described in Aggregations. The [] operator can also be used to index into fixed-size C arrays as well, as described in Pointers and Arrays.

Precedence and Associativity

In this tutorial, you'll learn about the precedence and associativity of operators with the help of examples.

Precedence of operators

The precedence of operators determines which operator is executed first if there is more than one [operator](#) in an expression.

Let us consider an example:

```
int x = 5 - 17 * 6;
```

In C, the precedence of * is higher than - and =. Hence, 17 * 6 is evaluated first. Then the expression involving - is evaluated as the precedence of - is higher than that of =.

Here's a table of operators precedence from higher to lower. The property of associativity will be discussed shortly.

Operators Precedence & Associativity Table

Operator	Meaning of operator	Associativity
()	Functional call	
[]	Array element reference	Left to right
->	Indirect member selection	
.	Direct member selection	
!	Logical negation	
~	Bitwise(1 's) complement	
+	Unary plus	
-	Unary minus	
++	Increment	Right to left
--	Decrement	
&	Dereference (Address)	
*	Pointer reference	
sizeof	Returns the size of an object	
(type)	Typecast (conversion)	
*	Multiply	
/	Divide	Left to right
%	Remainder	
+	Binary plus(Addition)	Left to right
-	Binary minus(subtraction)	

<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal to	Left to right
!=	Not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
?:	Conditional Operator	Right to left
=	Simple assignment	Right to left
*=	Assign product	
/=	Assign quotient	
%=	Assign remainder	
+=	Assign sum	
-=	Assign difference	
&=	Assign bitwise AND	
^=	Assign bitwise XOR	
=	Assign bitwise OR	

<<=	Assign left shift	
>>=	Assign right shift	
,	Separator of expressions	Left to right

Associativity of Operators

The associativity of operators determines the direction in which an expression is evaluated. For example,

`b = a;`

Here, the value of `a` is assigned to `b`, and not the other way around. It's because the associativity of the `=` operator is from right to left.

Also, if two operators of the same precedence (priority) are present, associativity determines the direction in which they execute.

Let us consider an example:

`1 == 2 != 3`

Here, operators `==` and `!=` have the same precedence. And, their associativity is from left to right. Hence, `1 == 2` is executed first.

The expression above is equivalent to:

`(1 == 2) != 3`

Expression

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

Let's see an example:

1. `a-b;`

In the above expression, minus character (`-`) is an operator, and `a`, and `b` are the two operands.

There are four types of expressions exist in C:

- Arithmetic expressions
- Relational expressions
- Logical expressions
- Conditional expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of a particular expression produces a specific value.

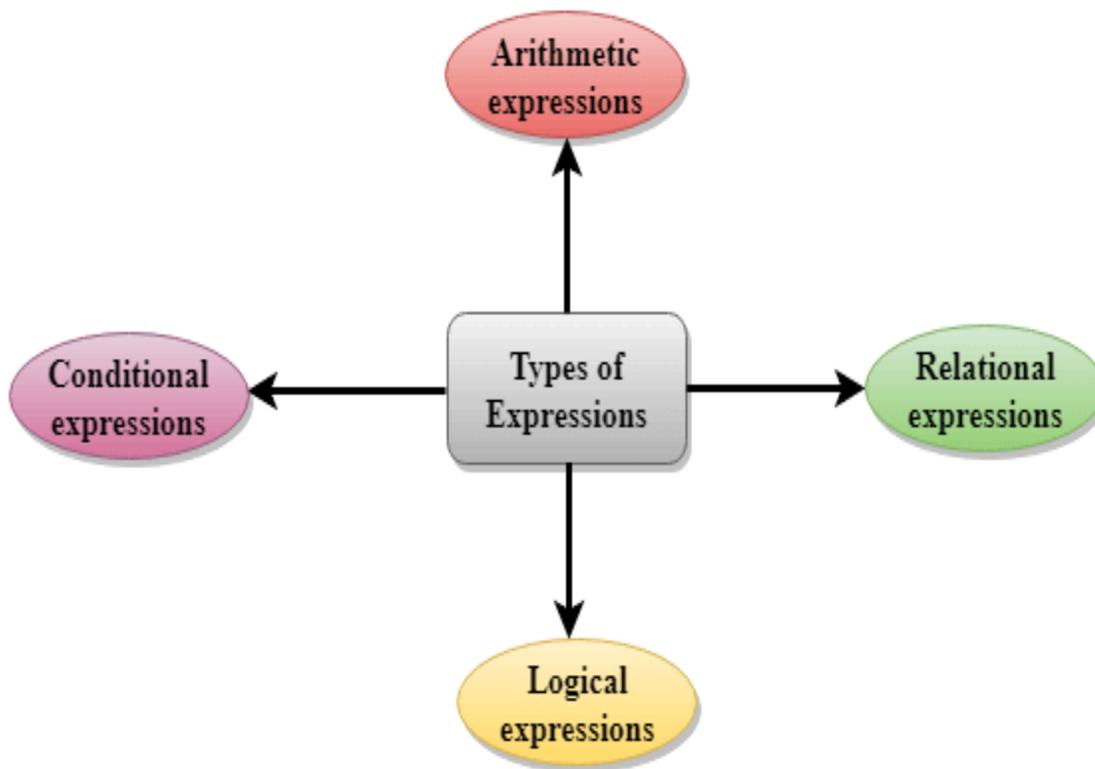
For example:

1. $x = 9/2 + a - b;$

The entire above line is a statement, not an expression. The portion after the equal is an expression.

Arithmetic Expressions

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression compute



s a value of type int, float or double.

When an expression contains only integral operands, then it is known as pure integer expression when it contains only real operands, it is known as pure real expression, and when it contains both integral and real operands, it is known as mixed mode expression.

Evaluation of Arithmetic Expressions

The expressions are evaluated by performing one operation at a time. The precedence and associativity of operators decide the order of the evaluation of individual operations.

When individual operations are performed, the following cases can be happened:

- When both the operands are of type integer, then arithmetic will be performed, and the result of the operation would be an integer value. For example, $3/2$ will yield 1 not 1.5 as the fractional part is ignored.
- When both the operands are of type float, then arithmetic will be performed, and the result of the operation would be a real value. For example, $2.0/2.0$ will yield 1.0, not 1.
- If one operand is of type integer and another operand is of type real, then the mixed arithmetic will be performed. In this case, the first operand is converted

into a real operand, and then arithmetic is performed to produce the real value. For example, 6/2.0 will yield 3.0 as the first value of 6 is converted into 6.0 and then arithmetic is performed to produce 3.0.

Let's understand through an example.

$$6*2/(2+1 * 2/3 + 6) + 8 * (8/4)$$

Evaluation of expression	Description of each operation
$6*2/(2+1 * 2/3 + 6) + 8 * (8/4)$	An expression is given.
$6*2/(2+2/3 + 6) + 8 * (8/4)$	2 is multiplied by 1, giving value 2.
$6*2/(2+0+6) + 8 * (8/4)$	2 is divided by 3, giving value 0.
$6*2/ 8+ 8 * (8/4)$	2 is added to 6, giving value 8.
$6*2/8 + 8 * 2$	8 is divided by 4, giving value 2.
$12/8 + 8 * 2$	6 is multiplied by 2, giving value 12.
$1 + 8 * 2$	12 is divided by 8, giving value 1.
$1 + 16$	8 is multiplied by 2, giving value 16.
17	1 is added to 16, giving value 17.

Relational Expressions

- A relational expression is an expression used to compare two operands.
- It is a condition which is used to decide whether the action should be taken or not.
- In relational expressions, a numeric value cannot be compared with the string value.
- The result of the relational expression can be either zero or non-zero value. Here, the zero value is equivalent to a false and non-zero value is equivalent to true.

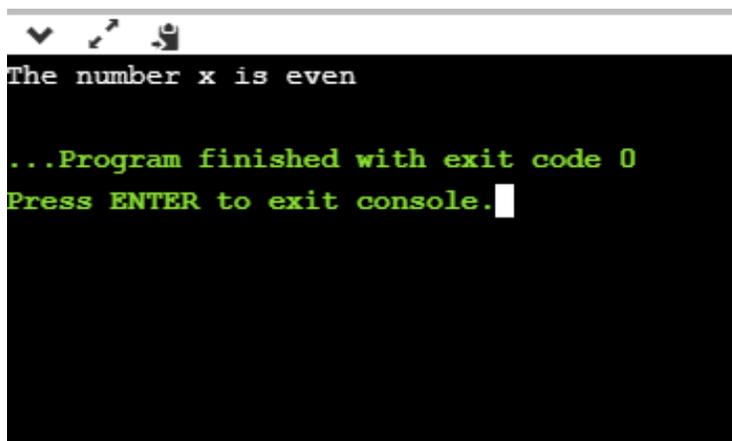
Relational Expression	Description
-----------------------	-------------

<code>x%2 == 0</code>	This condition is used to check whether the x is an even number or not. The relational expression results in value 1 if x is an even number otherwise results in value 0.
<code>a!=b</code>	It is used to check whether a is not equal to b. This relational expression results in 1 if a is not equal to b otherwise 0.
<code>a+b == x+y</code>	It is used to check whether the expression "a+b" is equal to the expression "x+y".
<code>a>=9</code>	It is used to check whether the value of a is greater than or equal to 9.

Let's see a simple example:

```
1. #include <stdio.h>
2. int main()
3. {
4.
5.     int x=4;
6.     if(x%2==0)
7.     {
8.         printf("The number x is even");
9.     }
10.    else
11.    printf("The number x is not even");
12.    return 0;
13.}
```

Output



```
The number x is even
...Program finished with exit code 0
Press ENTER to exit console.█
```

Logical Expressions

A logical expression is an expression that computes either a zero or non-zero value.

It is a complex test condition to take a decision.

Let's see some example of the logical expressions.

Logical Expressions	Description
<code>(x > 4) && (x < 6)</code>	It is a test condition to check whether the x is greater than 4 and x is less than 6. The result of the condition is true only when both the conditions are true.
<code>x > 10 y < 11</code>	It is a test condition used to check whether x is greater than 10 or y is less than 11. The result of the test condition is true if either of the conditions holds true value.
<code>! (x > 10) && (y = = 2)</code>	It is a test condition used to check whether x is not greater than 10 and y is equal to 2. The result of the condition is true if both the conditions are true.

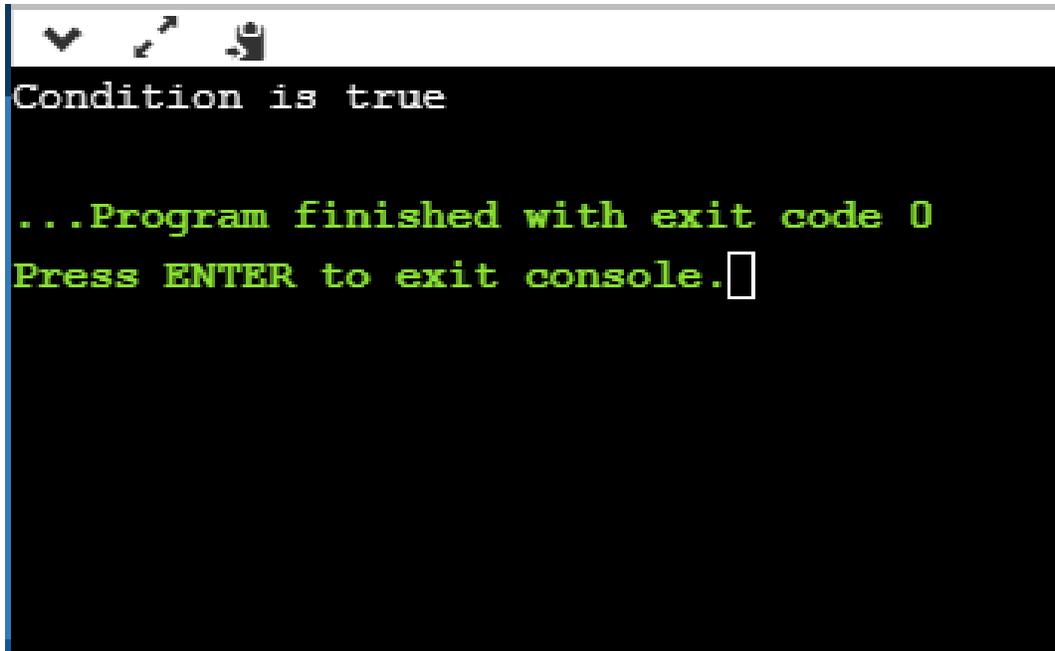
Let's see a simple program of "&&" operator.

```
#include <stdio.h>

int main()
{
    int x = 4;
    int y = 10;
    if ( ( x < 10 ) && ( y > 5 ) )
    {
        printf("Condition is true");
    }
}
```

```
else
printf("Condition is false");
return 0;
}
```

Output



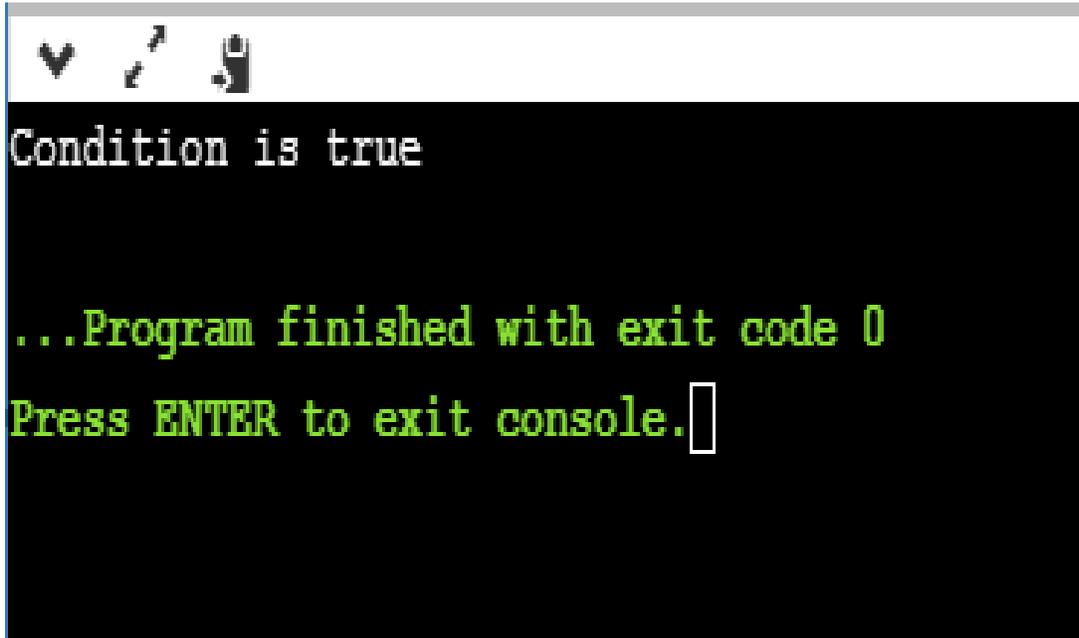
```
Condition is true

...Program finished with exit code 0
Press ENTER to exit console.█
```

Let's see a simple example of "||" operator

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x = 4;
5.     int y = 9;
6.     if ( (x <6) || (y>10))
7.     {
8.         printf("Condition is true");
9.     }
10. else
11.     printf("Condition is false");
12. return 0;
13.}
```

Output



```
Condition is true

...Program finished with exit code 0
Press ENTER to exit console.
```

Conditional Expressions

- A conditional expression is an expression that returns 1 if the condition is true otherwise 0.
- A conditional operator is also known as a ternary operator.

The Syntax of Conditional operator

Suppose **exp1**, **exp2** and **exp3** are three expressions.

exp1 ? exp2 : exp3

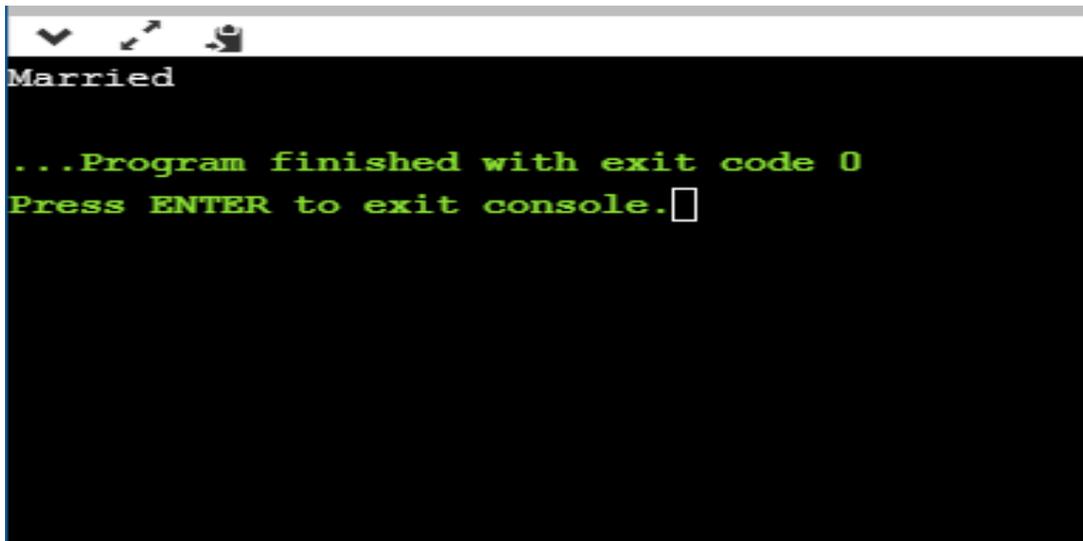
The above expression is a conditional expression which is evaluated on the basis of the value of the **exp1** expression. If the condition of the expression **exp1** holds true, then the final conditional expression is represented by **exp2** otherwise represented by **exp3**.

Let's understand through a simple example.

1. `#include<stdio.h>`
2. `#include<string.h>`
3. `int main()`
4. `{`
5. `int age = 25;`
6. `char status;`

```
7.  status = (age>22) ? 'M': 'U';
8.  if(status == 'M')
9.  printf("Married");
10. else
11.  printf("Unmarried");
12.  return 0;
13.}
```

Output



```
Married
...Program finished with exit code 0
Press ENTER to exit console. □
```

Statement and types of statements

Statement

C programs are collection of Statements, statements is an executable part of the program it will do some action. In general all arithmetic actions and logical actions are falls under Statements Categories anyway there are few Statement categories

Types of Statements

Like C++ it contains three types of statements.

1. Simple Statements
2. Compound Statements
3. Control Statements

We'll be covering the following topics in this tutorial:

- Simple Statements
- Compound Statements:
- Control Statements
- (b) Jump Statements
- (c) Selection Statements

Simple Statements

A simple statement is any expression that terminates with a semicolon. For example

```
var1 = var2 + var3;
```

```
var3 = var1++;
```

```
var3++;
```

Compound Statements:

Related statements can be grouped together in braces to form a compound statement or block. For example:

```
{
int i = 4;
Console.WriteLine (i);
i++;
}
```

Semantically, a block behave like a statement and can be used anywhere a single statement is allowed. There is no semicolon after the closing braces. Any variable declared in a block remain in scope up to the closing brace. Once the block is exited, the block variables cease to exist. Blocking improves readability of program code and can help make your program easier to control and debug.

Control Statements

Again control statements are divided into three statements

- (a) Loop statements
- (b) Jump statements
- (c) Selection statements

(a) Loop Statements

C# provides a number of the common loop statements:

- while
- do-while

- for
- foreach

while loops

Syntax: while (expression) statement[s]

A 'while' loop executes a statement, or a block of statements wrapped in curly braces, repeatedly until the condition specified by the Boolean expression returns false. For instance, the following code.

```
int a =0;
While (a < 3) {
System.Console. WriteLine (a);
a++;
}
```

Produces the following output:

```
0
1
2
```

do-while loops

Syntax: do statement [s] while (expression)

A 'do-while' loop is just like a 'while' loop except that the condition is evaluated after the block of code specified in the 'do' clause has been run. So even where the condition is initially false, the block runs once. For instance, the following code outputs '4':

```
Int a = 4;
do
{
System.Console. WriteLine (a);
a++;
} while (a < 3);
```

for loops

Syntax: for (statement1; expression; statement2) statement[s]3

The 'for' clause contains three part. Statement1 is executed before the loop is entered.

The loop which is then executed corresponds to the following 'while' loop:

Statement1

```
while (expression) {statement[s]3; statement2}
```

'for' loops tend to be used when one needs to maintain an iterator value. Usually, as in the following example, the first statement initializes the iterator, the condition evaluates it against an end value, and the second statement changes the iterator value.

```
for (int a =0; a<5; a++)  
{  
system.console.WriteLine(a);  
}
```

foreach loops

syntax:foreach (variable1 in variable2) statement[s]

The 'foreach' loop is used to iterate through the values contained by any object which implements the IEnumerable interface. When a 'foreach' loop runs, the given variable1 is set in turn to each value exposed by the object named by variable2. As we have seen previously, such loops can be used to access array values. So, we could loop through the values of an array in the following way:

```
int[] a = new int[]{1,2,3};  
foreach (int b in a)  
system.console.WriteLine(b);
```

The main drawback of 'foreach' loops is that each value extracted (held in the given example by the variable 'b') is read-only

(b) Jump Statements

The jump statements include

- break
- continue
- goto
- return

- throw

break

The following code gives an example – of how it could be used. The output of the loop is the numbers from 0 to 4.

```
int a = 0;
while (true)
{
system.console.WriteLine(a);
a++;
if (a == 5)
break;
}
```

Continue

The 'continue' statement can be placed in any loop structure. When it executes, it moves the program counter immediately to the next iteration of the loop. The following code example uses the 'continue' statement to count the number of values between 1 and 100 inclusive that are not multiples of seven. At the end of the loop the variable y holds the required value.

```
int y = 0;
for (int x=1; x<101; x++)
{
if ((x % 7) == 0)
continue;
y++;
}
```

Goto

The 'goto' statement is used to make a jump to a particular labeled part of the program code. It is also used in the 'switch' statement described below. We can use a 'goto'

statement to construct a loop, as in the following example (but again, this usage is not recommended):

```
int a = 0;
start:
system.console.WriteLine(a);
a++;
if (a < 5)
goto start;
```

(c) Selection Statements

C# offers two basic types of selection statement:

- if-else
- switch-default

if-else

'if-else' statements are used to run blocks of code conditionally upon a boolean expression evaluating to true. The 'else' clause, present in the following example, is optional.

```
if (a == 5)
system.console.WriteLine("A is 5");
else
system.console.WriteLine("A is not 5");
```

'if' statements can also be emulated by using the conditional operator. The conditional operator returns one of two values, depending upon the value of a boolean expression. To take a simple example,

```
int i = (myBoolean) ? 1 : 0;
```

The above code sets *i* to 1 if *myBoolean* is true, and sets *i* to 0 if *myBoolean* is false. The 'if' statement in the previous code example could therefore be written like this:

```
system.console.WriteLine (a==5 ? "A is 5" "A is not 5");
```

switch-default

'switch' statements provide a clean way of writing multiple if – else statements. In the following example, the variable whose value is in question is 'a'. If 'a' equals 1, then the

output is 'a>0'; if a equals 2, then the output is 'a> 1 and a>0'. Otherwise, it is reported that the variable is not set.

```
switch(a)
{
case 2:
Console.WriteLine("a>1 and ");
goto case 1;
case 1:
console.WriteLine("a>0");
break;
default:
console.WriteLine("a is not set");
break;
}
```

Each case (where this is taken to include the 'default' case) will either have code specifying a conditional action, or no such code. Where a case does have such code, the code must (unless the case is the last one in the switch statement) end with one of the following statements:

```
break; .
goto case k; (where k is one of the cases specified)
goto default;
```

From the above it can be seen that C# 'switch' statements lack the default 'fall through' behavior found in C++ and Java. However, program control does fall through wherever a case fails to specify any action. The following example illustrates this point; the response "a>0"

is given when a is either 1 or 2.

```
switch(a)
{
case 1:
case 2:
Console.WriteLine("a>0");
break;
default:
```

```
Console.WriteLine("a is not set");
break;
}
```

Build in Operators and function

Console based I/O and related built in I/O function:

printf():- The printf() function outputs a formatted string.

The arg1, arg2, ++ parameters will be inserted at percent (%) signs in the main string. This function works "step-by-step". At the first % sign, arg1 is inserted, at the second % sign, arg2 is inserted, etc.

Note: If there are more % signs than arguments, you must use placeholders. A placeholder is inserted after the % sign, and consists of the argument- number and "\$". See example two.

Tip: Related functions: sprintf(), vprintf(), vsprintf(), fprintf() and vfprintf()

Syntax

```
printf(format,arg1,arg2,arg++)
```

Parameter Values

Parameter	Description
format	Required. Specifies the string and how to format the variables in it. Possible format values: <ul style="list-style-type: none">• %% - Returns a percent sign• %b - Binary number• %c - The character according to the ASCII value• %d - Signed decimal number (negative, zero or positive)• %e - Scientific notation using a lowercase (e.g. 1.2e+2)

- %E - Scientific notation using a uppercase (e.g. 1.2E+2)
- %u - Unsigned decimal number (equal to or greater than zero)
- %f - Floating-point number (local settings aware)
- %F - Floating-point number (not local settings aware)
- %g - shorter of %e and %f
- %G - shorter of %E and %f
- %o - Octal number
- %s - String
- %x - Hexadecimal number (lowercase letters)
- %X - Hexadecimal number (uppercase letters)

Additional format values. These are placed between the % and the letter (example %.2f):

- + (Forces both + and - in front of numbers. By default, only negative numbers are marked)
- ' (Specifies what to use as padding. Default is space. Must be used together with the width specifier. Example: %'x20s (this uses "x" as padding)
- - (Left-justifies the variable value)
- [0-9] (Specifies the minimum width held of to the variable value)
- .[0-9] (Specifies the number of decimal digits or maximum string length)

Note: If multiple additional format values are used, they must be in the same order as above.

arg1	Required. The argument to be inserted at the first %-sign in the format string
arg2	Optional. The argument to be inserted at the second %-sign in the format string
arg++	Optional. The argument to be inserted at the third, fourth, etc. %-sign in the format string

Technical Details

Return Value: Returns the length of the outputted string

scanf():- The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

1. printf("format string",argument_list);

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

1. scanf("format string",argument_list);

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

1. #include<stdio.h>
2. **int** main(){
3. **int** number;
4. printf("enter a number:");
5. scanf("%d",&number);
6. printf("cube of number is:%d ",number*number*number);
7. **return** 0;
8. }

Output

```
enter a number:5
cube of number is:125
```

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number*number*number)** statement prints the cube of number on the console.

Program to print sum of 2 numbers

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

1. #include<stdio.h>
2. **int** main(){
3. **int** x=0,y=0,result=0;
- 4.
5. printf("enter first number:");
6. scanf("%d",&x);
7. printf("enter second number:");
8. scanf("%d",&y);
- 9.
10. result=x+y;
11. printf("sum of 2 numbers:%d ",result);
- 12.
13. **return** 0;
14. }

Output

```
enter first number:9
enter second number:9
sum of 2 numbers:18
```

getch():- getch() function is a function in C programming language which waits for any character input from keyboard. Please find below the description and syntax for above file handling function.

File operation	Declaration & Description
getch()	Declaration: int getch(void); This function waits for any character input from keyboard. But, it won't echo the input character on to the output

screen

EXAMPLE PROGRAM FOR GETCH() FUNCTION IN C PROGRAMMING LANGUAGE:

This is a simple Hello World! C program. After displaying Hello World! in output screen, this program waits for any character input from keyboard. After any single character is typed/pressed, this program returns 0. But, please note that, getch() function will just wait for any keyboard input (and press ENTER). It won't display the given input character in output screen.

C



```
1 #include <stdio.h>
2 int main()
3 {
4     /* Our first simple C basic program */
5     printf("Hello World! ");
6     getch();
7     return 0;
8 }
```

OUTPUT:

Hello World!

OTHER INBUILT FILE HANDLING FUNCTIONS IN C PROGRAMMING LANGUAGE:

C programming language offers many other inbuilt functions for handling files. They are given below. Please click on each function name below to know more details, example programs, output for the respective file handling function.

File handling functions	Description
fopen ()	fopen () function creates a new file or opens an existing file.
fclose ()	fclose () function closes an opened file.
getw ()	getw () function reads an integer from file.
putw ()	putw () functions writes an integer to file.
fgetc ()	fgetc () function reads a character from file.
fputc ()	fputc () functions write a character to file.
gets ()	gets () function reads line from keyboard.
puts ()	puts () function writes line to o/p screen.
fgets ()	fgets () function reads string from a file, one line at a time.
fputs ()	fputs () function writes string to a file.
feof ()	feof () function finds end of file.
fgetchar ()	fgetchar () function reads a character from keyboard.
fprintf ()	fprintf () function writes formatted data to a file.
fscanf ()	fscanf () function reads formatted data from a file.

fputchar ()	fputchar () function writes a character onto the output screen from keyboard input.
fseek ()	fseek () function moves file pointer position to given location.
SEEK_SET	SEEK_SET moves file pointer position to the beginning of the file.
SEEK_CUR	SEEK_CUR moves file pointer position to given location.
SEEK_END	SEEK_END moves file pointer position to the end of file.
ftell ()	ftell () function gives current position of file pointer.
rewind ()	rewind () function moves file pointer position to the beginning of the file.
getc ()	getc () function reads character from file.
getch ()	getch () function reads character from keyboard.
getche ()	getche () function reads character from keyboard and echoes to o/p screen.
getchar ()	getchar () function reads character from keyboard.
putc ()	putc () function writes a character to file.
putchar ()	putchar () function writes a character to screen.

printf ()	printf () function writes formatted data to screen.
sprintf ()	sprintf () function writes formatted output to string.
scanf ()	scanf () function reads formatted data from keyboard.
sscanf ()	sscanf () function Reads formatted input from a string.
remove ()	remove () function deletes a file.
fflush ()	fflush () function flushes a file.

getchar():- The C library function **int getchar(void)** gets a character (an unsigned char) from stdin. This is equivalent to **getc** with stdin as its argument.

Declaration

Following is the declaration for getchar() function.

```
int getchar(void)
```

Parameters

- **NA**

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getchar() function.

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

```
int main () {
    char c;

    printf("Enter character: ");
    c = getchar();

    printf("Character entered: ");
    putchar(c);

    return(0);
}
```

Let us compile and run the above program that will produce the following result –

```
Enter character: a
Character entered: a
```

putchar():- The C library function **int putchar(int char)** writes a character (an unsigned char) specified by the argument char to stdout.

Declaration

Following is the declaration for putchar() function.

```
int putchar(int char)
```

Parameters

- **char** – This is the character to be written. This is passed as its int promotion.

Return Value

This function returns the character written as an unsigned char cast to an int or EOF on error.

Example

The following example shows the usage of putchar() function.

```
#include <stdio.h>

int main () {
    char ch;

    for(ch = 'A' ; ch <= 'Z' ; ch++) {
        putchar(ch);
    }
}
```

```
}  
  
return(0);  
}
```

Let us compile and run the above program that will produce the following result –
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Concept of header files

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms –

#include <file>

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the **-I** option while compiling your source code.

#include "file"

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the **-I** option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting

from the included file, followed by the output that comes from the text after the #include directive. For example, if you have a header file header.h as follows –

```
char *test (void);
```

and a main program called program.c that uses the header file, like this –

```
int x;  
#include "header.h"  
  
int main (void) {  
    puts (test ());  
}
```

the compiler will see the same token stream as it would if program.c read.

```
int x;  
char *test (void);  
  
int main (void) {  
    puts (test ());  
}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this –

```
#ifndef HEADER_FILE  
#define HEADER_FILE  
  
the entire header file file  
  
#endif
```

This construct is commonly known as a wrapper #ifndef. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows –

```

#if SYSTEM_1
    # include "system_1.h"
#elif SYSTEM_2
    # include "system_2.h"
#elif SYSTEM_3
    ...
#endif

```

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of #include, you simply put a macro name there –

```

#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H

```

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the #include had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.

Preprocessor directives

The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation. Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

Preprocessor Directives in C#

The following table lists the preprocessor directives available in C# –

Sr.No.	Preprocessor Directive & Description
1	<p>#define</p> <p>It defines a sequence of characters, called symbol.</p>
2	<p>#undef</p>

	It allows you to undefine a symbol.
3	#if It allows testing a symbol or symbols to see if they evaluate to true.
4	#else It allows to create a compound conditional directive, along with #if.
5	#elif It allows creating a compound conditional directive.
6	#endif Specifies the end of a conditional directive.
7	#line It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.
8	#error It allows generating an error from a specific location in your code.
9	#warning It allows generating a level one warning from a specific location in your code.
10	#region It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.
11	#endregion It marks the end of a #region block.

The #define Preprocessor

The #define preprocessor directive creates symbolic constants.

#define lets you define a symbol such that, by using the symbol as the expression passed to the #if directive, the expression evaluates to true. Its syntax is as follows –

#define symbol

The following program illustrates this –

```
#define PI
using System;

namespace PreprocessorDAppl {
    class Program {
        static void Main(string[] args) {
            #if (PI)
                Console.WriteLine("PI is defined");
            #else
                Console.WriteLine("PI is not defined");
            #endif
            Console.ReadKey();
        }
    }
}
```

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

PI is defined

Conditional Directives

You can use the #if directive to create a conditional directive. Conditional directives are useful for testing a symbol or symbols to check if they evaluate to true. If they do evaluate to true, the compiler evaluates all the code between the #if and the next directive.

Syntax for conditional directive is –

#if symbol [operator symbol]...

Where, symbol is the name of the symbol you want to test. You can also use true and false or prepend the symbol with the negation operator.

The operator symbol is the operator used for evaluating the symbol. Operators could be either of the following –

- == (equality)
- != (inequality)

- && (and)
- || (or)

You can also group symbols and operators with parentheses. Conditional directives are used for compiling code for a debug build or when compiling for a specific configuration. A conditional directive beginning with a **#if** directive must explicitly be terminated with a **#endif** directive.

The following program demonstrates use of conditional directives –

```
#define DEBUG
#define VC_V10
using System;

public class TestClass {
    public static void Main() {
        #if (DEBUG && !VC_V10)
            Console.WriteLine("DEBUG is defined");
        #elif (!DEBUG && VC_V10)
            Console.WriteLine("VC_V10 is defined");
        #elif (DEBUG && VC_V10)
            Console.WriteLine("DEBUG and VC_V10 are defined");
        #else
            Console.WriteLine("DEBUG and VC_V10 are not defined");
        #endif
        Console.ReadKey();
    }
}
```

When the above code is compiled and executed, it produces the following result –
DEBUG and VC_V10 are defined

#include

The **#include** preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of **#include** directive, we provide information to the preprocessor where to look for the header files. There are two variants to use **#include** directive.

1. **#include <filename>**
2. **#include "filename"**

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

1. #include<stdio.h>
2. **int** main(){
3. printf("Hello C");
4. **return** 0;
5. }

Output:

Hello C

#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

#define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

1. #define token value

Let's see an example of #define to define a constant.

1. #include <stdio.h>
2. #define PI 3.14
3. main() {

```
4. printf("%f",PI);
5. }
```

Output:

```
3.140000
```

Let's see an example of #define to create a macro.

```
1. #include <stdio.h>
2. #define MIN(a,b) ((a)<(b)?(a):(b))
3. void main() {
4.     printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
5. }
```

Output:

```
Minimum between 10 and 20 is: 10
```

Control structures

Decision making structures:-

If:- Simplest control structure available in C is 'if' statement. The 'if' control structure consists of 'if' statement followed by an expression enclosed in parentheses. The 'if' statement evaluates the expression and if it evaluates to true, it will result in execution of immediate expression/statement following. Syntax of 'if' statement is given below.

```
1. if(expression)
2.     statement
```

The statement 'statement' will be executed if the 'expression' is evaluated to true. If the 'expression' is evaluated to false, 'statement' will not be executed.

The above syntax allows only execution of the immediately following statement. What if you want to execute more than one statement based on the evaluation of condition? In that case the set of statements that you want to execute needs to be enclosed in curly braces as given below.

```
1. if(expression)
2. {
3.     statement1
4.     statement2
5.     ..
```

```
6.  ..
7.  statementN
8.  }
```

It is good practice to enclose the statements you want to execute in braces even if there is only a single statement to be executed. This clears the confusion of which statements are executed if the expression of an 'if' statement is satisfied.

An important thing to remember while using 'if' statement is that every expression always return either true or false (0 is considered as false and others are considered as true), even assignment statement inside the conditional expression will not throw any compilation error. Sometimes programmers may mistakenly write `if (a = 10)` instead of `if (a == 10)`. In this case, first the value '10' will be assigned to variable 'a' and then 'a' is tested to see if it's zero (false) or non-zero(true). So the 'if' statement block will always be executed no matter what the value of 'a' and to worsen things this will not generate any compilation error. To avoid this problem, it's better to use `if (0 == a)` instead of `if(a == 0)`, as forgetting one '=' in the first case will produce compilation error.

If-else:- Sometimes using only 'if' statement is not enough. Suppose we have a program to print whether the input number (say variable 'i') is odd or even. The decision part of the program is as follows

```
1. if ((i % 2) == 1)
2. {
3.     printf("Odd");
4. }
```

This way we can decide whether the number is odd or not. But to decide whether it is even, we have to again write almost same expression with changing the value 1 to 0. But C provides us a different solution to reduce the number of lines of codes that you need to write. It is called 'if ... else' statement, which is used to handle the false case of the 'if' expression.

Syntax of 'if ... else' statement is as given below:

```
1. if (expression)
2. {
3.     statement block1
4. } else {
5.     statement block2
6. }
```

In the above syntax, if the 'expression' evaluates to true, 'statement block1' will be executed. If the 'expression' evaluates to false, 'statement block2' will be executed.

Nested If-else:- Nested 'if' statement is similar to 'if-else' statement with the addition of more than one 'if-else'block. The syntax is given below.

```
1. if (expression 1)
2. {
3.     statement block 1
4. }
5. else if (expression 2)
6. {
7.     statement block 2
8. }
9. ...
10. ...
11. else if (expression n)
12. {
13.     statement block n
14. }
15. else
16. {
17.     statement block n+1
18. }
```

While executing nested 'if' statement, first 'expression 1' will be checked and if it evaluates to true 'statement block1' will be executed. If it is false, then 'expression 2' will be tested and it proceeds further in the same fashion.

Switch:- The alternative of nested 'if' statement is called 'switch' statement, which has different statement blocks for different cases. Basically 'switch' statement tries to match evaluated value of an expression against different cases. Syntax of 'switch' statement is as follows.

```
1. switch(expression)
2. {
3.     case value 1:
4.         statement block 1;
5.         break;
```

```
6.  
7.  case value 2:  
8.     statement block 2;  
9.     break;  
10.  
11.  ...  
12.  ...  
13.  
14.  case value n:  
15.     statement block n;  
16.     break;  
17.  
18.  default:  
19.     default statement block;  
20.     break;  
21. }
```

The expression 'expression' is evaluated first and the value is checked with different case values (value 1, value 2, ... etc. If one of the case values matches the result of expression, then respective statement block is executed. If none is matched, then the statement block of 'default' case is executed. You may notice another keyword in the switch statement which is break. If break keyword is omitted, all the the statements after that will be executed even if the corresponding case values do not match the value of expression. This is called a *'fall through'* statement. For example, if 'break' is omitted and say, case 'value 1' is matched, then after execution of 'statement block 1' the code will execute statement block 2, 3 and so on until a 'break' statement is reached.

Rules for Writing 'switch' Statements in C Language

- The values of case label must be unique, else compiler will throw error
- The values of case should be an integral constant (integer or character). If any variable is specified as part of a case, compiler will throw error, but a 'const' variable is allowed.
- The 'default' case is optional and it can be placed anywhere in the switch block.
- 'switch' statements can be nested; that means you can have another 'switch' statement as part of a statement block associated with a case.
- The statement block of a case is optional and it can be shared by more than one cases.

Loop Control structures:- Most commonly used loop statement in C programming language is 'for' loop. Syntax of 'for' loop is as given below.

```
1. for(initialize expression; test condition; index modification expression)
2. {
3.     Statement block
4. }
```

The 'for' loop generally includes three expressions separated by semicolon (;),

1. Init expression, which is executed first. You can initialize the loop control variable at this point. This expression is optional.
2. Test condition is evaluated next. The body of loop is executed only if this condition evaluates to true. If this condition evaluates to false, the control moves to the next line after 'for' loop
3. Index modification expression is executed after body of the loop. You can update the loop control variable at this location. After the execution of index modification expression, loop test condition is evaluated again and this procedure follows until the test condition is evaluated to false

While:- After 'for' loop, let's look at 'while' loop in C language.

Unlike 'for' loop, 'while' loop contains only a single expression and it is used as the test condition. The initialization and modification expressions of 'for' loop is omitted in 'while' loop. The general syntax of 'while' loop is given below.

```
1. while(expression)
2. {
3.     statement block
4. }
```

If the 'expression' evaluates to true, i.e. it evaluates to a non-zero value, code in the 'statement block' gets executed. And after execution of 'statement block', the expression

is evaluated again to check its truth value and this process continues until the expression evaluates to false.

So if you place a non-zero value as the 'expression', 'while' loop executes infinitely unless you have an explicit 'break' statement as part of 'statement block'.

```
1. while(1) {  
2.     statement block  
3. }
```

In the above code snippet, the 'statement block' gets executed infinitely.

Do-while:- 'do...while' loop is similar to 'while' except that the loop test expression is evaluated after execution of the loop body. The syntax of 'do...while' is given below.

```
1. do  
2. {  
3.     Statement block  
4. } while(expression);
```

Also, note that there is a semicolon (;) that is followed by while statement. Since the condition expression of 'do...while' is evaluated after the execution of loop body, it is guaranteed that the loop body will be executed at least once even if the test condition is false.

For :- A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop syntax in C is as follows:

```
for (initial value; condition; incrementation or decrementation )  
{  
    statements;  
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

Following program illustrates the for loop in C programming example:

```
#include<stdio.h>  
int main()
```

```

{
    int number;
    for(number=1;number<=10;number++)    //for loop to print 1-10 numbers
    {
        printf("%d\n",number);          //to print the number
    }
    return 0;
}

```

Output:

```

1
2
3
4
5
6
7
8
9
10

```

The above program prints the number series from 1-10 using for loop.

Nested for loop:- C supports nesting of loops in C. **Nesting of loops** is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a '**for**' loop.

Syntax of Nested loop

1. Outer_loop
2. {
3. Inner_loop
4. {
5. // inner loop statements.
6. }
7. // outer loop statements.
8. }

Outer_loop and **Inner_loop** are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

Nested for loop

The nested for loop means any type of loop which is defined inside the 'for' loop.

1. **for** (initialization; condition; update)
2. {
3. **for**(initialization; condition; update)
4. {
5. // inner loop statements.
6. }
7. // outer loop statements.
8. }

Example of nested for loop

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** n;// variable declaration
5. printf("Enter the value of n :");
6. // Displaying the n tables.
7. **for**(**int** i=1;i<=n;i++) // outer loop
8. {
9. **for**(**int** j=1;j<=10;j++) // inner loop
10. {
11. printf("%d\t",i*j); // printing the value.
12. }
13. printf("\n");
14. }

Other statements:- break:- The keyword 'break' allow us to exit the entire 'switch' statement or jump out of a loop without waiting for the test condition to be false.

Continue:- In the previous section we've seen that how to use 'break' statement to exit the loop on particular condition. Now let's see how to skip a particular cycle of a loop? You can use 'continue' statement inside a loop block for that purpose. The 'continue' statement does not terminate the loop. It just skips the remaining statements in the body of the loop for the current pass and then next pass is started. To use 'continue', you just have to include the keyword 'continue' followed by a semicolon.

Goto:- The 'goto' statement is used to transfer the control of execution of the program to another part of the program where a particular label is defined. The 'goto' statement is used alongside a label and has the general form goto label;

The label is the identifier which defines where the control should go on the execution of 'goto' statement. The target statement block must be labelled like - label: statement /statement block

Exit :- In the C Programming Language, the **exit function** calls all functions registered with atexit and terminates the program. File buffers are flushed, streams are closed, and temporary files are deleted.

Syntax

The syntax for the exit function in the C Language is:

```
void exit(int status);
```

Parameters or Arguments

status

Indicates whether the program terminated normally. It can be one of the following:

Value	Description
EXIT_SUCCESS	Successful termination
0	Successful termination
EXIT_FAILURE	Unsuccessful termination

Returns

The exit function does not return anything.

Required Header

In the C Language, the required header for the exit function is:

```
#include <stdlib.h>
```

Applies To

In the C Language, the exit function can be used in the following versions:

- ANSI/ISO 9899-1990

Similar Functions

Other C functions that are similar to the exit function:

- abort function <stdlib.h>

See Also

Other C functions that are noteworthy when dealing with the exit function:

- atexit function <stdlib.h>

UNIT-III

Introduction to problem solving

Concept:

problem solving :- In the workplace, many of the communications tasks you perform are designed to solve a problem or improve a situation. Whether you are doing work for a client, for your employer, with your team, or for someone else, you will typically use some sort of design process to tackle and solve the problem. A clearly-articulated design process provides you with a clear, step-by-step plan for finding the best solution for your situation.

Take a moment to search the Internet for the term “design process” and look at “images.” You will find many variations. Have a look at several of them and see if you can find a common pattern.

One commonality you will likely find in examining other people’s design process diagrams is this: the first step in designing any solution is to clearly define the problem. Figure 1.1.1 shows NASA’s basic design process. Think about the kind of communication that each step of this process might entail.

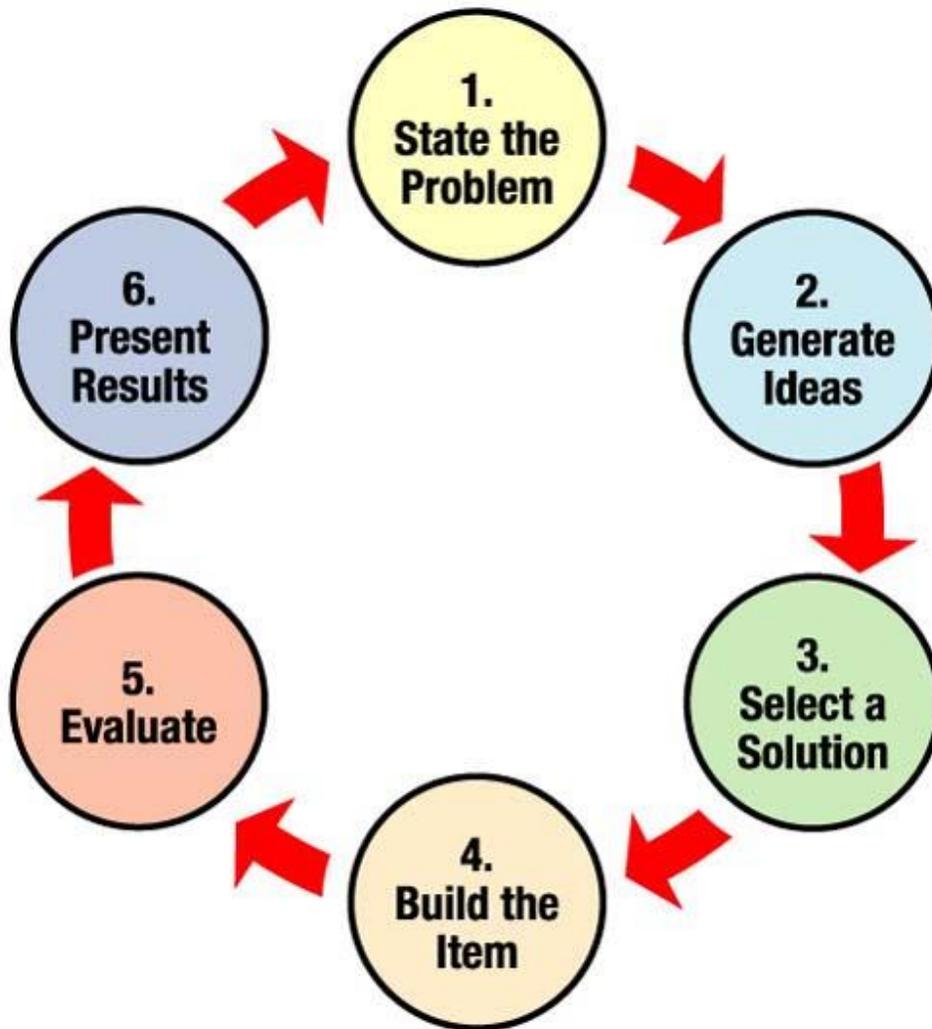


Figure 1.1.1 NASA's Design Process Diagram.

You cannot begin to work on solutions until you have a clear definition of the problem and goals you want to achieve. This critical first stage of the design process requires that you effectively communicate with the “client” or whoever has the “problem” that needs solving. Poor communication at this stage can derail a project from the start.

For our purposes, we will use Barry Hyman’s Problem Formulation model ^[2] to clearly define a problem. Hyman’s Problem Formulation model consists of 4 elements:

1. **Need Statement:** recognizes and describes the need for a solution or improvement to an “unsatisfactory situation.” It answers the questions, “what is wrong with the way things are currently? What is unsatisfactory about it? What negative effects does this situation cause?” You may need to do research and supply data to quantify the negative effects.

2. **Goal Statement:** describes what the improved situation would look like once a solution has been implemented. The goal statement defines the scope of your search for a solution. At this point, do **not** describe your solution, only the goal that any proposed solution should achieve. The broader you make your goal, the more numerous and varied the solution can be; a narrowly focused goal limits the number and variety of possible solutions.
3. **Objectives:** define measurable, specific outcomes that any feasible solution **should** optimize (aspects you can use to “grade” the effectiveness of the solution). Objectives provide you with ways to quantifiably measure how well any solution will solve the problem; ideally, they will allow you to compare multiple solutions and figure out which one is most effective (which one gets the highest score on meeting the objectives?).
4. **Constraints:** define the limits that any feasible solution must adhere to in order to be acceptable (pass/fail conditions, range limits, etc.). The key word here is must — constraints are the “go/no go” conditions that determine whether a solution is acceptable or not. These often include budget and time limits, as well as legal, safety and other regulatory requirements.

Communication as Solution

This model can apply to a communications task as well as more physical design tasks. Imagine your communications task as something that will solve a problem or improve a situation. Before you begin drafting this document or presentation, define the problem you want to solve with this document:

- **Understand the Need:** consider what gave rise to the need to communicate. Does someone lack sufficient information to make a decision or take a position on an issue? Did someone request information? Is there some unsatisfactory situation that needs to be remedied by communicating with your audience? What specifically is unsatisfactory about it? Consider your audience. For example
 - A potential client lacks sufficient information on whether the solution I have proposed to solve the client’s problem will be feasible, affordable, and effective.
 - My instructor lacks sufficient examples of my written work to assign a grade for how well I met the course learning objectives.
- **Establish a Goal:** consider your purpose in writing. What do you want your reader to do, think, or know? Do you want your reader to make a decision? Change their opinion or behaviour? Follow a course of action? What is your desired outcome? And what form and style of communication will best lead to that outcome? For example
 - Provide the client with enough information, in an effective and readable format, to make a decision (ideally, to hire you to build the solution for the problem).
 - Provide my instructor with samples of my writing that demonstrate my achievement of the course learning objectives (provide relevant and complete information in a

professionally appropriate format, using evidence-based argument; earn an A+ grade on the assignment.)

- **Define Objectives:** consider the specifics of your message and your audience to determine what criteria you should meet. What form should it take? What content elements will you need to include? What kind of research will be required? What information does your audience want/need? What do they already know?
 - Review the client’s RFP to see what specific objectives it lists and how your proposal will be assessed.
 - Review the Assignment Description and Grading Rubric for your assignment to determine specific requirements and objectives that your instructor will use to evaluate your work.
- **Identify Constraints:** what are the pass/fail conditions of this document? Consider your rhetorical situation. What conditions exist that present barriers or challenges to communication? How can you address them? For example,
 - how much time is your audience willing to spend on this? How long can you make your document or presentation? (word length/time limit)
 - What format and style do they require? Is there a Style Guide you must follow? A template you can use?
 - How much time do you have to create it? Do you have a deadline? (due date)
 - Are there requirements for using sources? (academic integrity rules)

Keep in mind that the document you produce is evaluated in terms of how well it responds to the “problem” — that is, how well it meets the overall goal and demonstrates achievement of specific objectives while abiding by constraints.

Problem solving techniques (Trail & Error, Brain Storming, Divide & Conquer)

Many times, a problem will have multiple solutions (unless it’s math)! To generate solutions, you can try these various methods:

Brainstorm: Allow everyone (even yourself) to share opinions on what they think can be a useful solution. Don’t shut down ideas in this stage. First, let everything come up with and then analyze what is actually feasible or practical.

Divide: Sometimes, problems are so big that they feel overwhelming and can lead to the fear of making a decision at all. Try to break down problems into smaller pieces to divide and conquer it in steps.

Means-Ends Analysis: To achieve a particular goal, you can work backwards. For example, you may want to become a software engineer. For this goal, you want to earn a degree in Computer Science, but the problem is affordability. With the goal of earning your higher education, you can think of alternative solutions like an online and tuition-

free education (the University of the People). In this way, you've taken the outcome and worked backwards to find a solution.

Trial and Error: Problem solving often includes failure — but trial and error can lead to the best solutions. You have to be open to trying different solutions until you reach the right one through trial and error.

Evaluating the Best Solution

When assessing multiple solutions for a given problem, you obviously want to choose the best one. Here are some ways to do so.

Eliminate Early: Ineffective solutions should be removed early on. Sometimes, it's obvious what won't work. If not, define parameters and budgets. If something is too expensive to implement, then it can quickly be removed as a solution.

Develop a Decision Matrix: You can use a decision matrix to see solutions visually. You can create a scale, for example using the ratings 1-10. Then you can assign a percentage of importance to each criteria. Criteria can include: timeliness, cost, risk, manageability, for example. In this way, you can see what solution will be the best by creating this value system.

Implement and Follow Up: Trust your analysis and, once you choose a solution, implement it. Be sure to track and measure if it's helping to achieve your desired goals.

Document

In a business setting, there are so many moving parts. It's good practice to document problems and solutions to gauge their success. It also creates a history that one can refer back to down the line when the next problem comes to light.

Advantages of a Problem Solving Process

Having a problem solving process in place helps to alleviate stress. It also can provide the following benefits:

There is consistency across an organization for how to manage problems

The process promotes collaboration and teamwork

The decision-process is informed, and therefore easier

The solutions are rational and objective

Steps in problem solving (Define Problem, Analyze Problem, Explore Solution):-

In every aspect of life, problems are bound to arise. In a workplace, problems can come in the form of client complaints or issues with teams. In a school setting, you may face problems with learning or teaching. And, in personal relationships, problems may take the most complex shapes and forms. Mastering problem solving steps can help you succeed in your career and more.

While every challenge is unique in its nature, there are a few methods to problem solving that are worthwhile to learn.

What is Problem Solving?

The four basic steps to problem solving are:

1. Define the Problem

It's common to conflate symptoms of a problem with the problem itself. When understanding what the root of the problem is, be sure to ask the right questions. If you're problem solving in a workplace, get team feedback. If you're problem solving in school, ask for the help of other students.

2. Create Alternatives

Once you know the problem you're facing, it's good to consider possible solutions. Often, there are a variety of solutions to the same problem. Be sure to exhaust all possibilities. This is another step where feedback and teamwork is useful.

3. Choose a Solution

Assess which solution will work best for those involved. If it's in a business, then you'll likely have to address the costs and benefits of any given solution. For problem solving in school settings, you may want to ask professors or mentors what they think will be the most effective.

4. Implement the Solution

Once you've chosen the best solution to a problem, you can implement it. If more problems arise, you will have to solve the problem again. But don't give up! Overcoming challenges only makes you stronger.

What are Problem Solving Skills?

While problem solving is a skill in itself, it also intersects with other skills. These skills include:

Active Listening

There's a difference between hearing and actively listening. Active listening requires the listener to give undivided attention to the speaker. By using active listening, you maximize problem solving skills because you can actually understand the problem when someone explains it.

Analysis

Analytical skills are crucial for problem solving. Everyone brings a different opinion and understanding of a problem to the table. By critically thinking about what's actually happening, you can create the best solutions.

Research

In businesses, big data is becoming everything. Using data and research, you can prevent problems before they even arise.

Creativity

Sometimes, when facing a problem, you will also have constraints. In fact, the constraints could be what's causing the problem. Utilizing creativity can help to overcome such challenges by thinking outside the box.

Communication

Talking about problems and accurately describing their roots will allow for contributions from your team. In this way, being able to properly communicate can help to hasten problem solving.

Decision-Making

Since challenges can have multiple solutions, you will need to know how to make a decision to implement the proper solution.

Team-Building

There are not many issues in life that require someone to be alone. Because of this, having a team with a strong foundation will help better address issues when they arise.

Problem Solving Process: A 7-Step Process

The following problem solving process is especially effective in businesses. When facing a challenge of any kind, leaders can rely on this method to come to a solution.

1. Identify the Issues

Different people may have different views on an issue. To identify the issues, allow everyone affected to share what they think is the problem.

2. Understand Interests

This is a critical step that is often overlooked. Once everyone has shared their views on a situation, it's useful to analyze why they feel this way about it. In this step, you must be accepting of everyone's differences. Understanding interests accurately also relies on active listening. By understanding interests, you will be able to better choose a solution that satisfies everyone's needs.

3. Define the Problem

Defining the problem can easily be conflated by emotion. To efficiently solve problems, you should be objective rather than subjective. No matter if a problem is small like choosing what to eat or large like choosing your major in college, defining a problem accurately is the basis of solving it properly.

The Kipling Method: A well-known method to define a problem comes from Rudyard Kipling, a famous poet. The 6 necessary elements to describe a problem include:

1. What is the problem?
2. Why is it important to fix the problem?
3. When did the problem start? What is the deadline to fix it?
4. How did the problem begin? What's its cause?
5. Where is the problem happening?
6. Who is affected by it?

4. Define the Goals

To solve a problem, you need to know what the goals are. In a team, it's important that the goal is communicated. This way, everyone can work together to achieve the desired outcome.

Algorithms and Flowcharts (Definitions, Symbols):-

In programming, algorithm is a set of well defined instructions in sequence to solve the problem.

Qualities of a good algorithm

1. Input and output should be defined precisely.
2. Each steps in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Examples Of Algorithms In Programming

Write an algorithm to add two numbers entered by user. Step 1: Start Step 2: Declare variables num1, num2 and sum. Step 3: Read values num1 and num2. Step 4: Add num1 and num2 and assign the result to sum. $sum \leftarrow num1 + num2$ Step 5: Display sum Step 6: Stop

Write an algorithm to find the largest among three different numbers entered by user. Step 1: Start Step 2: Declare variables a,b and c. Step 3: Read variables a,b and c. Step 4: If $a > b$ If $a > c$ Display a is the largest number. Else Display c is the largest number. Else If $b > c$ Display b is the largest number. Else Display c is the greatest number. Step 5: Stop

Advantages of algorithm

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program

Disadvantages of algorithm.

1. Writing algorithm takes a long time.
2. An Algorithm is not a computer program, it is rather a concept of how a program should be.

Flowchart ->A flowchart is a type of diagram that represents an algorithm, workflow or process. The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. ... Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Symbol of Flowchart->

Flowchart Symbols

Flowcharts use special shapes to represent different types of actions or steps in a process. Lines and arrows show the sequence of the steps, and the relationships among them. These are known as flowchart symbols.

Common Flowchart Symbols

- **Rectangle Shape** – Represents a process
- **Oval or Pill Shape** – Represents the start or end
- **Diamond Shape** – Represents a decision
- **Parallelogram** – Represents input/output

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Advantages of flowchart:

1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyze problem using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier.
4. After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.
6. It is easy to convert the flowchart into any programming language code as it does not use any specific programming language concept.

Disadvantage of flowchart

1. The flowchart can be complex when the logic of a program is quite complicated.
2. Drawing flowchart is a time-consuming task.
3. Difficult to alter the flowchart. Sometimes, the designer needs to redraw the complete flowchart to change the logic of the flowchart or to alter the flowchart.
4. Since it uses special sets of symbols for every action, it is quite a tedious task to develop a flowchart as it requires special tools to draw the necessary symbols.
5. In the case of a complex flowchart, other programmers might have a difficult time understanding the logic and process of the flowchart.
6. It is just a visualization of a program, it cannot function like an actual program

Characteristics of an algorithm Conditionals in pseudo-code

In the previous video and blog post I told you that an ALGORITHM is a set of ordered steps that take you from a known starting point to a predictable end point. A CONDITIONAL is a type of step in an algorithm where a decision must be made. Computers follow logical instructions and they need to know how to handle different decisions so that programs can proceed no matter what the outcome of those selections may be.

IF-THEN-ELSE CONDITIONALS

One of the first things that programmers learn is how to use IF-THEN-ELSE statements. Every programming language has some version of these. The syntax and exact usage may be different but they all accomplish the same thing, which is to allow for program execution based on conditionals. The basic flow is: If some condition is true then do this, otherwise do that.

Complex conditional statements can have more than just two choices. As humans, the way we make decisions when we have several options to choose from is very different than computers. We are able to select one item out of a group of choices, however a computer program must proceed by making binary decisions, meaning that it can only select between two things at a time. Even the most complex conditional statements boil down to a series of binary choices.

In this example, let's say that you are deciding on where to go to eat. Your favorite place is a pizza restaurant, your next choice would be a seafood restaurant, followed by a burger joint, and if all else fails you'll just cook something at home.

Any options after the one that is chosen are completely ignored. This means that an if-then-else chain has a hierarchy and that the cases that come first have a higher priority than the others because they are evaluated first. Even if multiple conditions happen to all be true, the first true condition will be selected because it is higher in priority. When writing your program it is important to think through the possible outcomes and decide which cases should come earlier in the sequence so that an important case doesn't get skipped.

AND/OR CONDITIONALS

If you want to check to make sure multiple conditions are met then you can use AND/OR statements. Using AND will cause the program to do something if both conditions are true whereas using an OR statement will do something if one of the conditions is true. Using these together with if-then-else statements will give you more power and control over your programs.

SWITCH AND CASE CONDITIONALS

The switch statement is used to allow you to perform different actions based on different conditions. In some languages this was a common structure for conditional execution that makes it easier for a program to execute one of several cases depending on the value of the expression at the switch.

The switch is evaluated once and then compared to the value of each case. If there is a match then the block of code is executed. Let's say that today is Sunday and you want to order a pizza from a restaurant that has daily specials. In this example we'll use a switch and case approach instead of running through an entire if-then-else chain, The switch is evaluated once, that value is compared to the value of each case, and if there is a match then the code in that block is executed.

Switch and case statements can be quicker than running through an if-then-else chain.

These were a few types of CONDITIONALS you will see as you are programming and coding. To summarize, conditionals are steps in an algorithm where decisions are made. Just like the algorithms that contain them, conditionals range from being simple and straightforward all the way to being highly complex.

Loops in pseudo code Time complexity

Complexity

- The whole point of the big-O/ Ω / Θ stuff was to be able to say something useful about algorithms.
 - So, let's return to some algorithms and see if we learned anything.
- Consider this simple procedure that sums a list (of numbers, we assume):
 - procedure sum(list)
 - total = 0
 - for i from 0 to length(list)-1
 - total += list[i]
 - return total
 - First: is the algorithm correct? Does it solve the problem specified?
 - Second: is it fast?
- To evaluate the running time of an algorithm, we will simply ask how many “steps” it takes.
 - In this case, we can count the number of times it runs the += line.
 - For a list with n elements, it takes n steps.
- Or is counting the += line the right thing to do?
 - When implementing the **for** loop, each iteration requires an add (for the loop index) and a comparison (to check the exit condition). We should count those.
 - Also, the variable initialization and **return** steps.
 - So, 3n+2 steps.
- But, not all of those steps are the same.
 - How long does an x86 ADD instruction take compared to a CMP or RET instruction?
 - Will the compiler keep both **i** and **total** in registers, or will one/both be in RAM? (A factor of ~10 difference.)
 - How do those instructions interact in the pipeline? Which can be sent through parallel pipelines in the processor and executed concurrently?
 - The answer to those is simple: I don't know and you don't either.
 - That's part of the reason we're asking about algorithms, not programs.
- But both n and 3n+2 are perfectly reasonable proposals for the answer.
 - Deciding between them requires more knowledge about the actual implementation details than we have.
- Good thing we have the function growth notation.
 - Remember: this is easy for n=5 elements. A good or bad algorithm will both be fast then.
 - We want to know how the algorithm behaves for large n.
- Finally our answer: the sum procedure has running time $\Theta(n)$.
 - We'll say that this algorithm has time complexity $\Theta(n)$, or “runs in linear time”.

- Both n and $3n+2$ are $\Theta(n)$, and so is any other “exact” formula we could come up with.
- The easy answer (count the `+=` line) was just as correct as the very careful one.
- The big- Θ notation hides all of the details we can't figure out anyway.
- Another example: print out the sum of each two numbers in a list.
 - That is, given the list `[1,2,3,4,5]`, we want to find `1+2`, `1+3`, `1+4`, `1+5`, `2+3`, `2+4`,...
 - Pseudocode:


```

          procedure sum_pairs(list)
            for i from 0 to length(list)-2
              for j from i+1 to length(list)-1
                print list[i] + list[j]
```
 - For a list with n elements, the **for j** loop iterates $n-1$ times when it is called with **i==0**, then $n-2$ times, then $n-3$ times,...
 - So, the total number of times the **print** step runs is

$$(n-1)+(n-2)+\dots+2+1=\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}=\frac{n^2-n}{2}.$$
 - If we had counted the initialization of the **for** loops, counter incrementing, etc, we might have come up with something more like $3n^2+12n+1$.
 - Either way, the answer we give is that it takes $\Theta(n^2)$ steps.
 - Or, the algorithm “has time complexity $\Theta(n^2)$ ” or “has $\Theta(n^2)$ running time” or “has quadratic running time”.
- The lesson: when counting running time, you can be a bit sloppy.
 - We only need to worry about the inner-most loop(s), not the number of steps in there, or work in the outer levels.
 - ... because they are going to disappear anyway as constant factors and lower-order terms when they go into a big- $O/\Omega/\Theta$ anyway.

Average and Worst Case

- Consider a linear search: we want to find an element in a list and return its (first) position, or -1 if it's not there.
- procedure `linear_search(list, value)`
- for `i` from 0 to `length(list)-1`
- if `list[i] == value`
- return `i`
- return -1

- How many steps there?
- The answer is: it depends.
 - If the thing we're looking for is in the first position, it takes $\Theta(1)$ steps.
 - If it's at the end, or not there, it takes $\Theta(n)$ steps.
- The easiest thing to count is usually the worst case: what is the maximum steps required for any input of size n ?
 - The worst case is that we go all the way to the end of the list, but don't find it and return -1.
 - The only line that makes sense to count here is the **if** line. It's in the inner-most loop, and is executed for every iteration.
 - We could also count the number of comparisons made: the **==** and the implicit comparison in the **for** loop.
 - That is either n or $2n+1$ steps, so $\Theta(n)$ complexity.
- The other useful option is the average case: what is the average steps required over all inputs of size n ?
 - Much harder to calculate, since you need to consider every possible input to the algorithm.
 - Even if we assume the element is found, the possible number of comparisons are:

Found in position	Comparisons
1	2
2	4
⋮	⋮
n	$2n$

- On average, the number of comparisons is:

$$2+4+\dots+2n=n+1.$$

- Again, we have $\Theta(n)$ complexity.
- ... but it's a good thing we checked. Some algorithms are different.

Good/bad times

- We have said that these running times are important when it comes to running times of algorithm.

- But we are throwing away a lot of information when we look only at big- $O/\Omega/\Theta$.
 - The lower-order terms must mean something.
 - The leading constants definitely do.
- Assuming one million operations per second, this is the approximate running time of an algorithm given running time, with an input of size n :

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3.3 μ s	10 μ s	33 μ s	100 μ s	1 ms	1 ms
102	6.6 μ s	100 μ s	664 μ s	10 ms	1 s	4×10^{16} years
104	13 μ s	10 ms	133 ms	1.7 minutes	11.6 days	102997 years
106	20 μ s	1 s	20 s	11.6 days	32000 years	10300000 years

- Maybe that gives a little idea why we'll only worry about complexity
 - ... at least at first.
- A summary:
 - If you can get $O(\log n)$ life is good: hand it in and go home.
 - $O(n \log n)$ is pretty good: hard to complain about it.
 - $O(n^k)$ could be bad, depending on k : you won't be solving huge problems. These are polynomial complexity algorithms for $k \geq 1$.
 - $\Omega(k^n)$ is a disaster: almost as bad as no algorithm at all if you have double-digit input sizes. These are exponential complexity algorithms for $k > 1$.
 - See also: Numbers everyone should know
- A problem that has a polynomial-time algorithm is called tractable.
 - No polynomial time algorithm: intractable.
 - There is a big category of problems that nobody has a polynomial-time algorithm for, but also can't prove that none exists: the NP-complete problems.
 - Some examples: Boolean satisfiability, travelling salesman, Hamiltonian path, many scheduling problems, Sudoku (size n).
- If you have an algorithm with a higher complexity than necessary, no amount of clever programming will make up for it.
 - No combination of these will make a $O(n^2)$ algorithm faster than an $O(n \log n)$: faster language, better optimizer, hand-optimization of code, faster processor.
- Important point: the complexity notations only say things about large n .
 - If you always have small inputs, you might not care.

- Algorithms with higher complexity class might be faster in practice, if you always have small inputs.
 - e.g. Insertion sort has running time $\Theta(n^2)$ but is generally faster than $\Theta(n \log n)$ sorting algorithms for lists of around 10 or fewer elements.
- The most important info that the complexity notations throw away is the leading constant.
 - There is a difference between n^2 instructions and $100n^2$ instructions to solve a problem.
 - Once you have the right big-O, then it's time to worry about the constants.
 - That's what clever programming can do.
- When we're talking about algorithms (and not programming), the constants don't usually matter much.
 - It's rare to have an algorithm with a big leading constant.
 - So it's not really possible to decide between the algorithms.
 - Usually it's a choice between $4n \log n$ or $5n \log n$: you probably have to implement, compile, and profile to decide for sure.
- Example: sorting algorithms. There are several algorithms to sort a list/array.
 - Insertion/Selection/Bubble Sorts: $\Theta(n^2)$.
 - Merge/Heap Sorts: $\Theta(n \log n)$.
 - Quicksort: $\Theta(n \log n)$ average case but (very rarely) $\Theta(n^2)$ worst case.
 - But quicksort is usually faster in practice.
 - ... except when it isn't.
 - Several recent languages/libraries have implemented a heavily-optimized mergesort (e.g. Python, Perl, Java \geq JDK1.3, Haskell, some STL implementations) instead of Quicksort (C, Ruby, some other STL implementations).

Space Complexity

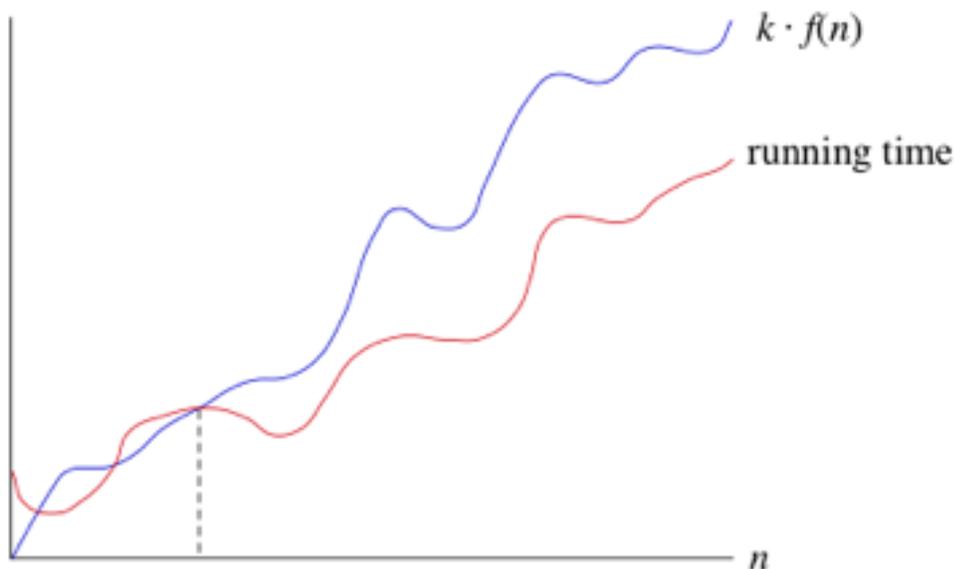
- We have only been talking about running time/speed so far.
 - It also makes good sense to talk about the complexity of other things.
- Most notably, memory use by an algorithm.
 - An algorithm that uses $\Theta(n^3)$ space is bad. Maybe as bad as $\Theta(n^3)$ time.
 - An algorithm that uses $O(1)$ extra space (in addition to space needed to store the input) is called in-place.
 - e.g. selection sort is in-place, but mergesort ($\Theta(n)$ extra space) and Quicksort ($\Theta(\log n)$ extra space, average case) aren't.

Big-Oh notation :- We use big- Θ notation to asymptotically bound the growth of a running time to within constant factors above and below. Sometimes we want to bound from only above.

For example, although the worst-case running time of binary search is $\Theta(\log_2 n)$, it would be incorrect to say that binary search runs in $\Theta(\log_2 n)$ time in *all* cases. What if we find the target value upon the first guess? Then it runs in $\Theta(1)$ time. The running time of binary search is never worse than $\Theta(\log_2 n)$, but it's sometimes better.

It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

If a running time is $O(f(n))$, then for large enough n , the running time is at most $k \cdot f(n)$ for some constant k . Here's how to think of a running time that is $O(f(n))$:



$6n^2$ vs $100n+300$

We say that the running time is "big-O of $f(n)$ " or just "O of $f(n)$." We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

Now we have a way to characterize the running time of binary search in all cases. We can say that the running time of binary search is *always* $O(\log_2 n)$.

parenthesis, log, start base, 2, end base, n, right parenthesis. We can make a stronger statement about the worst-case running time: it's $\Theta(\log_2 n)$. But for a blanket statement that covers all cases, the strongest statement we can make is that binary search runs in $O(\log_2 n)$ time.

If you go back to the definition of big- Θ notation, you'll notice that it looks a lot like big- O notation, except that big- Θ notation bounds the running time from both above and below, rather than just from above. If we say that a running time is $\Theta(f(n))$ in a particular situation, then it's also $O(f(n))$. For example, we can say that because the worst-case running time of binary search is $\Theta(\log_2 n)$, it's also $O(\log_2 n)$.

The converse is not necessarily true: as we've seen, we can say that binary search always runs in $O(\log_2 n)$ time but *not* that it always runs in $\Theta(\log_2 n)$ time.

Because big- O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first glance seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in $O(n)$ time. That's because the running time grows no faster than a constant times n . In fact, it grows slower.

Think of it this way. Suppose you have 10 dollars in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars." Your statement is absolutely true, though not terribly precise.

One million dollars is an upper bound on 10 dollars, just as $O(n)$ is an upper bound on the running time of binary search. Other, imprecise, upper bounds on binary search would be $O(n^2)$, $O(n^3)$, and $O(2^n)$. But none of $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, or $\Theta(2^n)$.

and $\Theta(2^n)$, left parenthesis, 2, start superscript, n, end superscript, right parenthesis would be correct to describe the running time of binary search in any case.

efficiency Simple Examples: Algorithms and flowcharts (Real Life Examples)

Algorithm and Flowcharts

Algorithm

Definition:-

An algorithm is set of instructions for solving a problem or accomplishing a task. One common example of an algorithm is a recipe, which consists of specific instructions for preparing a dish/meal. Every computerized device uses algorithms to perform its functions.

- An algorithm is set of instructions for solving a problem or accomplishing a task. Every computerized device uses algorithms to perform its functions.
- Algo trading, also known as automated trading or black-box trading, uses a computer program to buy or sell securities at a pace not possible for humans. Since prices of stocks, bonds, and commodities appear in various formats online and in trading data, the process by which an algorithm digests scores of financial data becomes easy.
- Computer algorithms make life easier by trimming the time it takes to manually do things. In the world of automation, algorithms allow workers to be more proficient and focused. Algorithms make slow processes more proficient. In many cases, especially in automation, algos save companies money.

How an Algorithm Works

Financial companies use algorithms in areas such as loan pricing, stock trading, asset-liability management, and many automated functions. For example, algorithmic trading, known as "algo" trading, is used for deciding the timing, pricing, and quantity of stock orders. Algo trading, also known as automated trading or black-box trading, uses a computer program to buy or sell securities at a pace not possible for humans.

Since prices of stocks, bonds, and commodities appear in various formats online and in trading data, the process by which an algorithm digests scores of financial data becomes easy. The user of the program simply sets the parameters and gets the desired output when securities meet the trader's criteria.

Computer algorithms make life easier by trimming the time it takes to manually do things. In the world of automation, algorithms allow workers to be more proficient and focused. Algorithms make slow processes more proficient. In many cases, especially in automation, algos save companies money.

Types of Algos

Several types of trading algorithms help investors decide whether to buy or sell. A mean reversion algorithm examines short-term prices over the long-term average price, and if a stock goes much higher than the average, a trader may sell it for a quick profit. Seasonality refers to the practice of traders buying and selling securities based on the time of year when markets typically rise or fall. A sentiment analysis algorithm gauges news about a stock price that could lead to higher volume for a trading period.

Algorithm Example

The following is an example of an algorithm for trading. A trader creates instructions within his automated account to sell 100 shares of a stock if the 50-day moving average goes below the 200-day moving average.

Contrarily, the trader could create instructions to buy 100 shares if the 50-day moving average of a stock rises above the 200-day moving average. Sophisticated algorithms consider hundreds of criteria before buying or selling securities. Computers quickly synthesize the automated account's instructions to produce the desired results. Without computers, complex trading would be time-consuming and likely impossible.

Algorithms in Computer Science

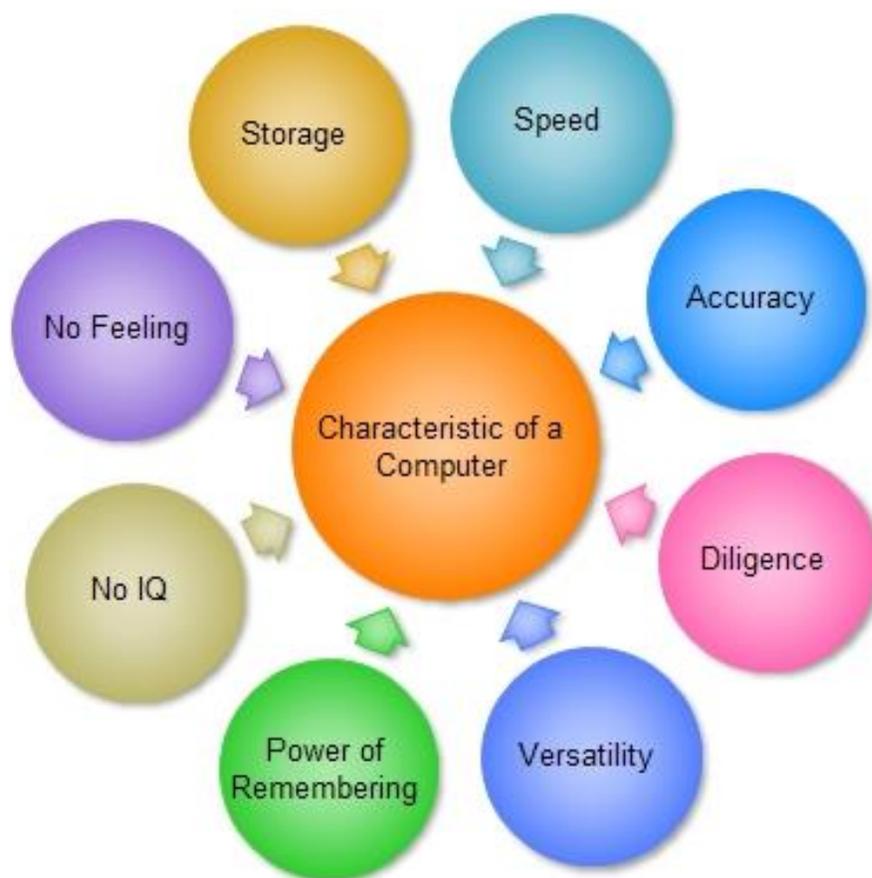
In computer science, a programmer must employ five basic parts of an algorithm to create a successful program.

First, he/she describes the problem in mathematical terms before creating the formulas and processes that create results. Next, the programmer inputs the outcome parameters, and then he/she executes the program repeatedly to test its accuracy. The conclusion of the algorithm is the result given after the parameters go through the set of instructions in the program.

For financial algorithms, the more complex the program, the more data the software can use to make accurate assessments to buy or sell securities. Programmers test complex algorithms thoroughly to ensure the programs are without errors. Many algorithms can be used for one problem; however, there are some that simplify the process better than others.

Characteristics

Basic characteristics about computer are:



1. Speed: – As you know computer can work very fast. It takes only few seconds for calculations that we take hours to complete. You will be surprised to know

that computer can perform millions (1,000,000) of instructions and even more per second.

Therefore, we determine the speed of computer in terms of microsecond (10⁻⁶ part of a second) or nanosecond (10 to the power -9 part of a second). From this you can imagine how fast your computer performs work.

2. Accuracy: – The degree of accuracy of computer is very high and every calculation is performed with the same accuracy. The accuracy level is **7.** determined on the basis of design of computer. The errors in computer are due to human and inaccurate data.

3. Diligence: – A computer is free from tiredness, lack of concentration, fatigue, etc. It can work for hours without creating any error. If millions of calculations are to be performed, a computer will perform every calculation with the same accuracy. Due to this capability it overpowers human being in routine type of work.

4. Versatility: – It means the capacity to perform completely different type of work. You may use your computer to prepare payroll slips. Next moment you may use it for inventory management or to prepare electric bills.

5. Power of Remembering: – Computer has the power of storing any amount of information or data. Any information can be stored and recalled as long as you require it, for any numbers of years. It depends entirely upon you how much data you want to store in a computer and when to lose or retrieve these data.

6. No IQ: – Computer is a dumb machine and it cannot do any work without instruction from the user. It performs the instructions at tremendous speed and with accuracy. It is you to decide what you want to do and in what sequence. So a computer cannot take its own decision as you can.

7. No Feeling: – It does not have feelings or emotion, taste, knowledge and experience. Thus it does not get tired even after long hours of work. It does not distinguish between users.

8. Storage: – The Computer has an in-built memory where it can store a large amount of data. You can also store data in secondary storage devices such as floppies, which can be kept outside your computer and can be carried to other computers.

Advantages and disadvantages

The invention of the computer is considered to be one of the greatest inventions of all time. The modern computer has changed our daily life to some extent. A computer is an integral part of human beings and we can not imagine our lives without the use of a computer.

As there are two sides of the coins there are advantages and disadvantages of computer system in points which we are going to discuss in detail.

We will focus on each and every topic briefly. The computer has reached to every section of human society, from schools to hospitals business organizations, institutions everywhere we cannot imagine our daily life without the use of computers.

The impact of computers on humans is beyond imagination, people use a computer for selling and purchasing goods, online studies (E-Learning) for searching virtually anything on the web, playing games, watching movies, downloading software's, for business promotions, railways, and airplanes tickets, etc almost everything can be done with the help of computer and the output can be obtained with just a few clicks.

Their uses and study of computers are mandatory for students because of its worldwide use and acceptance.

The computers have made a vital impact on education, students can learn online, and get the required skills and knowledge just sitting at home, the computers have made the distance virtually zero.

Students are getting a higher education, certificates, and degrees from the reputed institutes before they just dreamed of it computer has made their dreams into reality.

They even had made a significant impact in business, now businessman prefers to use a computer to a huge extent they have made their business life so peaceful that now they can perform multitasking, with almost 100% accuracy, The advantages of computer in research has made the scientist to solve complex to complex problems with ease before would take long hours or even months to solve.

As there are 2 sides of coins there are Advantages and Disadvantages of computers ::

- Speed
- Accuracy
- Stores Huge Amount of Data
- Online Trading
- Online Education | Distance Learning
- Research
- Forecasting Weather, and Predicting Earthquakes , Volcano Eruptions
- Produce Employment
- Internet
- In Business

Speed

The speed of computer has made a vital impact on human society before some decades computer were just used for the purpose of some numerical calculations, but nowadays computers are used in virtually every single part of human life.

The modern computer is not just a calculating device anymore with their speed and accuracy it can perform multiple tasks, operations, and complex numerical problems within fractions of second. They can perform about Trillions of instruction per second.

People can play songs, perform their documentation work, surf the internet, check emails, search for your requirements on the internet with great speed and accuracy.

The speed is considered as the biggest advantage of computers because it can perform all operations with an incredible speed which can reduce the amount of time spend when working manually.

For Example : It can calculate the salaries of Employees within a fraction of seconds before it would take long hours when done manually by people.

Accuracy

Not only speed but a computer can work with almost 100% accuracy. This is also one of its advantages, as it can perform complex numerical calculations not only with speed but with unbelievable accuracy..

Imagine you have given a task to calculate the gross salary of ten thousand employees with reduction as per the rules and regulations of the organization. That's not a simple task to calculate the given salaries manually, you are bound to make mistakes and even a small mistake can harm you badly economically.

And now here comes the magic of computers it can perform all the hard work of calculating the salaries of the employees and deduction as per the requirements. You can derive the results at your fingertips within a fraction of seconds when given correct and proper input, with great speed and incredible accuracy.

Hence the big organizations with hundreds of employees working for them use computer-generated bills and payslips. Even in the Government sector, all the manual calculations had been shifted to computers.

Stores Huge Amount of Data

Data storing capabilities of computers these days are "HUGE" as compared to previous years. They can save or store any volume of data or instruction given to them permanently.

Users can recall the data or instructions or information given to them anytime & any place.

The Storing capability of computer is measured in MB (Mega-Bytes), GB (Giga-Bytes), TB (Tera-Bytes) Railways use modern computer for storing their passenger details, route of trains and even their employee's details. India has a population of around 125 cr people, 93% of them have received their aadhaar card issued by the Government of India.

Online Trading

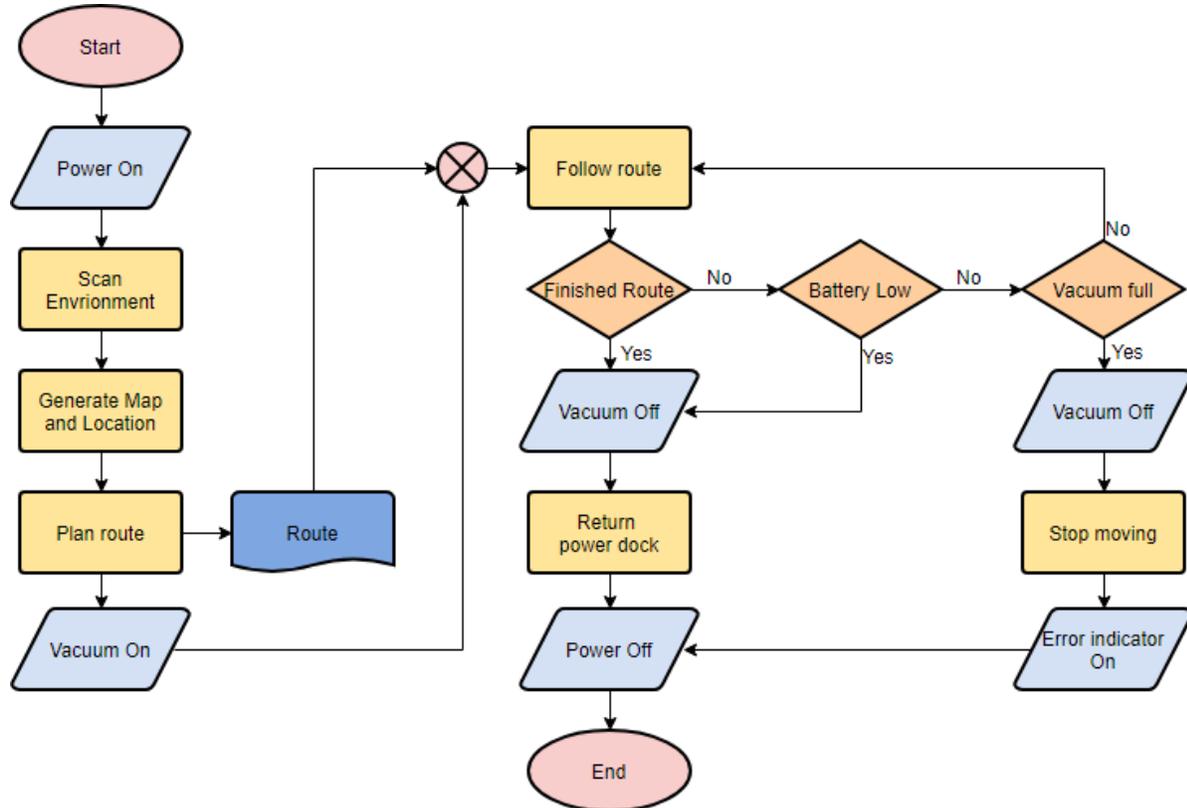
People tend to use computer and internet while purchasing and selling their goods, according to a recent survey more than 50% of people across the world will use computers for their online trading.

Online trading is very simple and time saving, you have a variety of products to choose with the best prices, many websites offer their users a heavy discount. People are more inclined toward online shopping and trading these days due to their simplicity in use.

Examples Flowchart:

Definition

A flowchart is simply a graphical representation of steps. It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes. Typically, a flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows.



What is a Flowchart?

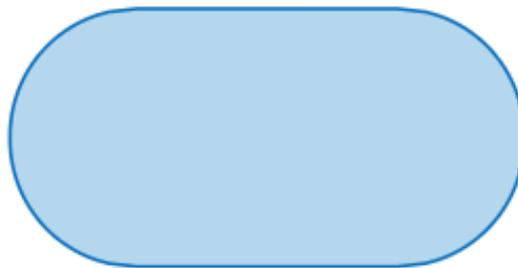
A flowchart is a graphical representations of steps. It was originated from computer science as a tool for representing algorithms and programming logic but had extended to use in all other kinds of processes. Nowadays, flowcharts play an extremely important role in displaying information and assisting reasoning. They help us visualize complex processes, or make explicit the structure of problems and tasks. A flowchart can also be used to define a process or project to be implemented.

Define symbols of flowchart

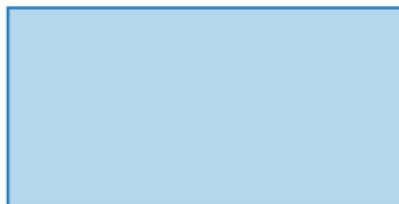
Different flowchart shapes have different conventional meanings. The meanings of some of the more common shapes are as follows:

Terminator

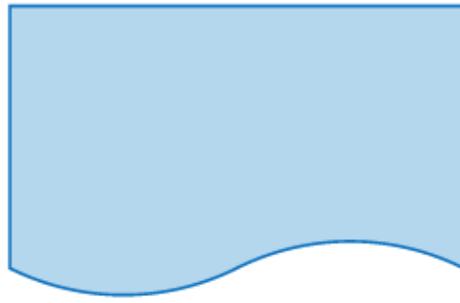
The terminator symbol represents the starting or ending point of the system.



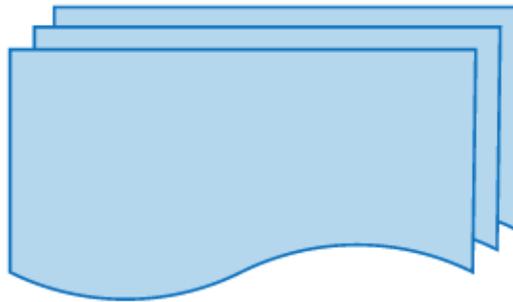
Start/End Symbol



Action or Process Symbol

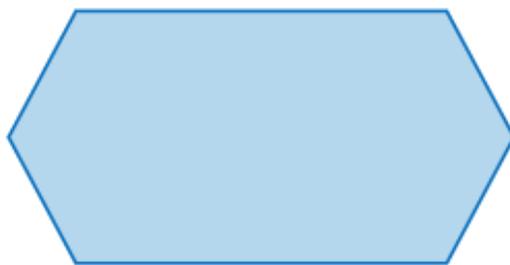


Document Symbol

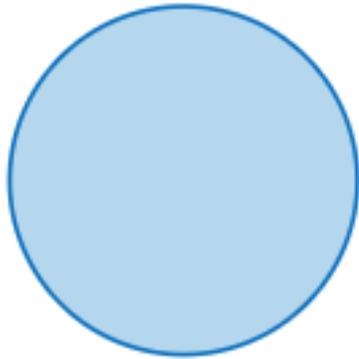


Multiple Documents

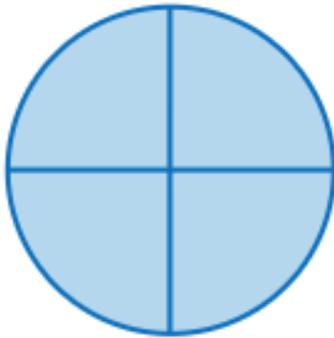
Symbol



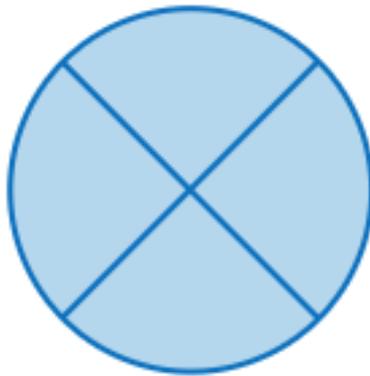
Preparation Symbol



Connector Symbol



Or Symbol



Summoning Junction Symbol

Advantages and disadvantages

A flowchart is actually a graphical representation of a computer program in relation to its sequence of functions. Which will help the programmer to draw out the basic build block and working mechanism of the program or the system?

In the **flowchart**, we generally use geometric symbols like a rectangle, oval shapes and arrows to define the relationships. In simple terms, it will explain the start and end of the program.

Advantages of Flowchart

1. It is a convenient method of communication.
2. It indicates very clearly just what is being done, where a program has logical complexities.
3. A key to correct programming.
4. It is an important tool for planning and designing a new system.
5. It clearly indicates the role-played at each level.
6. It saves the inconveniences in future and serves the purpose of documentation for a system.
7. It provides an overview of the system and also demonstrates the relationship between various steps.
8. Facilitates troubleshooting.
9. It promotes logical accuracy.
10. It makes sure that no logical path is left incomplete without any action being taken.

Disadvantages of Flowchart

1. The flowchart is a waste of time and slows down the process of software development.
2. The flowchart is quite costly to produce and difficult to use and manage.
3. Flowcharts are not meant for man to computer communication.
4. Sometimes the Complex logic of the program logic is quite complicated to draw out on by using different defined shapes. In that case, flowchart becomes

complex and clumsy. This will become a pain for the user, resulting in a waste of time and money trying to correct the problem

5. If you need to modify or alternate the process then it will be very hard to do in the flowchart. Because either you will have to erase the end of the flowchart or start.

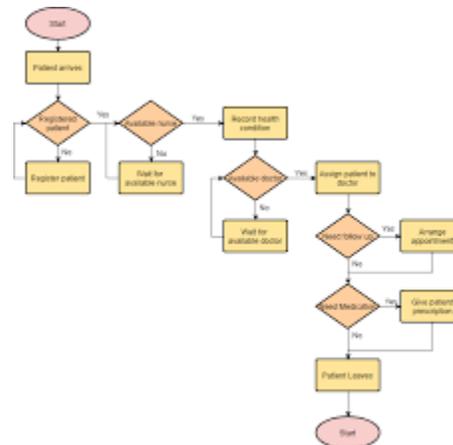
Flowchart Advantages

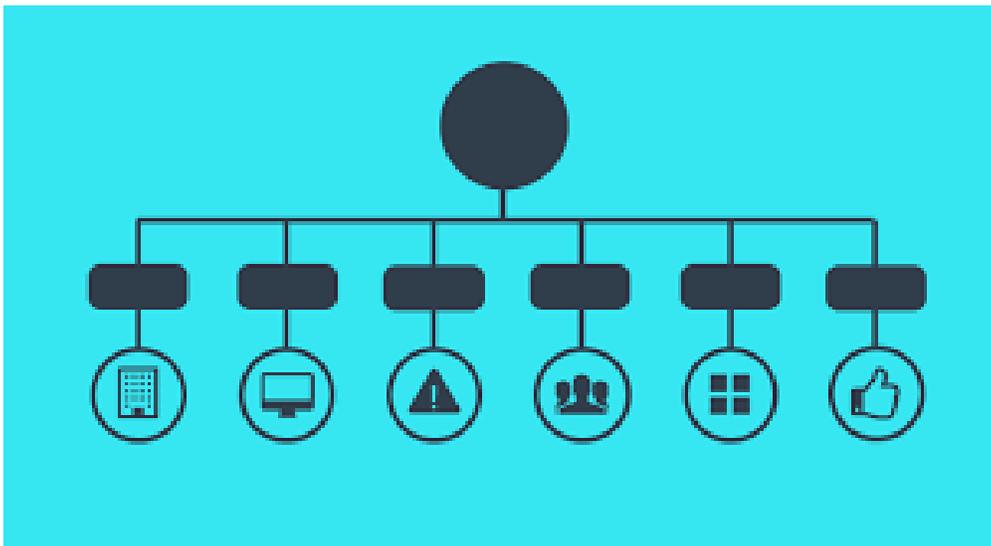
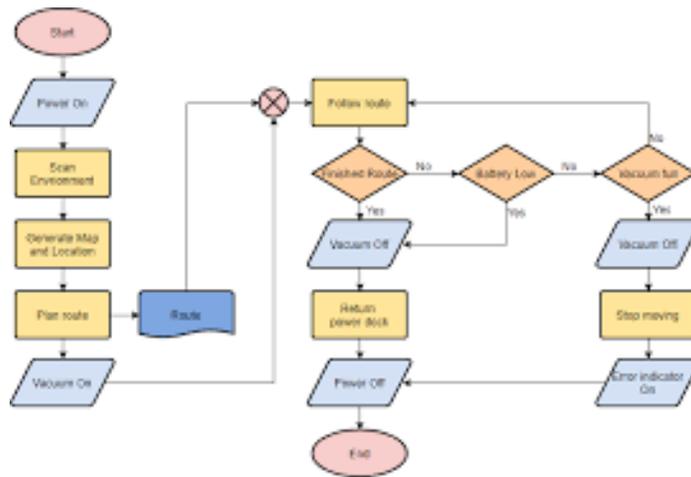
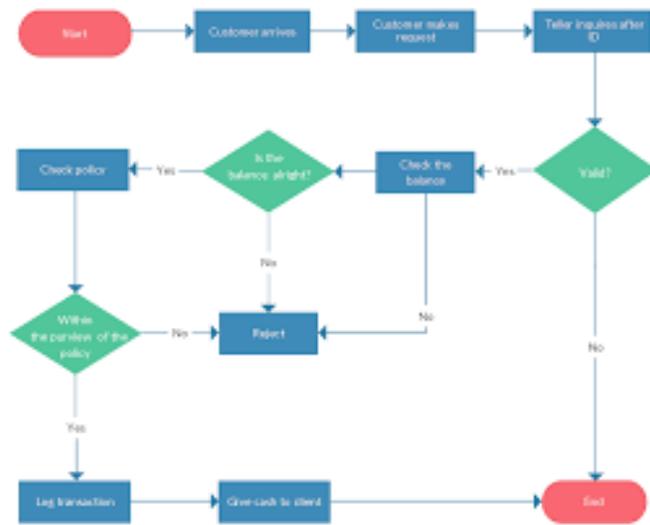
- Flowcharts are easier to understand compare to Algorithms and Pseudo code.
- It helps us to understand Logic of given problem.
- It is very easy to draw flowchart in any word processing software like MS Word.
- Using only very few symbol, complex problem can be represented in flowchart.
- Software like RAPTOR can be used to check correctness of flowchart drawn in computers.
- Flowcharts are one of the good way of documenting programs.
- It helps us in debugging process.

Flowchart Disadvantages

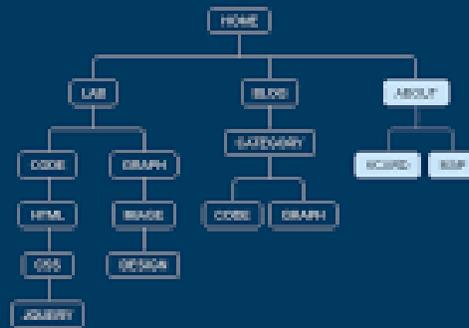
- Manual tracing is needed to check correctness of flowchart drawn on paper.
- Simple modification in problem logic may leads to complete redraw of flowchart.
- Showing many branches and looping in flowchart is difficult.
- In case of complex program/algorithm, flowchart becomes very complex and clumsy.
- Modification of flowchart is sometimes time consuming.

Examples





Menu parent children as a diagram...



UNIT-IV

Simple Arithmetic Problems

Addition / Multiplication of integers

Integer numbers are whole numbers. These can be negative, positive or a zero. When we perform mathematical operations of integers, we follow a different set of rules that are specific to the character of the number. Not every rule of mathematical operation applies to the integers. Let's see how handling integers are different from the normal mathematical operations.

Integer Numbers and Mathematical Operations

Mathematical operations include addition, subtraction, multiplication, and division of any number. When we perform these operations with integer numbers we always keep in mind the sign before every number.

As we already know that an integer includes a number with a positive or negative sign, therefore, these have to be dealt with different perceptions. Before delving into further operations, we first need to know the properties related to these mathematical operations.

Property of Closure in Addition and Subtraction

When we add two whole numbers, the answer is a whole number. This is called the closure property of additions. Now, do integers also follow this property? Let's see:

$$10 + (-12) = -2$$

Is -2 an integer number. Since it has a negative sign with it, therefore we call it an integer number. From the above example, we can say that the sum of two integers is also an integer. So when an integer a is added with another integer b the answer is always an integer.

Let's take some more examples to support our observation:

- $19+10= +29$
- $15 + (-20)= -5$

In both the examples, the solution is an integer. Therefore every integer when summed up with an integer gives an integer. What happens when an integer is subtracted from an integer. Let's see the following examples:

- $10 - 3 = +7$
- $15 - (-7) = -22$

In both the examples, the difference between two integers gives an integer in solution. This means that integers are also closed under subtraction. Hence, in the case of subtraction of two integers a and b the answer to $a-b$ is also an integer.

Commutative Property

Commutative property states the following:

$$\mathbf{a+b =b+a}$$

For whole numbers, we already know that the sum of two whole numbers is always the same. But is the case true for integer numbers as well? The sum of two integer numbers is also, always the same. This means that integer numbers also follow the commutative property like whole numbers. Let's see the following examples:

- $15 + 20 = 35$; $20 + 15 = 35$
- $-10 + (-5) = -15$; $-5 + (-10) = -15$

The above examples prove that the addition of integers is commutative. Is the case true with subtractions? Are subtractions also commutative? The following examples will let us know this:

- $5 - (-3) = +8$
- $-3 - 5 = -8$

Are the integer numbers same? The answer is a no! This brings us to a conclusion that subtractions of integers are not commutative. Therefore, $\mathbf{a-b \neq b-a}$

Associative Property

According to associative property of whole numbers a, b, c ,

$$\mathbf{[a+b]+c =a+[b+c]}$$

Does this property also apply to integer numbers? Let's see:

- $[(-4)+(-6)]+(-2) = -12$
- $(-4) + [(-6)+(-2)] = -12$

The answer in both the cases is the same. So the sum of integer numbers abides the associative property of addition.

Additive Identity

The additive identity of any number is checked with the help of zero. For all the whole numbers, zero proves to be their additive identity. This means that when a whole number is added with a zero, the answer is the whole number itself. Is the case same with integer numbers as well? The answer here is a yes. The following examples prove our observation:

- $+10 + 0 = +10$
- $-9 + 0 = -9$

Therefore for any integer, x ,

$$x+0 = x = 0+x$$

Multiplication of Integers

After addition and subtraction, we need to understand the multiplication of two integer numbers. Let's look at some of its properties.

Closure Property

Is the product of two integer numbers also an integer number? Well, yes the product of every integer is an integer.

Commutative Property

The product of two same integers is always the same. This means that

$$a \times b = b \times a$$

Multiplication with Zero

Every integer, when multiplied with a zero, gives zero as the answer.

$$a \times 0 = 0$$

Multiplicative Identity

Every integer number, when multiplied with 1, gives an integer number in an answer.

- $a \times 1 = a$
- $(-a) \times 1 = -a$

Associative Property

Integer numbers also abide by the associative property of multiplication. This implies that like whole numbers the grouping of integers does not affect the product of integers. Thus,

$$(a \times b) \times c = (a) \times (b \times c)$$

Distributive Property

Here we shall check the distributivity of integers over addition is true or not.

- $(-4) \times (2 + 6) = -32$
- $(-4) \times (2) + (-4) \times (+6) = (-8) + (-24) = -32$

The example above shows that multiplication of integers also shows distributivity of multiplication over addition. Thus,

$$a \times (b+c) = (a \times b) + (a \times c)$$

Does the distributive property also comply with subtractions in multiplication? The answer is a yes! The example below will help you understand:

- $(-8) \times (5 - 3) = -16$
- $[(-8) \times (+5)] - [(-8) \times (-3)] = (-40) - (+24) = -16$

From the above example we can thus state that;

$$\mathbf{a \times (b-c) = (a \times b) - (a \times c)}$$

Multiplication of a Negative and Positive Integer

As already said earlier, while performing the mathematical operations of integer numbers, we must always keep in mind the respective sign before every number. When multiplying two integers with a negative and positive sign, the product shall always be negative.

This means that in every case of multiplication between contradictorily signed integers the answer shall always have a negative sign. For example:

$$4 \times (-5) = -20$$

To simplify the product we write the above example as :

$$(-5) + (-5) + (-5) + (-5) = -20$$

Some more examples are:

- $-3 \times 4 = -12$
- $12 \times -6 = -72$

$$\text{Therefore } \mathbf{(-a) \times b = a \times (-b) = -(a \times b)}$$

The multiplication of integers is similar to multiplication of whole numbers. The only difference here is that after multiplying we put a minus sign (-) in the answer.

Multiplication of Two Negative Integers

The multiplication of two negative integer numbers is always a positive integer. This means the product of $\mathbf{(-a) \times (-b) = +c}$. The negative integer numbers in multiplication are multiplied like ordinary whole numbers. Some examples will help you understand this in a better way:

- $(-3) \times (-6) = 18$
- $(-30) \times (-15) = 450$

This means that: $\mathbf{(-a) \times (-b) = a \times b}$. The minus (-) sign of the integer numbers while multiplication is ignored.

Product of Three or More Integers

In the above topics, we multiplied only two integers. Now we shall multiply three or more than three integers. Consider the following image:

$$(-10) \times (+5) \times (+6) = (-10) \times (+30) = -300$$

In the above set of examples, what do you notice? When we multiply more than three integers, we multiply them in the set of two. The signs of the answer depend on the rules we already mentioned in the multiplication of positive and negative integer.

$$(-25) \times (-10) \times (-2) \times (-4) = (250) \times (8) = +2000$$

In both the example images we notice that when we multiply two negative integers the answer is a positive integer. When we multiply two integers with negative and positive signs respectively, the answer is a negative integer number.

In case you are confused about signs, you may follow another very simple way of applying a sign. Now, if the number of negatively signed numbers is odd, the answer shall be a negative integer. If the negative integers are in even number then the answer shall be a positive integer.

$$(-a) \times (-b) \times (-c) \times (-d) = +z$$

In the example given above, the total number of integers are even in number, hence the answer is a positive integer.

$$(-a) \times (-b) \times (-c) \times (-d) \times (-e) = -z$$

See the example given above. The total number of integers are odd in number, hence answer is a negative integer.

Division of Integer numbers

We know that divisions are inverse of multiplications. How do we then perform divisions in integers? Division of integers is similar to the division of whole numbers. While dividing an integer with an integer we divide them like normal whole numbers and then put the sign respective to the integers involved.

This means that when we divide a positive integer with a positive integer the answer is a positive integer. Similarly, when the division involves one positive and one negative integer then, we divide them like normal whole numbers and put a negative sign in the quotient. For example:

- $72 \div (-8) = -9$
- $(-80) \div (5) = -16$

Now, let's look at some of the properties for the division of integers.

Closure Property of Division

Is the division of two integer numbers also an integer number? Well, no. The division of two numbers does not give an integer in the answer.

- $(-8) \div (2) = -4$
- $(-9) \div (2) = -9/2$

So the division of integers does not give an integer in the answer.

Commutative Property of Division

This property does not apply to divisions between integers. This means that $a \div b \neq b \div a$

Division by Zero

Like whole numbers, an integer number also cannot be divided by (zero) 0, but when zero (0) is divided by an integer the answer is zero (0).

Division by One

Every integer number, when divided by 1, gives the same integer number in the answer. But the case is not the same when an integer is divided by -1

- $a \div 1 = a$
- $(a) \div (-1) = -a$

From the above illustrations, we have come to know that the mathematical operations of integer number involve cautious calculations as these follow a different set of rules.

Solved Examples for You

Question 1: In a test (+5) marks are given for every correct answer and (-2) marks are given for every incorrect answer.

- Shyam answered all the questions and scored 40 marks though he got 10 correct answers.**
- Renu also answered all the questions and scored (-16) marks though he got only 4 correct answers.**

How many incorrect answers had they attempted?

Answer : One correct answer gives = +5 marks. Shyam gave 10 correct answers, so, marks given for 10 correct answers = $5 \times 10 = 50$. Shyam's score = 40.

Marks obtained for incorrect answers = $40 - 50 = -10$

Marks given for one incorrect answer = (-2)

Therefore, number of incorrect answers = $(-10) \div (-2) = 5$

Shyam gave 5 incorrect answers.

Marks given for 4 correct answers = $5 \times 4 = 20$

Renu's score = -16

Marks obtained for incorrect answers = $-16 - 20 = -36$

Marks given for one incorrect answer = (-2)

Therefore number of incorrect answers = $(-36) \div (-2) = 18$

Renu gave 18 incorrect answers.

Question 2: Is a decimal an integer?

Answer: We can express any integers as a decimal however most of the numbers that can be expressed as a decimal are not integers. In addition, if all the digits after the decimal are zeros then the number is an integer, but if they are any other number except zero after the decimal point then it is not an integer.

Question 3: How to express answers in an integer?

Answer: Integers refers to all those numbers that do not contain any decimal point or are in fraction. Besides, fro converting any number to integers just count the number of significant figures in the number and write this down as a whole number.

Question 4: What is a valid integer value?

Answer: In a computer definition, integers are the whole number, which on sending the number 145.5732 to an integer function would return 145. Moreover, they can be signed (negative or positive) or unsigned (always positive).

Question 5: Is pi a real number?

Answer: It is an irrational number, which means that it is a real number that we cannot express using a simple fraction. When students first introduced to pi then they were told its value to be $22/7$ or 3.14 or 3.14159.

Determining if a number is +ve / -ve / even / odd :-

In this example, if...else statement is used to check whether a number entered by the user is even or odd.

To understand this example, you should have the knowledge of the following C++ programming topics:

C++ if, if...else and Nested if...else

Integers which are perfectly divisible by 2 are called even numbers.

And those integers which are not perfectly divisible by 2 are not known as odd number.

To check whether an integer is even or odd, the remainder is calculated when it is divided by 2 using modulus operator %. If remainder is zero, that integer is even if not that integer is odd.

Example 1: Check Whether Number is Even or Odd using if else

```
#include <iostream>

using namespace std;

int main()
{
    int n;

    cout << "Enter an integer: ";
    cin >> n;

    if ( n % 2 == 0)
        cout << n << " is even.";
    else
        cout << n << " is odd.";

    return 0;
}
```

Output

Enter an integer: 23

23 is odd.

In this program, if..else statement is used to check whether $n\%2 == 0$ is true or not. If this expression is true, n is even if not n is odd.

You can also use ternary operators ?: instead of if..else statement. Ternary operator is short hand notation of if...else statement.

Example 2: Check Whether Number is Even or Odd using ternary operators

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int n;
```

```
    cout << "Enter an integer: ";
```

```
    cin >> n;
```

```
    (n % 2 == 0) ? cout << n << " is even." : cout << n << " is odd.";
```

```
    return 0;
```

```
}
```

Sum of first n numbers:-

In this example, you will learn to calculate the sum of natural numbers entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

C for Loop

C while and do...while Loop

The positive numbers 1, 2, 3... are known as natural numbers. The sum of natural numbers up to 10 is:

$$\text{sum} = 1 + 2 + 3 + \dots + 10$$

Sum of Natural Numbers Using for Loop

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

```
int main() {
```

```
    int n, i, sum = 0;
```

```
    printf("Enter a positive integer: ");
```

```
    scanf("%d", &n);
```

```
    for (i = 1; i <= n; ++i) {
```

```
        sum += i;
```

```
    }
```

```
    printf("Sum = %d", sum);
```

```
    return 0;
```

```
}
```

The above program takes input from the user and stores it in the variable n. Then, for loop is used to calculate the sum up to n.

Sum of Natural Numbers Using while Loop

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

```
int main() {
```

```
    int n, i, sum = 0;
```

```
    printf("Enter a positive integer: ");
```

```

scanf("%d", &n);

i = 1;

while (i <= n) {
    sum += i;
    ++i;
}

printf("Sum = %d", sum);
return 0;
}

```

Output

Enter a positive integer: 100

Sum = 5050

In both programs, the loop is iterated n number of times. And, in each iteration, the value of i is added to sum and i is incremented by 1.

Though both programs are technically correct, it is better to use for loop in this case. It's because the number of iterations is known.

The above programs don't work properly if the user enters a negative integer. Here is a little modification to the above program where we keep taking input from the user until a positive integer is entered.

Read Input Until a Positive Integer is Entered

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

```
int main() {
```

```
    int n, i, sum = 0;
```

```

do {
    printf("Enter a positive integer: ");
    scanf("%d", &n);
} while (n <= 0);

for (i = 1; i <= n; ++i) {
    sum += i;
}

printf("Sum = %d", sum);

return 0;
}

```

Integer Division

If one integer divides another in a subexpression then type of that subexpression is INTEGER. Confusion often arises about integer division; in short, division of two integers produces an integer result by truncation (towards zero).

Consider,
REAL a, b, c, d, e

```

a = 1999/1000
b = -1999/1000
c = (1999+1)/1000
d = 1999.0/1000
e = 1999/1000.0

```

- a is (about) 1.000. The integer expression 1999/1000 is evaluated and then truncated towards zero to produce an integral value. Its says in the Fortran 90 standard, [1], P84 section 7.2.1.1, ``The result of such an operation [integer division] is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively."
- b is (about) -1.000 for the same reasons as above.
- c is (about) 2.000 because, due to the parentheses 2000/1000 is calculated.

- d and e are (about) 1.999 because both RHS's are evaluated to be real numbers, in 1999.0/1000 and 1999/1000.0 the integers are promoted to real numbers before the division.

Digit reversing

The following is a C program to reverse the digits of a number:

```

1 /*****
2  * Program to reverse the digits of a number
3  *****/
4
5 #include<stdio.h> // include stdio.h
6
7 int main()
8 {
9     long int num, rev = 0;
10    int rem;
11
12    printf("Enter a number: ");
13    scanf("%ld", &num);
14
15    while(num != 0)
16    {
17        rem = num % 10;    // get the last digit of num
18        rev = rev * 10 + rem; // reverse the number
19        num = num / 10;    // remove the last digit from num
20    }
21
22    printf("%ld", rev);
23
24    return
25
```

Table generation for n, a^b

In this example, you will learn to generate the multiplication table of a number entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

- C Programming Operators

- **C for Loop**

- *The program below takes an integer input from the user and generates the multiplication tables up to 10.*

-

- **Multiplication Table Up to 10**

```
#include <stdio.h>
int main() {
    int n, i;
    printf("Enter an integer: ");
    scanf("%d", &n);
    for (i = 1; i <= 10; ++i) {
        printf("%d * %d = %d \n", n, i, n * i);
    }
    return 0;
}
```

- **Output**

- *Enter an integer: 9*
- $9 * 1 = 9$
- $9 * 2 = 18$
- $9 * 3 = 27$
- $9 * 4 = 36$
- $9 * 5 = 45$
- $9 * 6 = 54$
- $9 * 7 = 63$
- $9 * 8 = 72$
- $9 * 9 = 81$
- $9 * 10 = 90$

Factorial

Factorial Program in C: Factorial of n is the product of all positive descending integers. Factorial of n is denoted by n!. For example:

1. $5! = 5 * 4 * 3 * 2 * 1 = 120$
2. $3! = 3 * 2 * 1 = 6$

Here, 5! is pronounced as "5 factorial", it is also called "5 bang" or "5 shriek".

The factorial is normally used in Combinations and Permutations (mathematics).

There are many ways to write the factorial program in c language. Let's see the 2 ways to write the factorial program.

- Factorial Program using loop
- Factorial Program using recursion

Factorial Program using loop

Let's see the factorial Program using loop.

```
1. #include<stdio.h>
2. int main()
3. {
4.     int i,fact=1,number;
5.     printf("Enter a number: ");
6.     scanf("%d",&number);
7.     for(i=1;i<=number;i++){
8.         fact=fact*i;
9.     }
10.    printf("Factorial of %d is: %d",number,fact);
11.    return 0;
12.}
```

Output:

```
Enter a number: 5
Factorial of 5 is: 120
```

Factorial Program using recursion in C

Let's see the factorial program in c using recursion.

```
1. #include<stdio.h>
2.
3. long factorial(int n)
4. {
5.     if (n == 0)
6.         return 1;
7.     else
8.         return(n * factorial(n-1));
9. }
```

```
10.
11. void main()
12. {
13. int number;
14. long fact;
15. printf("Enter a number: ");
16. scanf("%d", &number);
17.
18. fact = factorial(number);
19. printf("Factorial of %d is %ld\n", number, fact);
20. return 0;
21. }
```

Output:

```
Enter a number: 6
Factorial of 5 is: 720
```

sine series

Before going to the program for Sine Series first let us understand what is a Sine Series?

Sine Series:

Sine Series is a series which is used to find the value of $\sin(x)$.

where, x is the angle in **degree** which is converted to **Radian**.

The formula used to express the $\sin(x)$ as Sine Series is

Expanding the above notation, the formula of Sine Series is

For example,

Let the value of x be **30**.

So, Radian value for **30** degree is **0.52359**.

So, the value of **$\sin(30)$** is **0.5**.

Program code for Sine Series in C:

?

```
1      #include<stdio.h>
2      #include<conio.h>
3
4      void main()
5      {
6          int i, n;
7          float x, sum, t;
8          clrscr();
9
10         printf(" Enter the value for x : ");
11         scanf("%f",&x);
12
13         printf(" Enter the value for n : ");
14         scanf("%d",&n);
15
16         x=x*3.14159/180;
17         t=x;
18         sum=x;
```

```

19
20      /* Loop to calculate the value of Sine */
21      for(i=1;i<=n;i++)
22      {
23          t=(t*(-1)*x*x)/(2*i*(2*i+1));
24          sum=sum+t;
25      }
26
27      printf(" The value of Sin(%f) = %.4f",x,sum);
28      getch();
29      }

```

Related: C program for Cosine Series

Working:

- First the computer reads the value of 'x' and 'n' from the user.
- Then 'x' is converted to radian value.
- Then using for loop the value of Sin(x) is calculate.
- Finally the value of Sin(x) is printed.

Related: C program for Exponential Series

Step by Step working of the above Program Code:

Let us assume that the user enters the value of 'x' as **45** and 'n' as **4**.

1. Converting 'x' to radian value
 $x = x * 3.14159 / 180$ ($x = 45 * 3.14159 / 180$) So, **x=0.785398**
2. It assigns t=x and sum=x (i.e. **t=0.785398** and **sum=0.785398**)
3. It assigns the value of **i=1** and the loop continues till the condition of the for loop is true.
 - 3.1. $i \leq n$ (**1 ≤ 4**) for loop condition is true
 $t = (0.785398 * (-1) * 0.785398 * 0.785398) / (2 * 1 * (2 * 1 + 1))$

So, **t = - 0.08074**
sum = 0.785398 + (- 0.08074)

So, **sum=0.70465**
i++

So, **i=2**
3.2. $i \leq n$ (**2<=4**) for loop condition is true
 $t = (- 0.08074 * (-1) * 0.785398 * 0.785398) / (2 * 2 * (2 * 2 + 1))$

So, **t = 0.00249**
sum = 0.70465 + 0.00249

So, **sum=0.70714**
i++

So, **i=3**
3.3. $i \leq n$ (**3<=4**) for loop condition is true
 $t = (0.00249 * (-1) * 0.785398 * 0.785398) / (2 * 3 * (2 * 3 + 1))$

So, **t = - 0.000032**
sum = 0.70714 + (- 0.000032)

So, **sum=0.707108**
i++

So, **i=4**
3.4. $i \leq n$ (**4<=4**) for loop condition is true
 $t = (- 0.000032 * (-1) * 0.785398 * 0.785398) / (2 * 4 * (2 * 4 + 1))$

So, **t = 0.000000274**
sum = 0.707108 + 0.000000274

So, **sum=0.707108274**
i++

So, **i=5**
3.5. $i \leq n$ (**5<=4**) for loop condition is false
It comes out of the for loop.

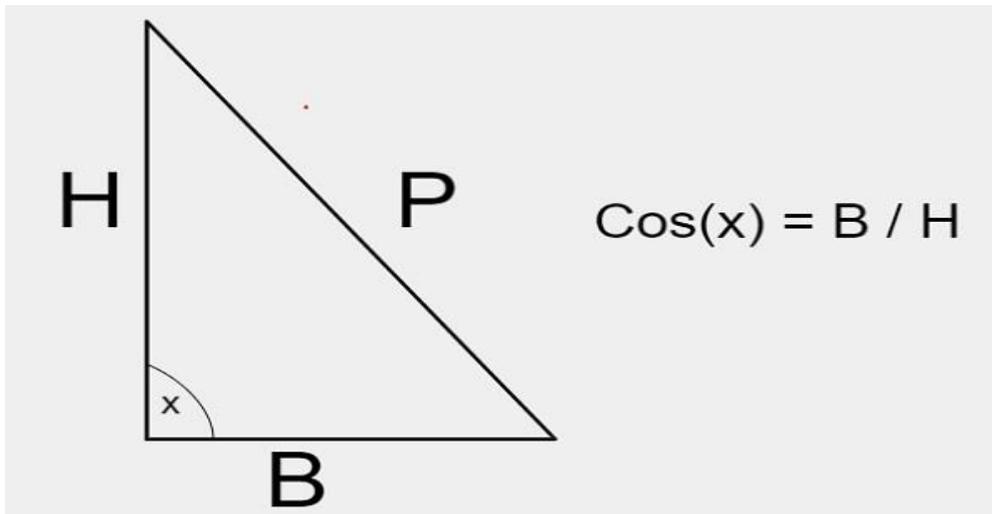
4. Finally it prints **The value of Sin(0.785398) = 0.7071**
5. Thus program execution is completed.

cosine series

terms in the $\cos(x)$ series.

For Cos(x)

Cos(x) is a trigonometric function which is used to calculate the value of x angle.



Formula

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} \quad \sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

For Cos(x) series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

Example

Input-: x = 10, n = 3

Output-: 0.984804

Input-: x = 8, n = 2

Output-: 0.990266

Approach used in the below program is as follows –

- Input the value of x and n
- Apply the formula for calculating cos(x) series
- Print the result as the sum of all the series

Algorithm

Start

Step 1 Declare and initialize const double PI = 3.142

Step 2 In function double series_sum(double x, int n)

Set x = x * (PI / 180.0)

Set result = 1

Set s = 1, fact = 1, pow = 1

Loop For i = 1 and i < 5 and i++

```

    Set s = s * -1
    Set fact = fact * (2 * i - 1) * (2 * i)
    Set pow = pow * x * x
    Set result = result + s * pow / fact
End Loop
Return result
Step 3 In function int main() s
    Declare and set x = 10
    Declare and set n = 3
    Print series_sum(x, n)
Stop

```

Example

```

#include <stdio.h>

const double PI = 3.142;

//will return the sum of cos(x)
double series_sum(double x, int n) {
    x = x * (PI / 180.0);
    double result = 1;
    double s = 1, fact = 1, pow = 1;
    for (int i = 1; i < 5; i++) {
        s = s * -1;
        fact = fact * (2 * i - 1) * (2 * i);
        pow = pow * x * x;
        result = result + s * pow / fact;
    }
    return result;
}

//main function
int main() {
    float x = 10;

```

```
int n = 3;

printf("%lf\n", series_sum(x, n));

return 0;

}
```

Output

```
X=10; n=30.984804
X=13; n=80.974363
X=8; n=2 0.990266
```

ⁿC:-

To convert a nanocoulomb measurement to a coulomb measurement, divide the electric charge by the conversion ratio. One coulomb is equal to 1,000,000,000 nanocoulombs, so use this simple formula to convert:

$$\text{coulombs} = \text{nanocoulombs} \div 1,000,000,000$$

The electric charge in coulombs is equal to the nanocoulombs divided by 1,000,000,000.

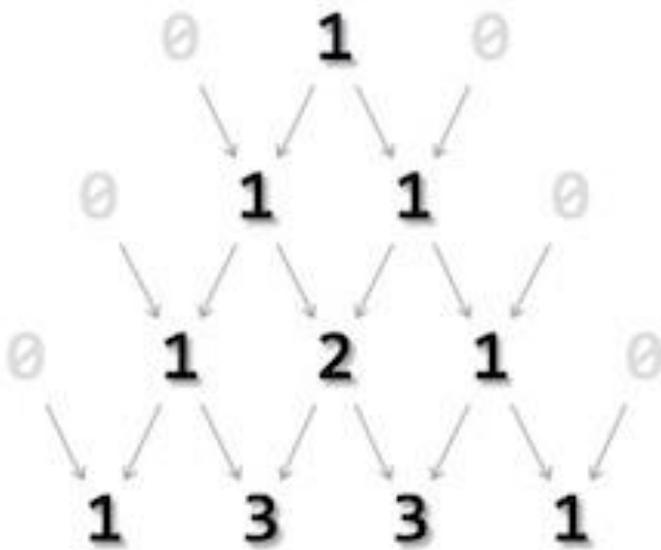
For example, here's how to convert 5,000,000,000 nanocoulombs to coulombs using the formula above.

$$5,000,000,000 \text{ nC} = (5,000,000,000 \div 1,000,000,000) = 5 \text{ C}$$

Nanocoulombs and coulombs are both units used to measure [electric charge](#). Keep reading to learn more about each unit of measure.

Pascal Triangle

Pascal's triangle is one of the classic example taught to engineering students. It has many interpretations. One of the famous one is its use with binomial equations.



All values outside the triangle are considered zero (0). The first row is 0 1 0 whereas only 1 acquires a space in Pascal's triangle, 0s are invisible. The second row is acquired by adding (0+1) and (1+0). The output is sandwiched between two zeroes. The process continues till the required level is achieved.

Pascal's triangle can be derived using the binomial theorem. We can use combinations and factorials to achieve this.

Algorithm

Assuming that we're well aware of factorials, we shall look into the core concept of drawing a Pascal triangle in step-by-step fashion –

START

- Step 1 - Take number of rows to be printed, n.
- Step 2 - Make outer iteration I for n times to print rows
- Step 3 - Make inner iteration for J to (N - 1)
- Step 4 - Print single blank space " "
- Step 5 - Close inner loop
- Step 6 - Make inner iteration for J to I
- Step 7 - Print nCr of I and J
- Step 8 - Close inner loop
- Step 9 - Print NEWLINE character after each inner iteration
- Step 10 - Return

STOP

Pseudocode

We can derive a pseudocode for the above mentioned algorithm, as follows –

```
procedure pascals_triangle
```

```
  FOR I = 0 to N DO  
    FOR J = 0 to N-1 DO  
      PRINT " "  
    END FOR
```

```
  FOR J = 0 to I DO  
    PRINT nCr(i,j)  
  END FOR
```

```
  PRINT NEWLINE  
END FOR
```

```
end procedure
```

Implementation

Let's implement this program in full length. We shall implement functions for factorial (non-recursive) as well ncr (combination).

```
#include <stdio.h>

int factorial(int n) {
    int f;

    for(f = 1; n > 1; n--)
        f *= n;

    return f;
}

int ncr(int n,int r) {
    return factorial(n) / ( factorial(n-r) * factorial(r) );
}

int main() {
    int n, i, j;

    n = 5;

    for(i = 0; i <= n; i++) {
        for(j = 0; j <= n-i; j++)
            printf(" ");

        for(j = 0; j <= i; j++)
            printf(" %3d", ncr(i, j));
```

```

    printf("\n");
}
return 0;
}

```

The output should look like this –

```

    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

```

Prime number

Prime number in C: **Prime number** is a number that is greater than 1 and divided by 1 or itself. In other words, prime numbers can't be divided by other numbers than itself or 1. For example 2, 3, 5, 7, 11, 13, 17, 19, 23.... are the prime numbers.

Note: Zero (0) and 1 are not considered as prime numbers. Two (2) is the only one even prime number because all the numbers can be divided by 2.

Let's see the prime number program in C. In this c program, we will take an input from the user and check whether the number is prime or not.

1. #include<stdio.h>
2. **int** main(){
3. **int** n,i,m=0,flag=0;
4. printf("Enter the number to check prime:");
5. scanf("%d",&n);
6. m=n/2;
7. **for**(i=2;i<=m;i++)
8. {
9. **if**(n%i==0)
10. {
11. printf("Number is not prime");
12. flag=1;
13. **break**;
14. }
15. }
16. **if**(flag==0)

```
17. printf("Number is prime");
18. return 0;
19. }
```

Output:

```
Enter the number to check prime:56
Number is not prime
```

```
Enter the number to check prime:23
Number is prime
```

Factors of a number

Program to find factors of a number is discussed here. Given a number, all the numbers that divide the given number are produced as output.

For example, the factors of number 15 are

$$1 * 15 = 15$$

$$3 * 5 = 15$$

$$5 * 3 = 15$$

$$15 * 1 = 15$$

1, 3, 5, 15 are the factors of 15.

Program to find factors of a number using loops

```

C
C++
Java
Python 3
1
// C program to find factors of a number using loops
2
3

```

```

#include <stdio.h>
4
5
int main()
6
{
7
int num;
8
printf("\nEnter the number : ");
9
scanf("%d",&num);
10
int i,count = 0;
11
printf("\nThe factors of %d are : ",num);
12
for(i = 1;i <= num; i++)
13
{
14
if(num % i == 0)
15
{
16
++count;
17
printf("%d ",i);
18
}
19
}
20
printf("\n\nTotal factors of %d : %d\n",num,count);
21
}

```

Output

```

Input- Enter the number :60 Output- The factors of are 60 :1 2 3 4 5 6 10 12 15 20 30 60
Total factors of 60:12

```

Program to find factors of a number - an efficient approach

```
C
C++
Java
Python 3
1
#include <stdio.h>
2
#include <math.h>
3
int find_factors(int num)
4
{
5
for (int i=1; i<=sqrt(num); i++)
6
{
7
if (num % i == 0)
8
{
9
if (num/i == i)
10
printf("%d ", i);
11
12
else
13
printf("%d %d ", i, num/i);
14
}
15
}
16
}
17
18
int main()
19
{
```

```

                                                                    20
int num;
                                                                    21
printf("\nEnter the number : ");
                                                                    22
scanf("%d",&num);
                                                                    23
int i,count = 0;
                                                                    24
printf("\nThe factors of %d are : ",num);
                                                                    25
find_factors(num);
                                                                    26
                                                                    27
}
                                                                    28
                                                                    29

```

Output

```

Input- Enter the number :60 Output- The factors of are 60 :1 2 3 4 5 6 10 12 15 20 30 60
Total factors of 60:12

```

Other problems such as Perfect number

This is a C Program to check whether a given number is perfect number.

Problem Description

This C Program checks whether a given number is perfect number.

Problem Solution

Perfect number is a number which is equal to sum of its divisor. For eg, divisors of 6 are 1,2 and 3. The sum of these divisors is 6. So 6 is called as a perfect number.

Program/Source Code

Here is source code of the C Program to check whether a given number is perfect number. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

```

/*
 * C Program to Check whether a given Number is Perfect Number
 */
#include <stdio.h>

int main()
{
    int number, rem, sum = 0, i;

    printf("Enter a Number\n");
    scanf("%d", &number);
    for (i = 1; i <= (number - 1); i++)
    {
        rem = number % i;
        if (rem == 0)
        {
            sum = sum + i;
        }
    }
    if (sum == number)
        printf("Entered Number is perfect number");
    else
        printf("Entered Number is not a perfect number");
    return 0;
}

```

Program Explanation

In this C program, we are reading the integer value using 'number' variable. Perfect number is a number which is equal to sum of its divisor. For example, divisors of 6 are 1, 2 and 3. The sum of these divisors is 6. So the number 6 is called as perfect number.

For loop statement is used to assign the modulus of the value of 'number' variable by the value of 'i' variable. If condition statement is used to check the value of 'rem' variable is equal to 0, if the condition is true to execute if condition statement and compute the summation the value of 'sum' variable with the value of 'i' variable.

Another If-else condition statement is used to check that both the value of 'sum' variable and the value of 'number' variable are equal, if the condition is true print the statement as perfect number. Otherwise, execute else condition statement and print the statement as not a perfect number.

Runtime Test Cases

```

Output:
$ cc pgm42.c

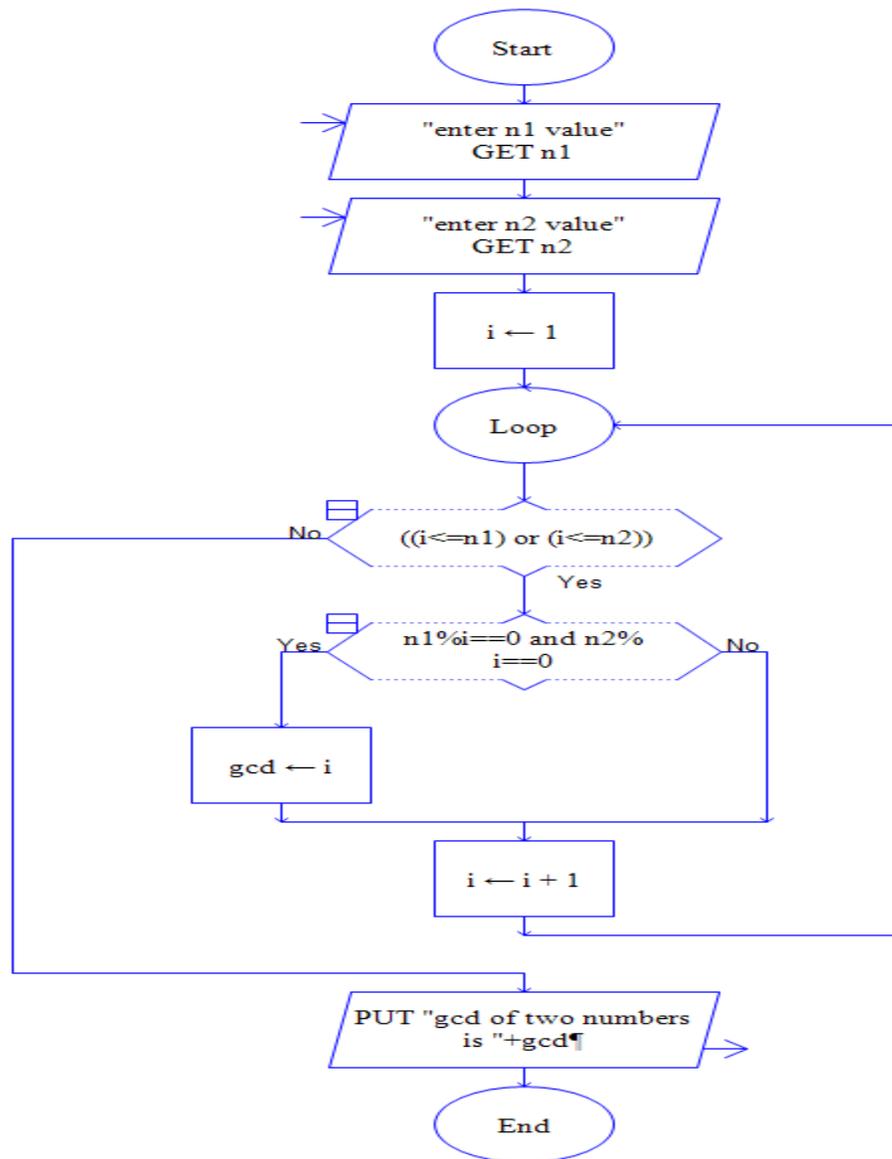
```

```
$ a.out  
Enter a Number  
6  
Entered Number is perfect number
```

```
$ a.out  
Enter a Number  
100  
Entered Number is not a perfect number
```

GCD numbers etc (Write algorithms and draw flowchart)

The Flowchart given here represents the calculation of GCD (Greatest Common Divisor).



The above flowchart is drawn in the Raptor tool. The flowchart represents the flow for finding Greatest Common Divisor

Example: GCD of two numbers 12 & 24 is 12

The code for calculating the LCM and GCD is given in the below link.

Swapping

In this example, you will learn to swap two numbers in C programming using two different techniques.

To understand this example, you should have the knowledge of the following C programming topics:

- C Data Types
- C Programming Operators
- C Input Output (I/O)

Swap Numbers Using Temporary Variable

```
#include<stdio.h>
int main() {
    double first, second, temp;
    printf("Enter first number: ");
    scanf("%lf", &first);
    printf("Enter second number: ");
    scanf("%lf", &second);

    // Value of first is assigned to temp
    temp = first;

    // Value of second is assigned to first
    first = second;

    // Value of temp (initial value of first) is assigned to second
    second = temp;

    printf("\nAfter swapping, firstNumber = %.2lf\n", first);
    printf("After swapping, secondNumber = %.2lf", second);
    return 0;
}
```

Output

```
Enter first number: 1.20
Enter second number: 2.45
```

After swapping, firstNumber = 2.45
After swapping, secondNumber = 1.20

In the above program, the `temp` variable is assigned the value of the `first` variable. Then, the value of the `first` variable is assigned to the `second` variable. Finally, the `temp` (which holds the initial value of `first`) is assigned to `second`. This completes the swapping process.

Swap Numbers Without Using Temporary Variables

```
#include <stdio.h>
int main() {
    double a, b;
    printf("Enter a: ");
    scanf("%lf", &a);
    printf("Enter b: ");
    scanf("%lf", &b);

    // Swapping

    // a = (initial_a - initial_b)
    a = a - b;

    // b = (initial_a - initial_b) + initial_b = initial_a
    b = a + b;

    // a = initial_a - (initial_a - initial_b) = initial_b
    a = b - a;

    printf("After swapping, a = %.2lf\n", a);
    printf("After swapping, b = %.2lf", b);
    return 0;
}
```

Output

Enter a: 10.25

Enter b: -12.5

After swapping, a = -12.50

After swapping, b = 10.25

UNIT-V

Functions

Basic types of function:-

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is

called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

1. return_type function_name(data_type parameter...){
2. //code to be executed
3. }

Types of Functions

There are two types of functions in C programming:

1. **Library Functions:** are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions:** are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

1. **void** hello(){
2. printf("hello c");
3. }

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

1. **int** get(){
2. **return** 10;
3. }

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

1. **float** get(){
2. **return** 10.2;
3. }

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

1. #include<stdio.h>

```
2. void printName();
3. void main ()
4. {
5.     printf("Hello ");
6.     printName();
7. }
8. void printName()
9. {
10.    printf("Javatpoint");
11.}
```

Output

```
Hello Javatpoint
```

Example 2

```
1. #include<stdio.h>
2. void sum();
3. void main()
4. {
5.     printf("\nGoing to calculate the sum of two numbers:");
6.     sum();
7. }
8. void sum()
9. {
10.    int a,b;
11.    printf("\nEnter two numbers");
12.    scanf("%d %d",&a,&b);
13.    printf("The sum is %d",a+b);
14.}
```

Output

```
Going to calculate the sum of two numbers:
```

```
Enter two numbers 10
```

```
24
```

```
The sum is 34
```

Example for Function without argument and with return value

Example 1

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     int result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     result = sum();
8.     printf("%d",result);
9. }
10.int sum()
11.{
12.    int a,b;
13.    printf("\nEnter two numbers");
14.    scanf("%d %d",&a,&b);
15.    return a+b;
16.}
```

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example 2: program to calculate the area of the square

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     printf("Going to calculate the area of the square\n");
6.     float area = square();
7.     printf("The area of the square: %f\n",area);
8. }
9. int square()
10.{
```

```
11. float side;
12. printf("Enter the length of the side in meters: ");
13. scanf("%f",&side);
14. return side * side;
15.}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1

```
1. #include<stdio.h>
2. void sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     sum(a,b);
10.}
11. void sum(int a, int b)
12.{
13. printf("\nThe sum is %d",a+b);
14.}
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the average of five numbers.

```
1. #include<stdio.h>
2. void average(int, int, int, int, int);
```

```

3. void main()
4. {
5.     int a,b,c,d,e;
6.     printf("\nGoing to calculate the average of five numbers:");
7.     printf("\nEnter five numbers:");
8.     scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9.     average(a,b,c,d,e);
10.}
11.void average(int a, int b, int c, int d, int e)
12.{
13.    float avg;
14.    avg = (a+b+c+d+e)/5;
15.    printf("The average of given five numbers : %f",avg);
16.}

```

Output

```

Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000

```

Example for Function with argument and with return value

Example 1

```

1. #include<stdio.h>
2. int sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     result = sum(a,b);
10.    printf("\nThe sum is : %d",result);
11.}
12.int sum(int a, int b)
13.{
14.    return a+b;

```

15.}

Output

```
Going to calculate the sum of two numbers:  
Enter two numbers:10  
20  
The sum is : 30
```

Example 2: Program to check whether a number is even or odd

```
1. #include<stdio.h>  
2. int even_odd(int);  
3. void main()  
4. {  
5.     int n,flag=0;  
6.     printf("\nGoing to check whether a number is even or odd");  
7.     printf("\nEnter the number: ");  
8.     scanf("%d",&n);  
9.     flag = even_odd(n);  
10.    if(flag == 0)  
11.    {  
12.        printf("\nThe number is odd");  
13.    }  
14.    else  
15.    {  
16.        printf("\nThe number is even");  
17.    }  
18.}  
19.int even_odd(int n)  
20.{  
21.    if(n%2 == 0)  
22.    {  
23.        return 1;  
24.    }  
25.    else  
26.    {  
27.        return 0;  
28.    }  
29.}
```

Output

```
Going to check whether a number is even or odd
Enter the number: 100
The number is even
```

C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension `.h`. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include `stdio.h` in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

SN	Header file	Description
1	<code>stdio.h</code>	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	<code>conio.h</code>	This is a console input/output header file.
3	<code>string.h</code>	It contains all string related library functions like <code>gets()</code> , <code>puts()</code> , etc.
4	<code>stdlib.h</code>	This header file contains all the general library functions like <code>malloc()</code> , <code>calloc()</code> , <code>exit()</code> , etc.
5	<code>math.h</code>	This header file contains all the math operations related functions like <code>sqrt()</code> , <code>pow()</code> , etc.
6	<code>time.h</code>	This header file contains all the time-related functions.
7	<code>ctype.h</code>	This header file contains all character handling functions.

8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Declaration and definition

In C++, declaration and definition are often confused. A declaration means (in C) that you are telling the compiler about type, size and in case of function declaration, type and size of its parameters of any variable, or user-defined type or function in your program. No space is reserved in memory for any variable in case of the declaration.

The Definition on the other hand means that in additions to all the things that declaration does, space is additionally reserved in memory. You can say "DEFINITION = DECLARATION + SPACE RESERVATION".

Following are examples of declarations –

```
extern int a;           // Declaring a variable a without defining it
struct _tagExample { int a; int b; }; // Declaring a struct
int myFunc (int a, int b); // Declaring a function
```

While following are examples of definition –

```
int a;
int b = 0;
int myFunc (int a, int b) { return a + b; }
struct _tagExample example;
```

Function call

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 –

```
#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.

Parameter passing

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.

In C, there are two types of parameters and they are as follows...

- **Actual Parameters**
- **Formal Parameters**

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

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

- **Call by Value**

- **Call by Reference**

Call by Value

In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. The changes made on the formal parameters does not effect the values of actual parameters. That means, after the execution control comes back to the calling function, the actual parameter values remains same. For example consider the following program...

Example 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) ;

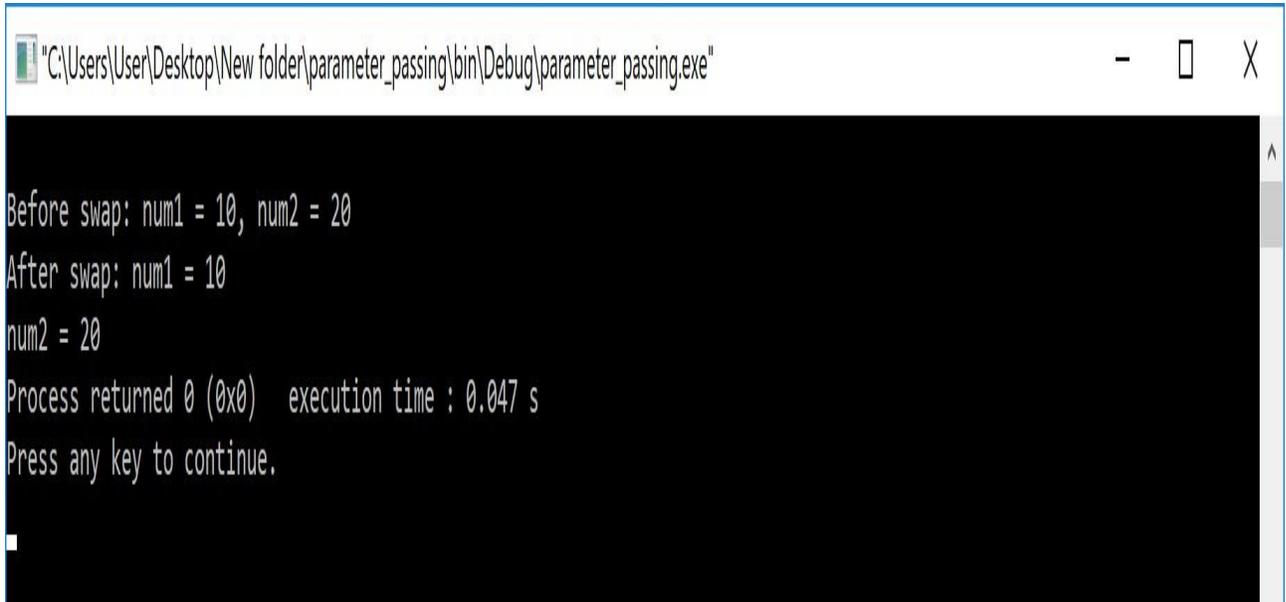
    swap(num1, num2) ; // calling function

    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 ;  
}
```

Output:



```
"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 above 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 recieved 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 following program...

Example 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 ;
}

```

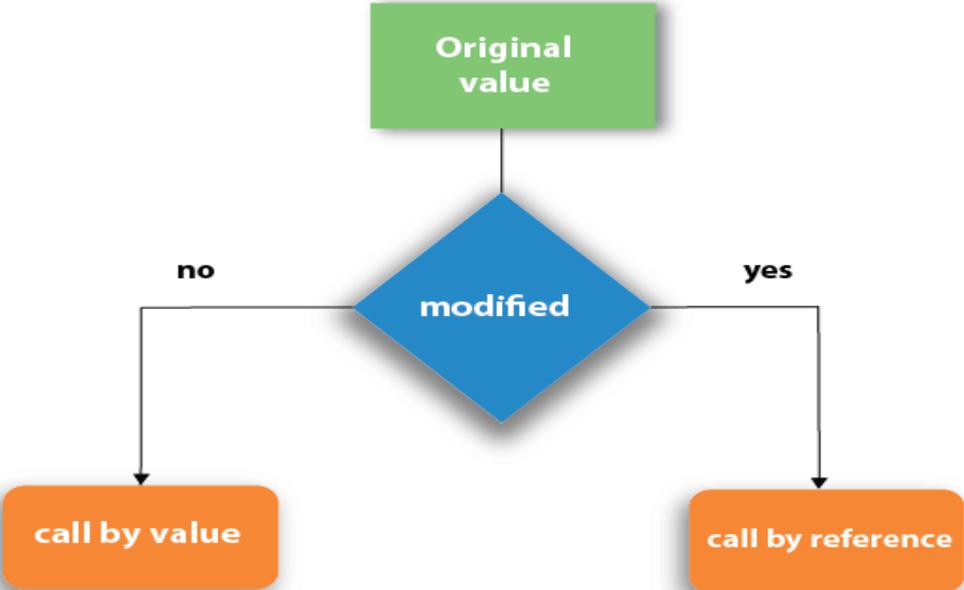
Output:

```
"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 above 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.

Call by value

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12. return 0;
13.}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int , int); //prototype of the function
3. int main()
4. {
```

```

5.  int a = 10;
6.  int b = 20;
7.  printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.  swap(a,b);
9.  printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
10.}
11. void swap (int a, int b)
12.{
13.  int temp;
14.  temp = a;
15.  a=b;
16.  b=temp;
17.  printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters , a = 20, b = 10
18.}

```

Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20

```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```

1. #include<stdio.h>
2. void change(int *num) {
3.  printf("Before adding value inside function num=%d \n",*num);
4.  (*num) += 100;
5.  printf("After adding value inside function num=%d \n", *num);

```

```

6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(&x);//passing reference in function
11.    printf("After function call x=%d \n", x);
12. return 0;
13.}

```

Output

```

Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200

```

Call by reference Example: Swapping the values of the two variables

```

1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
10.}
11.void swap (int *a, int *b)
12.{
13.    int temp;
14.    temp = *a;
15.    *a=*b;
16.    *b=temp;
17.    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
18.}

```

Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10

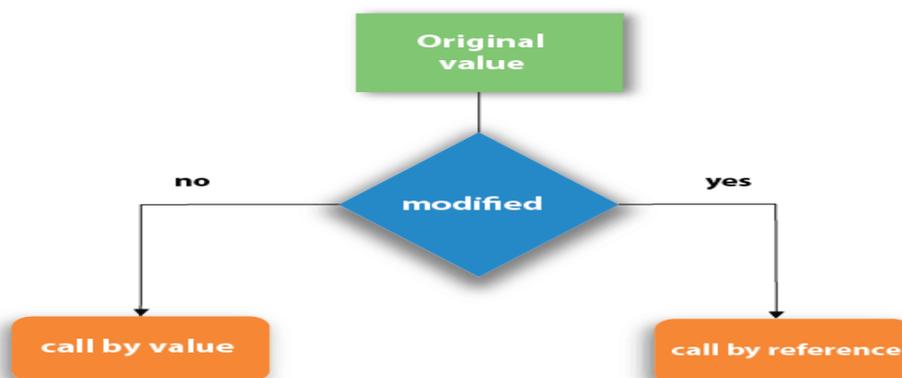
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Call by reference

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12. return 0;
13.}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int , int); //prototype of the function
3. int main()
```

```

4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(a,b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
10.}
11.void swap (int a, int b)
12.{
13.    int temp;
14.    temp = a;
15.    a=b;
16.    b=temp;
17.    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters , a = 20, b = 10
18.}

```

Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20

```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```

1. #include<stdio.h>
2. void change(int *num) {
3.     printf("Before adding value inside function num=%d \n",*num);
4.     (*num) += 100;

```

```

5.   printf("After adding value inside function num=%d \n", *num);
6.   }
7.   int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(&x);//passing reference in function
11.    printf("After function call x=%d \n", x);
12.   return 0;
13.}

```

Output

```

Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200

```

Call by reference Example: Swapping the values of the two variables

```

1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
10.}
11.void swap (int *a, int *b)
12.{
13.   int temp;
14.   temp = *a;
15.   *a=*b;
16.   *b=temp;
17.   printf("After swapping values in function a = %d, b = %d\n", *a,*b); // Formal parameters, a = 20, b = 10
18.}

```

Output

```

Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10

```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Scope of variable

A scope is a region of the program and broadly speaking there are three places, where variables can be declared –

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

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

Local Variables

Variables that are declared inside a function or block are 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. Following is the example using local variables –

```
#include <iostream>
using namespace std;

int main () {
    // Local variable declaration:
    int a, b;
    int c;

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

    cout << c;

    return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your 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. Following is the example using global and local variables –

```
#include <iostream>
using namespace std;

// Global variable declaration:
int g;

int main () {
    // Local variable declaration:
    int a, b;

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

    cout << g;

    return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example –

```
#include <iostream>
using namespace std;

// Global variable declaration:
int g = 20;

int main () {
    // Local variable declaration:
    int g = 10;

    cout << g;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –
10

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	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

Storage classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program –

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

    while(count-->0) {
        func();
    }

    return 0;
}

/* function definition */
void func( void ) {

    static int i = 5; /* local static variable */
    i++;

    printf("i is %d and count is %d\n", i, count);
}
```

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

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
    count = 5;
    write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
    printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows –

```
$gcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it produces the following result –

```
count is 5
```

Recursion

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
1. #include <stdio.h>
2. int fact (int);
3. int main()
4. {
5.     int n,f;
6.     printf("Enter the number whose factorial you want to calculate?");
7.     scanf("%d",&n);
8.     f = fact(n);
9.     printf("factorial = %d",f);
10.}
11.int fact(int n)
12.{
13.    if (n==0)
14.    {
15.        return 0;
16.    }
17.    else if ( n == 1)
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return n*fact(n-1);
```

```
24. }  
25.}
```

Output

```
Enter the number whose factorial you want to calculate?5  
factorial = 120
```

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
1. if (test_for_base)  
2. {  
3.   return some_value;  
4. }  
5. else if (test_for_another_base)  
6. {  
7.   return some_another_value;  
8. }  
9. else  
10. {  
11.  // Statements;  
12.  recursive call;  
13. }
```

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```
1. #include<stdio.h>  
2. int fibonacci(int);  
3. void main ()  
4. {  
5.   int n,f;
```

```

6.  printf("Enter the value of n?");
7.  scanf("%d",&n);
8.  f = fibonacci(n);
9.  printf("%d",f);
10.}
11.int fibonacci (int n)
12.{
13.  if (n==0)
14.  {
15.    return 0;
16.  }
17.  else if (n == 1)
18.  {
19.    return 1;
20.  }
21.  else
22.  {
23.    return fibonacci(n-1)+fibonacci(n-2);
24.  }
25.}

```

Output

```

Enter the value of n?12
144

```

Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```

1. int display (int n)
2. {
3.  if(n == 0)
4.    return 0; // terminating condition
5.  else

```

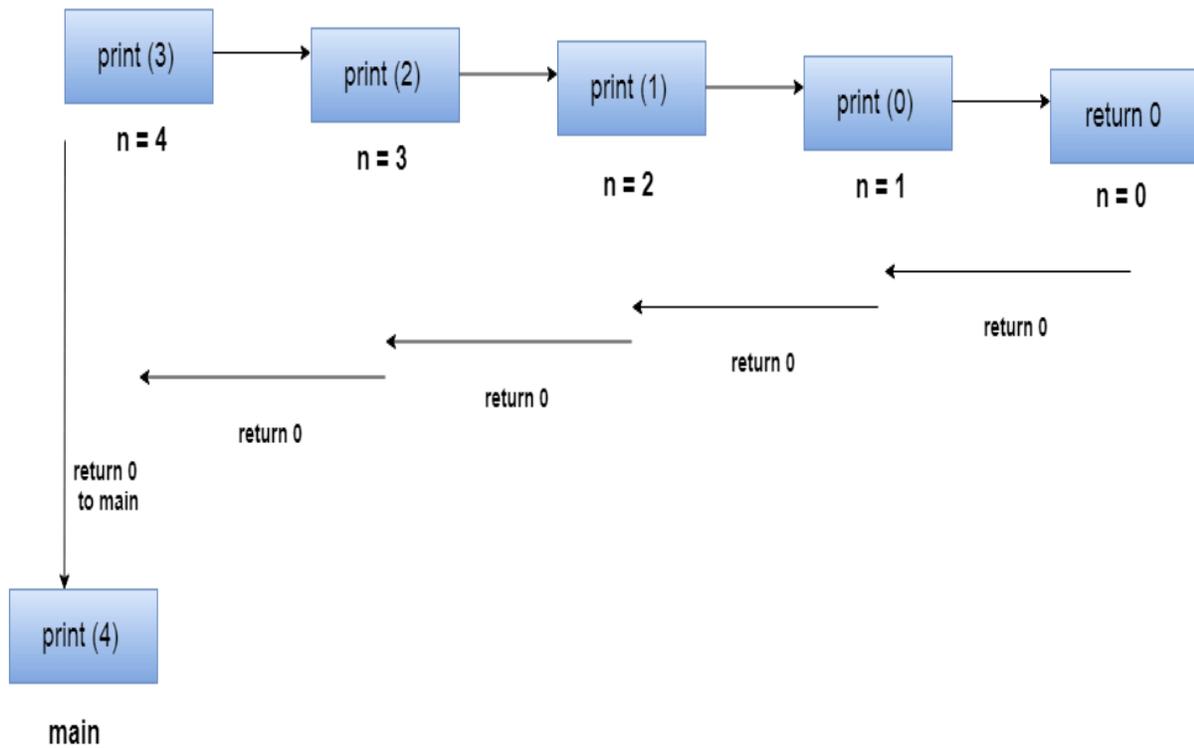
```

6.  {
7.    printf("%d",n);
8.    return display(n-1); // recursive call
9.  }
10.}

```

Explanation

Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

Recursion in C language is basically the process that describes the action when a function calls a copy of itself in order to work on a smaller problem. Recursive functions are the functions that calls themselves and these type of function calls are known as recursive calls. The recursion in C generally involves various numbers of recursive calls. It is considered to be very important to impose a termination condition of recursion.

Recursion code in the C language is generally shorter than the iterative code and it is known to be difficult to understand.

Recursion cannot be applied to all the problem, but Recursion in C language is very useful for the tasks that can be generally be defined in terms of similar subtasks but it cannot be applied to all the problems. For instance: recursion in C can be applied to sorting, searching, and traversal problems.

As function call is always overhead, iterative solutions are more efficient than recursion. Any of the problem that can generally be solved recursively, it can be also solved iteratively. Apart from these facts, there are some problems that are best suited to only be solved by the recursion for instance: tower of Hanoi, factorial finding, Fibonacci series, etc.

Write a program to calculate the factorial number

```
#include <stdio.h>
int factorial (int);
int main()
{
    int num,fact;
    printf("Enter any number to find factorial ");
    scanf("%d",&num);
    fact = factorial(num);
    printf("factorial = %d",fact);
}
int factorial(int num)
{
    if (num==0)
    {
        return 0;
    }
    else if ( num == 1)
    {
        return 1;
    }
    else
    {
        return num*factorial(num-1);
    }
}
}
```

Output : Enter any number to find factorial 5

factorial = 120

What is Recursive Function

The working of a recursive function involves the tasks by dividing them generally into the subtasks. Some specific subtask have a termination condition defined that has to be satisfied by them. In the next step, the recursion in C stops and the final result is derived from the function.

The base case is the case at which the function doesn't recur in C and there are instances where the function keeps calling itself in order to perform a subtask and that is known as the recursive case. Here is the following format in which all the recursive functions can be written:

Example of Recursive function

Write a Program to print 10 number of Fibonacci Series

```
#include<stdio.h>
int fibo(int);
void main ()
{
    int x,f;
    printf("Enter value of n number?");
    scanf("%d",&x);
    f = fibo(x);
    printf("%d",f);
}
int fibo (int x)
{
    if (x==0)
    {
        return 0;
    }
    else if (x == 1)
    {
        return 1;
    }
    else
    {
        return fibo(x-1)+fibo(x-2);
    }
}
```