Department of Commerce & Business Management
University Arts & Science College

Programming with C & C++

[Type the document subtitle]

K. Phanindra Kumar
Asst. Professor
Dept. of Commerce & Busi. Management
University Arts & Science College
(Autonomous)
Kakatiya University, Warangal.

Department of Commerce & Business Management, Kakatiya University, Warangal.

Paper DSC 203: PROGRAMMING WITH C & C++

Hours Per Week: 5	Credits: 5
Exam Hours: 1 ½	Marks: 50U+35P+15I

Objective: *Fundamental Concepts of Programming in C and Object Oriented Programming in C++.*

UNIT-I: **Introduction:** Computer of Languages- Flow charts-algorithms–History of C language – Basic Structure–Programming Rules –Commonly used library functions - Executing the C Program - Pre-processors in "C"- Keywords & Identifiers – Constants – **Variables:** Rules for defining variables - Scope and Life of a Variable--**Data types** - Type Conversion - Formatted Input and Output operations. **Operators:** Introduction – Arithmetic – Relational – Logical – Assignment - Conditional - Special - Bitwise -Increment / Decrement operator.

UNIT-II: Conditional statements: Introduction - If statements - If-else statements – nested if-else – break statement-continue and exit-statement - goto-statement-Switch statements. **Looping statements:** Introduction-While statements – Do-while statements - For Statements-nested loop statements.

UNIT-III: Functions: Definition and declaration of functions- Function proto type-return statement- types of functions and Built-in functions. **User-defined functions:** Introduction-Need for user defined Function and Components of functions. **Arrays:** Introduction-Defining an array-Initializing an array-One dimensional array- Multi dimensional array. **Strings:** Introduction–Declaring and initializing string- and Handling Strings -String handling functions. **Pointers:** Features of pointers- Declaration of Pointers-advantages of pointers.

UNIT-IV: Structures: Features of Structures - Declaring and initialization of Structures – Structure within Structure-Array of Structures- Enumerated data type. **Unions** - Definition and advantages of Unions comparison between Structure & Unions.

Object Oriented Programming: Introduction to Object Oriented Programming - Structure of C++ –Simple program of C++-Differences between C & C++

UNIT-V: Classes and Objects: Data Members-Member Functions - Object Oriented- Class-Object- Encapsulation-Abstraction concepts-Polymorphism (Function overloading and Operator Overloading) Inheritance- (Inheritance Forms and Inheritance Types).

UNIT-I: INTRODUCTION TO C LANGUAGE, VARIABLES, DATA TYPES AND OPERATORS

Introduction: *Types of Languages- History of C language – Basic Structure –Programming Rules – Flow charts-algorithms–Commonly used library functions - Executing the C Program - Pre-processors in "C"- Keywords & Identifiers – Constants –* ***Variables:*** *Rules for defining variables - Scope and Life of a Variable–-* ***Data types*** *- Type Conversion - Formatted Input and Output operations.* ***Operators:*** *Introduction – Arithmetic – Relational – Logical – Assignment - Conditional - Special - Bitwise - Increment / Decrement operator.*

INTRODUCTION:

A Computer is an electronic device that stores, manipulates and retrieves the data.

A Computer System is a group of several objects with a process. The following are the objects of computer System:

a) User (A person who uses the computer)

b) Hardware

c) Software

Hardware: Hardware of a computer system can be referred as anything which we can touch and feel. Example: Keyboard and Mouse. The hardware of a computer system can be classified as Input Devices (I/P) Processing Devices (CPU) Output Devices (O/P)

Software: Software of a computer system can be referred as anything which we can feel and see. Example: Windows, icons Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.

System Software: System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The operating system provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, system development software, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

Application software: Application software is broken in to two classes: general-purpose software and application – specific software. General purpose software is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors, database management systems ,and computer aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

Application –specific software can be used only for its intended purpose.

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks. The relationship between system and application software is shown below. In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.

COMPUTER LANGUAGES

To write a program (tells what to do) for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages.

The following is the summary of computer languages

1940‘s -- Machine Languages
1950‘s -- Symbolic Languages
1960‘s -- High Level Languages

Machine Language:

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made of streams of 0‘s and 1‘s. The instructions in machine language must be in streams of 0‘s and 1‘s. This is also referred as binary digits. These are so named as the machine can directly understood the programs.

Advantages:

High speed execution
The computer can understood instructions immediately
No translation is needed.

Disadvantages:

Machine dependent
Programming is very difficult
Difficult to understand
Difficult to write bug free programs
Difficult to isolate an error

Assembly Language:

In the early 1950‘s Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions. These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language. A special program called an Assembler translates symbolic code into the machine language. Hence they are called as Assembly language.

Advantages:

Easy to understand and use
Easy to modify and isolate error
High efficiency
More control on hardware

Disadvantages:

Machine Dependent Language
Requires translator
Difficult to learn and write programs
Slow development time
Less efficient

High-Level Languages

The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of high-level languages.

High-level languages are portable to many different computer allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.

C → A systems implementation Language
C++ → C with object oriented enhancements
JAVA → Object oriented language for internet and general applications using basic C syntax

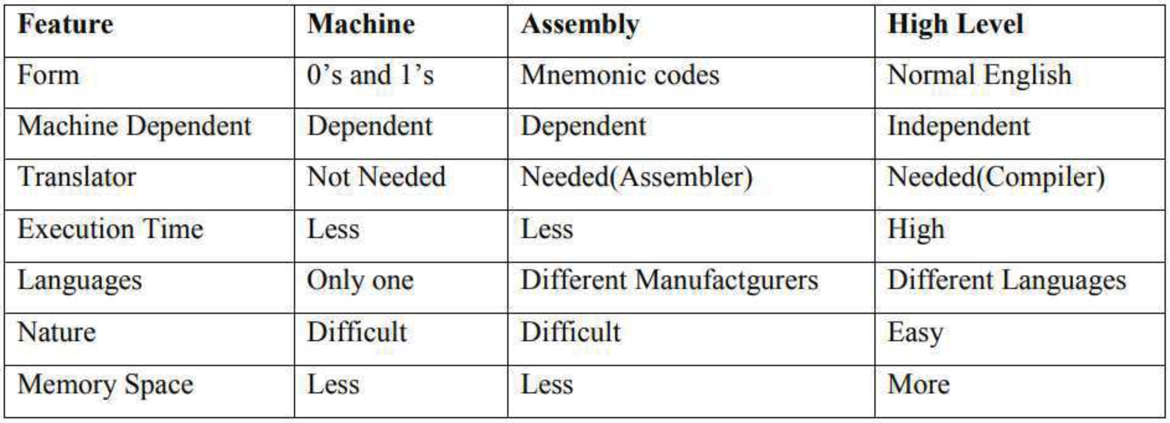
Advantages:

Easy to write and understand
Easy to isolate an error
Machine independent language
Easy to maintain
Better readability
Low Development cost
Easier to document
Portable

Disadvantages:

Needs translator
Requires high execution time
Poor control on hardware
Less efficient

Difference between Machine, Assembly, High Level Languages



Feature	Machine	Assembly	High Level
Form	0's and 1's	Mnemonic codes	Normal English
Machine Dependent	Dependent	Dependent	Independent
Translator	Not Needed	Needed(Assembler)	Needed(Compiler)
Execution Time	Less	Less	High
Languages	Only one	Different Manufactgurers	Different Languages
Nature	Difficult	Difficult	Easy
Memory Space	Less	Less	More

Language Translators

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be excuted by the computer.

1) Compiler: It is a program which is used to convert the high level language programs into machine language.

2) Assembler: It is a program which is used to convert the assembly level language programs into machine language.

3) Interpreter: It is a program, it takes one statement of a high level language program, translates it into machine language instruction and then immediately executes the resulting machine language instruction and so on.

Introduction to C Language

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the UNIX operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972.

The UNIX operating system and virtually all Unix applications are written in the C language. C has now become a widely used professional language for various reasons.

- ✓ Easy to learn
- ✓ Structured language
- ✓ It produces efficient programs.
- ✓ It can handle low-level activities.
- ✓ It can be compiled on a variety of computers.

Facts about C

- C was invented to write an operating system called UNIX
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the (ANSI)
- By 1973 UNIX OS almost totally written in C
- Today C is the most widely used System Programming Language
- Most of the state of the art software has been implemented using C

Why to use C?

C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language.

Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C Program File

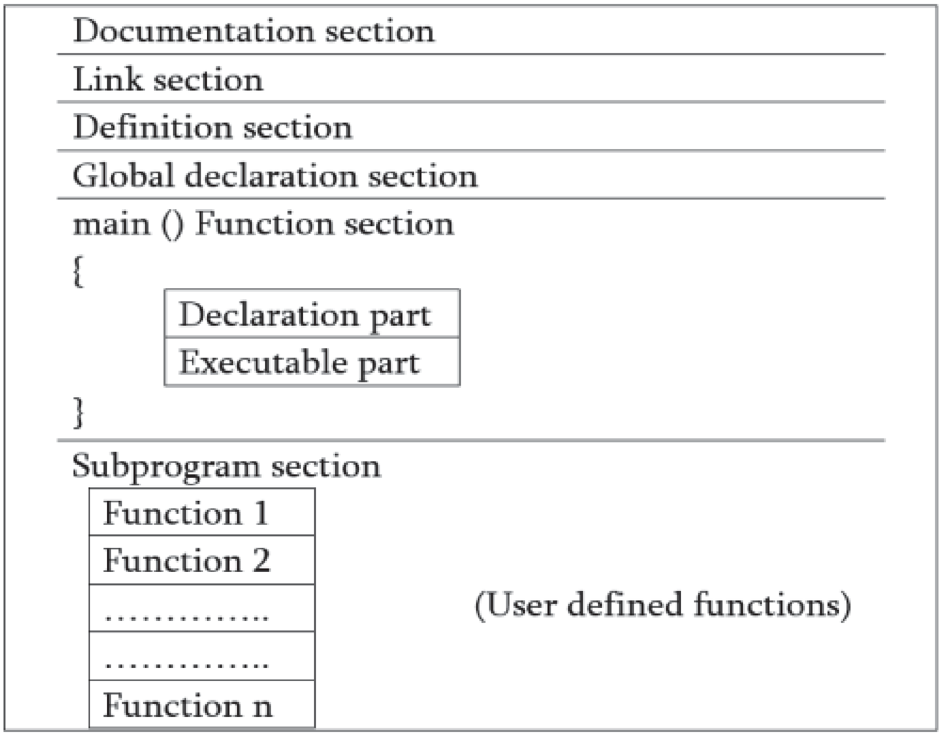
All the C programs are writen into text files with extension ".c" for example hello.c. You can use "vi" editor to write your C program into a file.

HISTORY TO C LANGUAGE

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C. Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP7. BCPL and B are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

BASIC STRUCTURE OF C PROGRAMMING



1. **Documentation section**: The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

Example

```
/**
* File Name: Helloworld.c
* Author: Manthan Naik
* date: 09/08/2019
* description: a program to display hello world
*           no input needed
*/
```

2. **Link section:** The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.

   Example

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

3. **Definition section**: The definition section defines all symbolic constants such using the #define directive. In this section, we define different constants. The keyword define is used in this part.

   ```
   #define PI=3.14
   ```

4. **Global declaration section**: There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global

declaration section that is outside of all the functions. This section also declares all the user-defined functions.

The user-defined functions are also declared in this part of the code.

```
float area(float r);
int a=7;
```

5. **main () function section**: Every C program must have one main function section. This section contains two parts; declaration part and executable part

   1. Declaration part: The declaration part declares all the variables used in the executable part.

   2. Executable part: There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

   Both the declaration and execution part are inside the curly braces.

   ```
   int main(void)
   {
   int a=10;
   printf(" %d", a);
   return 0;
   }
   ```

6. **Subprogram section**: If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

   Sub Program Section

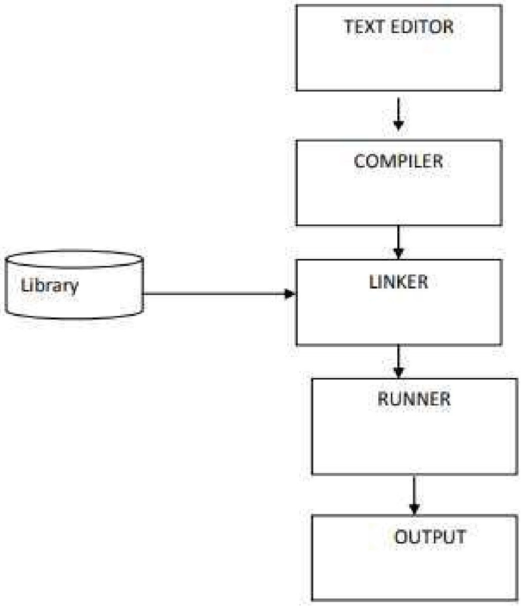
   All the user-defined functions are defined in this section of the program.

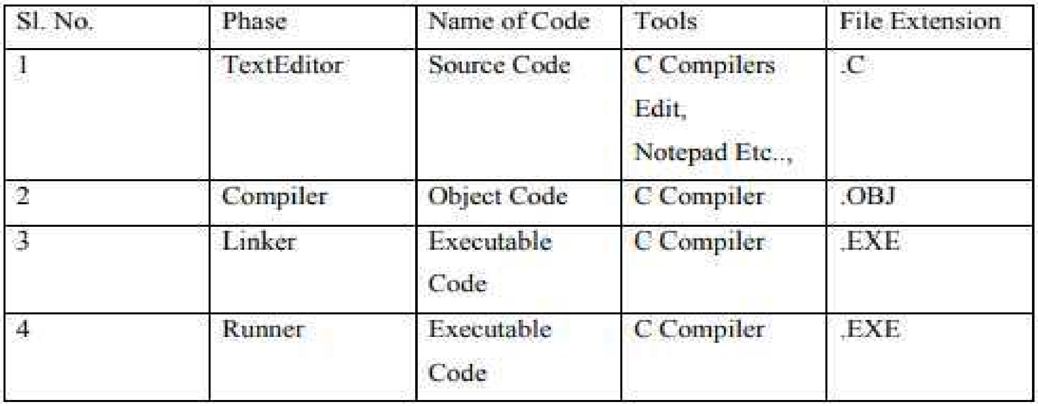
   ```
   int add(int a, int b)
   {
   return a+b;
   }
   ```

CREATING AND RUNNING PROGRAMS

The procedure for turning a program written in C into machine Language. The process is presented in a straightforward, linear fashion but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code. The following are the four steps in this process

1) Writing and Editing the program
2) Compiling the program
3) Linking the program with the required modules
4) Executing the program





Sl. No.	Phase	Name of Code	Tools	File Extension
1	TextEditor	Source Code	C Compilers Edit, Notepad Etc..,	.C
2	Compiler	Object Code	C Compiler	.OBJ
3	Linker	Executable Code	C Compiler	.EXE
4	Runner	Executable Code	C Compiler	.EXE

Writing and Editing Programs:

The software used to write programs is known as a text editor. A text editor helps us enter, change and store character data. Once we write the program in the text editor we save it using a filename stored with an extension of .C. This file is referred as source code file.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The Compiler is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language. This translation process is called compilation. The entire high level program is converted into the executable machine code file. The Compiler which executes C programs is called as C Compiler.

Example Turbo C, Borland C, GC etc.,

The C Compiler is actually two separate programs:

The Preprocessor
The Translator

The Preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries. The result of preprocessing is called the translation unit.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in the machine language.

Linking Programs

The Linker assembles all functions, the program‘s functions and system‘s functions into one executable program.

Executing Programs

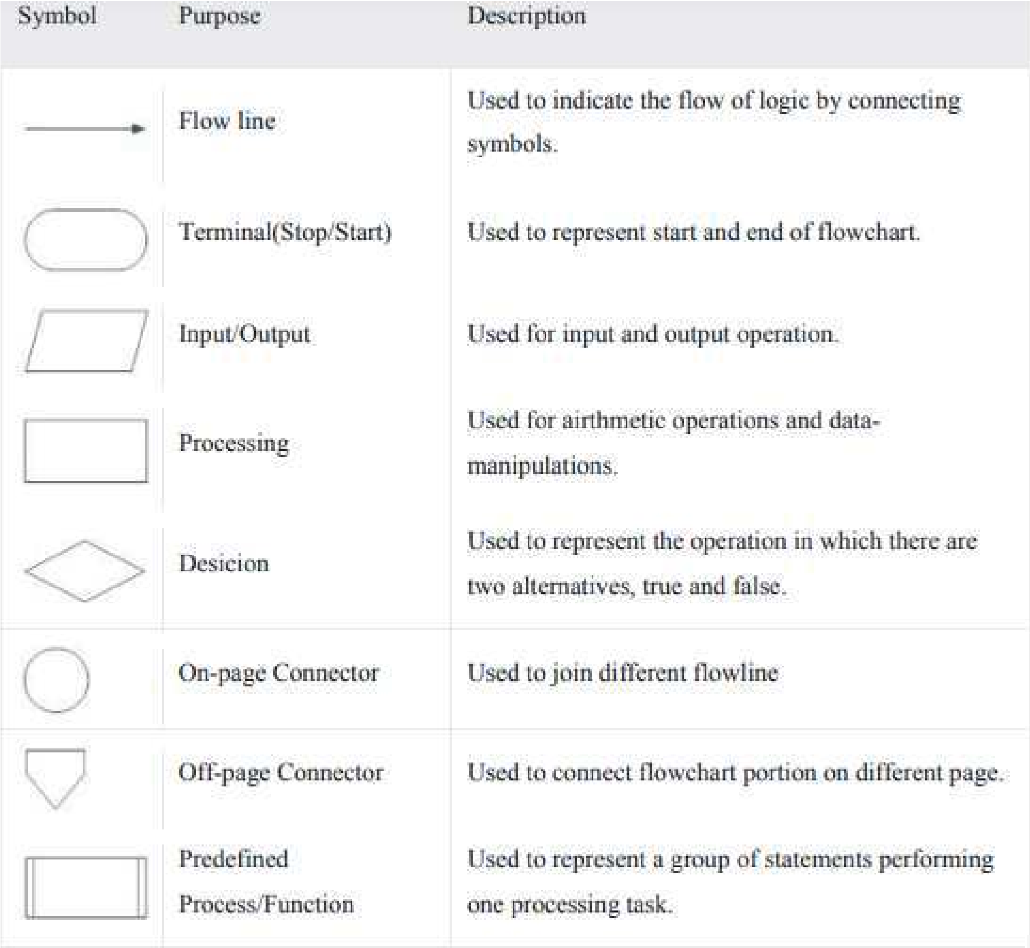
To execute a program we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the loader. It locates the executable program and reads it into memory. When everything is loaded the program takes control and it begins execution.

FLOWCHARTS:

Flowchart is a diagrammatic representation of an algorithm. Flowchart is very helpful in writing program and explaining program to others.

Symbols Used In Flowchart

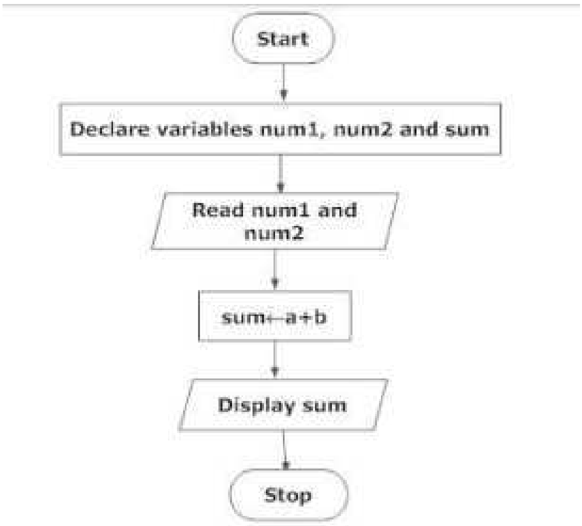
Different symbols are used for different states in flowchart, For example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart



Symbol	Purpose	Description
→	Flow line	Used to indicate the flow of logic by connecting symbols.
⬭	Terminal(Stop/Start)	Used to represent start and end of flowchart.
▱	Input/Output	Used for input and output operation.
▭	Processing	Used for airthmetic operations and data-manipulations.
◇	Desicion	Used to represent the operation in which there are two alternatives, true and false.
○	On-page Connector	Used to join different flowline
⬠	Off-page Connector	Used to connect flowchart portion on different page.
▥	Predefined Process/Function	Used to represent a group of statements performing one processing task.

Examples of flowcharts in programming

Draw a flowchart to add two numbers entered by user.



ALGORITHM

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements

The ordered set of instructions required to solve a problem is known as an algorithm. The characteristics of a good algorithm are:

Precision – the steps are precisely stated (defined).

Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.

Finiteness – the algorithm stops after a finite number of instructions are executed.

Input – the algorithm receives input.

Output – the algorithm produces output.

Generality – the algorithm applies to a set of inputs.

Example

Q. Write a algorithm to find out number is odd or even?

Ans.
```
Step 1 : start
step 2 : input number
Step 3 : rem=number mod 2
Step 4 : if rem=0 then
         print "number even"
         else
              print "number odd"
         endif
Step 5 : stop
```

COMMONLY USED LIBRARY FUNCTIONS:

C Standard library functions or simply C Library functions are inbuilt functions in C programming.

The prototype and data definitions of these functions are present in their respective header files. To use these functions we need to include the header file in our program. For example, If you want to use the printf() function, the header file <stdio.h> should be included.

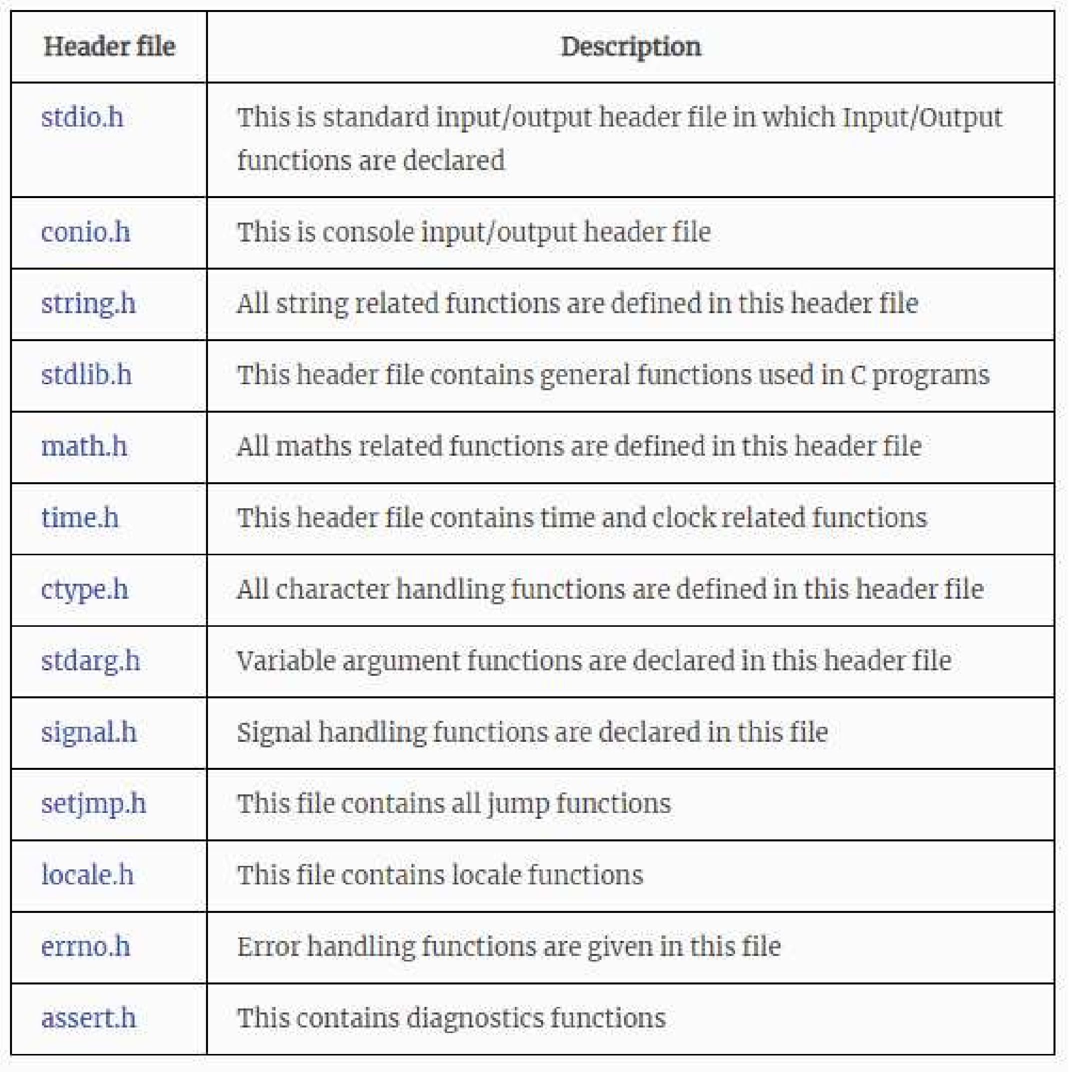
```c
#include <stdio.h>
int main()
{
   printf("Catch me if you can.");
}
```

If you try to use printf() without including the stdio.h header file, you will get an error.

LIST OF MOST USED HEADER FILES IN C PROGRAMMING LANGUAGE:

- Check the below table to know all the C library functions and header files in which they are declared.

- Click on the each header file name below to know the list of inbuilt functions declared inside them.



Header file	Description
stdio.h	This is standard input/output header file in which Input/Output functions are declared
conio.h	This is console input/output header file
string.h	All string related functions are defined in this header file
stdlib.h	This header file contains general functions used in C programs
math.h	All maths related functions are defined in this header file
time.h	This header file contains time and clock related functions
ctype.h	All character handling functions are defined in this header file
stdarg.h	Variable argument functions are declared in this header file
signal.h	Signal handling functions are declared in this file
setjmp.h	This file contains all jump functions
locale.h	This file contains locale functions
errno.h	Error handling functions are given in this file
assert.h	This contains diagnostics functions

Advantages of Using C library functions

1. They work: One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance: Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time: Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable: With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

PRE-PROCESSORS IN “C”:
KEYWORDS AND IDENTIFIERS

Character set

A character set is a set of alphabets, letters and some special characters that are valid in C language.

Alphabets

Uppercase: A B C .... X Y Z
Lowercase: a b c .... x y z

C accepts both lowercase and uppercase alphabets as variables and functions.

Digits

0 1 2 3 4 5 6 7 8 9

Special Characters

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

Special Characters in C Programming

White space Characters

Blank space, newline, horizontal tab, carriage, return and form feed.

C KEYWORDS

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

1. int money;

Here, int is a keyword that indicates money is a variable of type int (integer).

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

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

C Keywords

All these keywords, their syntax, and application will be discussed in their respective topics. However, if you want a brief overview of these keywords without going further, visit List of all keywords in C programming.

C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc. Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

1. int money;
2. double accountBalance;

Here, money and accountBalance are identifiers.

Also remember, identifier names must be different from keywords. You cannot use int as an identifier because int is a keyword.

Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.

3. You cannot use keywords as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

You can choose any name as an identifier if you follow the above rule, however, give meaningful names to identifiers that make sense.

Variables, Constants and Literals

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 have only 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: firstNameis 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.

Constants

If you want to define a variable whose value cannot be changed, you can use the constkeyword. 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.

```
1. const double PI = 3.14;
2. PI = 2.9; //Error
```

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	Character
\b	Backspace

Escape Sequences	Character
\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

Escape Sequences

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
```

DATA TYPES

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

1. 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
int	at least 2, usually 4	%d
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%li
long long int	at least 8	%lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
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.

1. int id;

Here, id is a variable of type integer.

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

1. 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;
```

What's the difference between float and double?

The size of float (single precision float data type) is 4 bytes. And the size of double (double precision float data type) is 8 bytes.

char

Keyword char is used for declaring character type variables. For example,

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

void is an incomplete type. It means "nothing" or "no type". You can think of void as **absent**.

For example, if a function is not returning anything, its return type should be void.

Note that, you cannot create variables of void type.

short and long

If you need to use a large number, you can use a type specifier long. Here's how:

```
long a;
long long b;
long double c;
```

Here variables a and b can store integer values. And, c can store a floating-point number.

If you are sure, only a small integer ([−32,767, +32,767] range) will be used, you can use short.

```
short d;
```

You can always check the size of a variable using the sizeof() operator.

```
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
  printf("size of long long = %d bytes\n", sizeof(c));
  printf("size of long double= %d bytes\n", sizeof(d));
  return 0;
}
```

signed and unsigned

In C, signed and unsigned are type modifiers. You can alter the data storage of a data type by using them. For example,

```
unsigned int x;
int y;
```

Here, the variable x can hold only zero and positive values because we have used the unsigned modifier.

Considering the size of int is 4 bytes, variable y can hold values from -2^{31} to $2^{31}-1$, whereas variable x can hold values from 0 to $2^{32}-1$.

Other data types defined in C programming are:

- bool Type
- Enumerated type
- Complex types

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

Input Output (I/O)

C Input

In C programming, scanf() is one of the commonly used function to take input from the user. The scanf() function reads formatted input from the standard input such as keyboards.

Example 5: Integer Input/Output

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d",testInteger);
    return 0;
}
```

Output

```
Enter an integer: 4
Number = 4
```

Here, we have used %d format specifier inside the scanf() function to take int input from the user. When the user enters an integer, it is stored in the testInteger variable.

Notice, that we have used &testInteger inside scanf(). It is because &testInteger gets the address of testInteger, and the value entered by the user is stored in that address.

Example 6: Float and Double Input/Output

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```

Output

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
num2 = 10.200000
```

We use %f and %lf format specifier for float and double respectively.

Example 7: C Character I/O

```
#include <stdio.h>
int main()
{
   char chr;
   printf("Enter a character: ");
   scanf("%c",&chr);
   printf("You entered %c.", chr);
   return 0;
}
```

Output

```
Enter a character: g
You entered g.
```

When a character is entered by the user in the above program, the character itself is not stored. Instead, an integer value (ASCII value) is stored.

And when we display that value using %c text format, the entered character is displayed. If we use %d to display the character, it's ASCII value is printed.

Example 8: ASCII Value

```
#include <stdio.h>
int main()
{
   char chr;
   printf("Enter a character: ");
   scanf("%c", &chr);

   // When %c is used, a character is displayed
   printf("You entered %c.\n",chr);

   // When %d is used, ASCII value is displayed
   printf("ASCII value is % d.", chr);
   return 0;
}
```

Output

```
Enter a character: g
You entered g.
ASCII value is 103.
```

I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

```
#include <stdio.h>
int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");

    // Taking multiple inputs
    scanf("%d%f", &a, &b);

    printf("You entered %d and %f", a, b);
    return 0;
}
```

Output

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

Format Specifiers for I/O

As you can see from the above examples, we use

- %d for int
- %f for float
- %lf for double
- %c for char

Here's a list of commonly used C data types and their format specifiers.

Data Type	Format Specifier
int	%d
char	%c
float	%f
double	%lf
short int	%hd

Data Type	Format Specifier
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

C Output

In C programming, printf() is one of the main output function. The function sends formatted output to the screen. For example,

Example 1: C Output

```
#include <stdio.h>
int main()
{
  // Displays the string inside quotations
  printf("C Programming");
  return 0;
}
```

Output

```
C Programming
```

How does this program work?

- All valid C programs must contain the main() function. The code execution begins from the start of the main() function.
- The printf() is a library function to send formatted output to the screen. The function prints the string inside quotations.
- To use printf() in our program, we need to include stdio.h header file using the #include <stdio.h> statement.
- The return 0; statement inside the main() function is the "Exit status" of the program. It's optional.

Example 2: Integer Output

```
#include <stdio.h>
int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

Output

```
Number = 5
```

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of testInteger.

Example 3: float and double Output

```
#include <stdio.h>
int main()
{
    float number1 = 13.5;
    double number2 = 12.4;

    printf("number1 = %f\n", number1);
    printf("number2 = %lf", number2);
    return 0;
}
```

Output

```
number1 = 13.500000
number2 = 12.400000
```

To print float, we use %f format specifier. Similarly, we use %lf to print double values.

Example 4: Print Characters

```
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c.", chr);
    return 0;
}
```

Output

```
character = a
```

To print char, we use %c format specifier.

OPERATORS

An operator is a symbol that operates on a value or a variable. For example: + is an operator to perform addition.
C has a wide range of operators to perform various operations.

C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;

   c = a+b;
   printf("a+b = %d \n",c);
   c = a-b;
   printf("a-b = %d \n",c);
   c = a*b;
   printf("a*b = %d \n",c);
   c = a/b;
   printf("a/b = %d \n",c);
   c = a%b;
   printf("Remainder when a divided by b = %d \n",c);

   return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators +, - and * computes addition, subtraction, and multiplication respectively as you might have expected.

In normal calculation, 9/4 = 2.25. However, the output is 2 in the program.

It is because both the variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a=9 is divided by b=4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

```
// Either one of the operands is a floating-point number
a/b = 2.5
a/d = 2.5
c/b = 2.5

// Both operands are integers
c/d = 2
```

C Increment and Decrement Operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--. Visit this page to learn more about how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

```

```
   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
   printf("%d >= %d is %d \n", a, b, a >= b);
   printf("%d >= %d is %d \n", a, c, a >= c);
   printf("%d <= %d is %d \n", a, b, a <= b);
   printf("%d <= %d is %d \n", a, c, a <= c);

   return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.

Operator	Meaning	Example
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

Example 5: Logical Operators

```
// Working of logical operators

#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) is %d \n", result);
   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) is %d \n", result);
   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) is %d \n", result);
   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) is %d \n", result);
   result = !(a != b);
   printf("!(a == b) is %d \n", result);
   result = !(a == b);
   printf("!(a == b) is %d \n", result);

   return 0;
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power. Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators	Operators	Meaning of operators
&	Bitwise AND	~	Bitwise complement
|	Bitwise OR	<<	Shift left
^	Bitwise exclusive OR	>>	Shift right

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example 6: size of Operator

```
#include <stdio.h>
int main()
{
  int a;
  float b;
  double c;
  char d;
  printf("Size of int=%lu bytes\n",sizeof(a));
  printf("Size of float=%lu bytes\n",sizeof(b));
  printf("Size of double=%lu bytes\n",sizeof(c));
  printf("Size of char=%lu byte\n",sizeof(d));

  return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

UNIT-II: WORKING WITH CONTROL STATEMENTS, LOOPS

Conditional statements: Introduction - If statements - If-else statements – nested if-else – break statement-continue statement-go to statement-Switch statements. **Looping statements:** Introduction- While statements – Do-while statements - For Statements-nested loop statements.

Control Statements:

This deals with the various methods that C can control the *flow* of logic in a program. Control statements can be classified as un-conditional and conditional branch statements and loop or iterative statements. The Branch type includes:

Conditional:

- if
- if – else
- Nested if
- switch case statement

Loop or iterative:

- for loop
- while loop
- do-while loop

Un-conditional:

- goto
- break
- return
- continue

Conditional Statements:

Sometimes we want a program to select an action from two or more alternatives. This requires a deviation from the basic sequential order of statement execution. Such programs must contain two or more statements that might *be* executed, but have some way to select only one of the listed options each time the program is run. This is known as conditional execution.

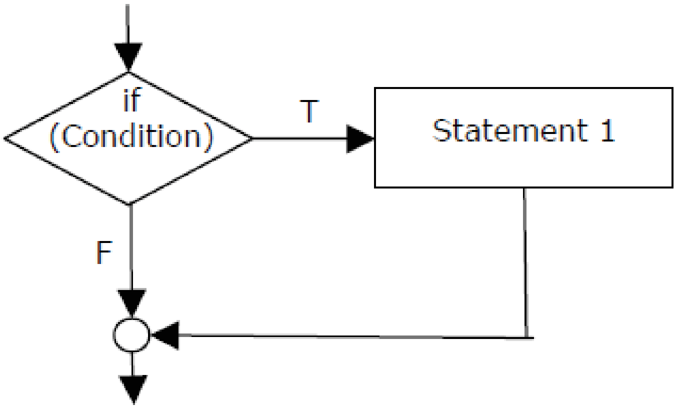
if statement:

Statement or set of statements can be conditionally executed using if statement. Here, logical condition is tested which, may either true or false. If the logical test is true (non zero value) the statement that immediately follows if is executed. If the logical condition is false the control transfers to the next executable statement.

The general syntax of simple **if** statement is:

```
if (condition)
statement_to_execute_if_condition_is_true;
or
if (condition)
{
statement 1;
statement 2;
_ _ _ _;
}
```

Flowchart Segment:



if – else statement:

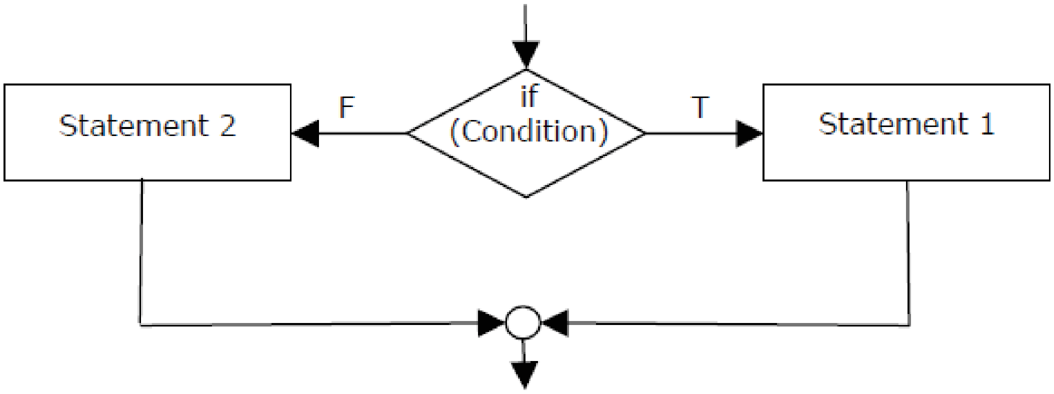
The if statement is used to execute only one action. If there are two statements to be executed alternatively, then if-else statement is used. The if-else statement is a two way branching. The general syntax of simple **if - else** statement is:

if (*condition*)
statement_to_execute_if_condition_is_true;
else
statement_to_execute_if_condition_is_false;

Where, *statement* may be a single statement, a block, or nothing, and the else statement is optional. The conditional statement produces a scalar result, i.e., an integer, character or floating point type.

It is important to remember that an if statement in C can execute only one statement on each branch (T or F). If we desire that multiple statements be executed on a branch, we must **block** them inside of a **{** and **}** pair to make them a single **compound statement**. Thus, the C code for the flowchart segment above would be:

Flowchart Segment:



Example:

```
main()
{
int num;
printf(“ Enter a number : “);
scanf(“%d”,&num);
if (num % 2 == 0)
printf(“ Even Number “);
else
printf(“ Odd Number “);
}
```

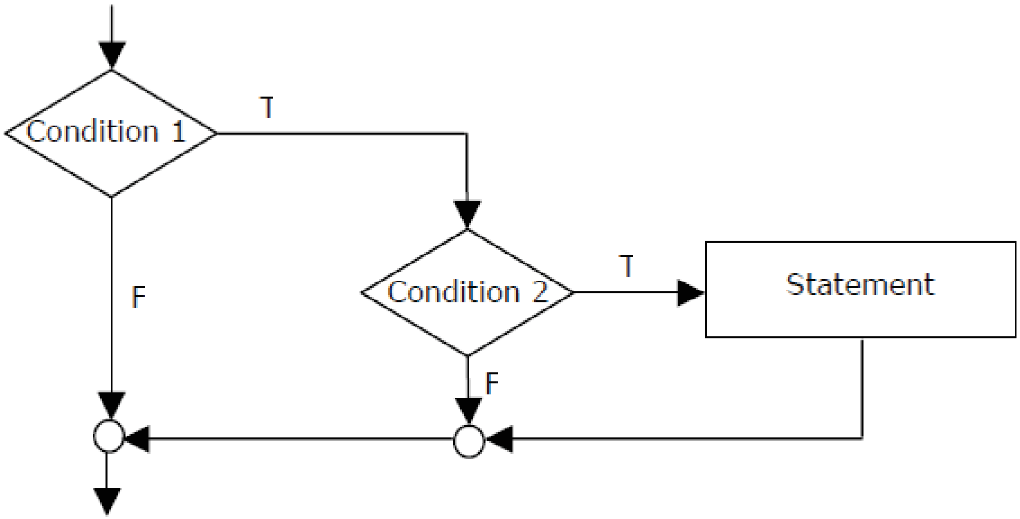
Nested if statement:

The ANSI standard specifies that 15 levels of nesting must be supported. In C, an else statement always refers to the nearest if statement in the same block and not already associated with if.

Example:

```
main()
{
int num;
printf(“ Enter a number : “);
scanf(“%d”,&num);
if( num > 0 )
{
if( num % 2 == 0)
printf(“Even Number“);
else
printf(“Odd Number“);
}
else
{
if( num < 0 )
printf(“Negative Number“);
else
printf(“ Number is Zero“);
}
}
```

Flowchart Segment:



if-else-if Ladder:

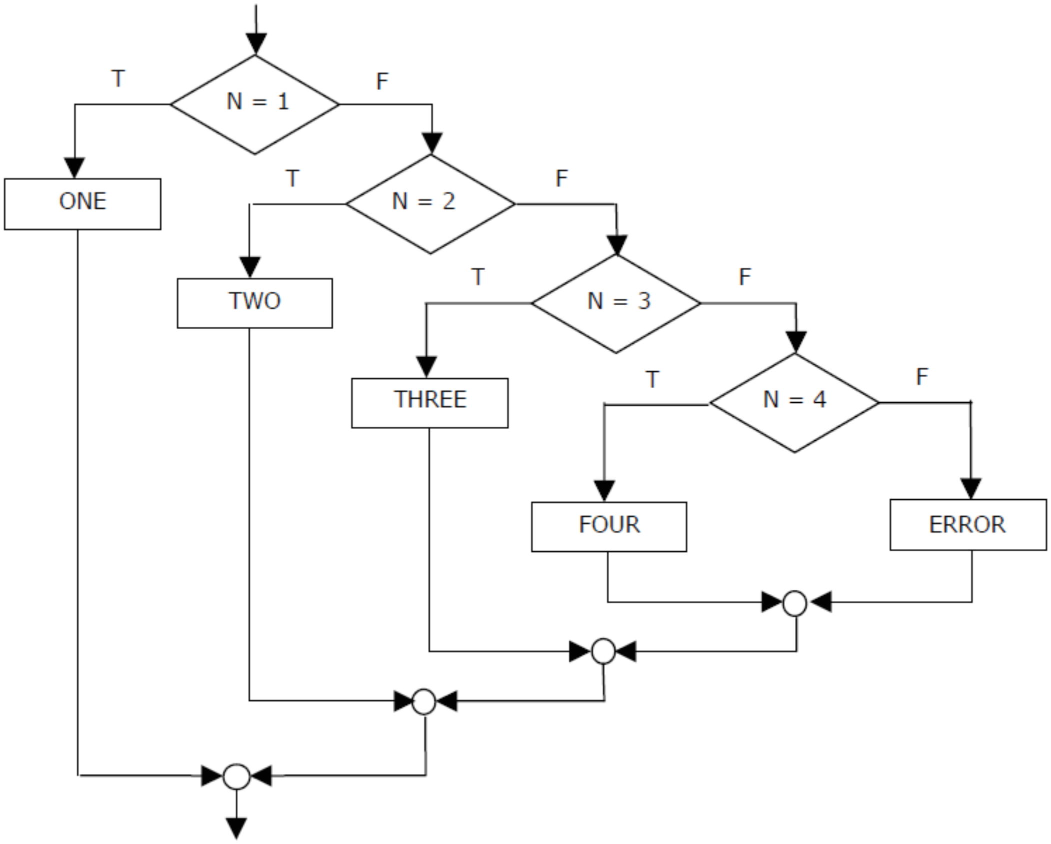
When faced with a situation in which a program must select from *many* processing alternatives based on the value of a single variable, an analyst must expand his or her use of the basic selection structure beyond the standard two processing branches offered by the if statement to allow for multiple branches. One solution to this is to use an approach called **nesting** in which one (or both) branch(es) of a selection contain another selection. This approach is applied to each branch of an algorithm until enough additional branches have been created to handle each alternative. The general syntax of a nested if statement is:

```
if (expression)
statement1
else if (expression)
statement2
..
..
else
statement3
```

Example:

```
#include <stdio.h>
void main (void)
{
int N; /* Menu Choice */
printf ("MENU OF TERMS\n\n");
printf ("1. Single\n");
printf ("2. Double\n");
printf ("3. Triple\n");
printf ("4. Quadruple\n\n");
printf ("Enter the numbe (1-4): ");
scanf ("%d", &N);
if (N == 1) printf ("one");
else if (N == 2) printf ("two");
else if (N == 3) printf ("three");
else if (N == 4) printf ("four");
else printf ("ERROR");
}
```

Flowchart Segment:



The ? : operator (ternary):

The ? (*ternary condition*) operator is a more efficient form for expressing simple if statements. It has the following form:

expression1 ? *expression2* : *expression3*

It simply states as:

if *expression1* **then** *expression2* **else** *expression3*

Example:

Assign the maximum of a and b to z:

```
main()
{
int a,b,z;
printf("\n Enter a and b ");
scanf("%d%d",&a,&b);
z = (a > b) ? a : b;
printf("Maximum number: %d", z);
}
which is the same as:
if (a > b)
z = a;
else
z=b;
```

The switch case statement:

The switch-case statement is used when an expression's value is to be checked against several values. If a match takes place, the appropriate action is taken. The general form of switch case statement is:

```
switch (expression)
{
case constant1:
statement;
break;
case constant2:
statement;
break;
default:
statement;
break;
}
```

In this construct, the expression whose value is being compared may be any valid expression, including the value of a variable, an arithmetic expression, a logical comparison rarely, a bit wise expression, or the return value from a function call, but not a floating-point expression. The expression's value is checked against each of the specified cases and when a match occurs, the statements following that case are executed. When a break statement is encountered, control proceeds to the end of the switch - case statement.

The break statements inside the switch statement are optional. If the break statement is omitted, execution will continue on into the next case statements even though a match has already taken place until either a break or the end of the switch is reached.

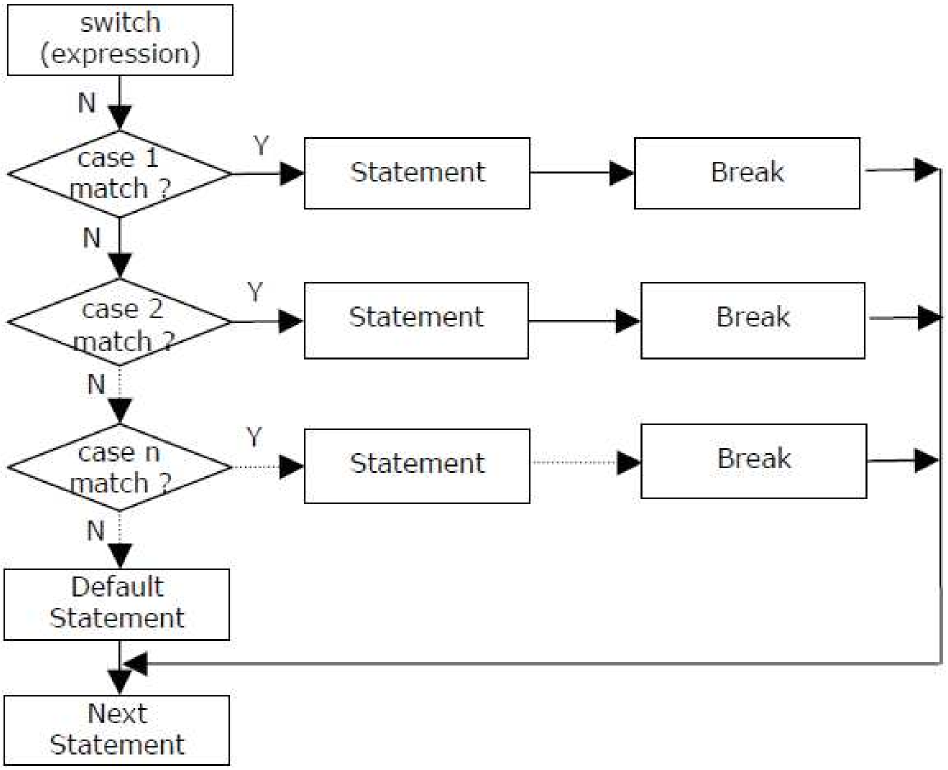
The keyword case may only be constants, they cannot be expressions. They may be integers or characters, but not floating point numbers or character string. Case constants may not be repeated within a switch statement. The last case is a special keyword default. The default statement is executed if no matches are found. The default is optional and if it is not present, no action takes place if all matches fail.

Three important things to know about switch statement:

1. The switch differs from the if in that switch can only test for equality whereas if can evaluate any type of relational or logical expression.
2. No two case constants in the same switch can have identical values. But, a switch statement enclosed by an outer switch may have case constants and either same.
3. If character constants are used in the switch statement, they are automatically converted to integers.

Flowchart Segment - Case Selection:

In the example below, five possible paths might be followed depending on the value stored in the character storage location X. Each path is selected based on the individual value(s) that might be stored in X.



Example 1:

```
main()
{
char gender;
printf (“Enter Gender code:(M/F)“);
scanf (“%c”, &gender);
switch (gender)
{
case ‘M’ : printf (“ Male“);
break;
case ‘F’ : prrintf (“Female”);
break;
default : printf (“Wrong code“);
}
}
```

We can also have null statements by just including a “**;**” or let the switch statement *fall through* by omitting any statements (see *example* below).

Example 2:

```
switch (letter)
{
case `A':
case `E':
case `I' :
case `O':
case `U':
numberofvowels++;
break;
case ‘ ':
numberofspaces++;
break;
default:
numberofconstants++;
break;
}
```

In the above example if the value of letter is `A', `E', `I', `O' or `U' then numberofvowels is incremented. If the value of letter is ` ' then numberofspaces is incremented. If none of these is true then the default condition is executed, that is numberofconstants is incremented.

Un-conditional (Jump) statements:

C has four jump statements to perform an unconditional branch:

- return
- goto
- break and
- continue

return statement:

A return statement is used to return from a function. A function can use this statement as a mechanism to return a value to its calling function. If now value is specified, assume that a garbage value is returned (some compilers will return 0).

The general form of return statement is:

```
return expression;
```

Where expression is any valid rvalue expression.

Example:

```
return x; or return(x);
return x + y or return(x + y);
return rand(x); or return(rand(x));
return 10 * rand(x); or return (10 * rand(x));
```

We can use as many return statements as we like within a function. However, the function will stop executing as soon as it encounters the first return. The } that ends a function also causes the function to return. It is same way as return without any specified value.

A function declared as void may not contain a return statement that specifies a value.

goto statement:

goto statement provides a method of unconditional transfer control to a labeled point in the program. The goto statement requires a destination label declared as:

```
label:
```

The label is a word (permissible length is machine dependent) followed by a colon. The goto statement is formally defined as:

```
goto label;
‘
‘
‘
label:
target statement
```

Since, C has a rich set of control statements and allows additional control using break and continue, there is a little need for goto. The chief concern about the goto is its tendency to render programs unreachable. Rather, it a convenience, it used wisely, can be a benefit in a narrow set of programming situation. So the usage of goto is highly discouraged.

Example:

```
Void main()
{
int x = 6, y = 12;
if( x == y)
x++;
else
goto error;
error:
printf (“Fatal error; Exiting”);
}
```

The compiler doesn’t require any formal declaration of the label identifiers.

break statement:

We can use it to terminate a case in a switch statement and to terminate a loop. Consider the following example where we read an integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is greater than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

Example:

```
void main()
{
int value;
while (scanf(“%d'', &value ) == 1 && value != 0)
{
if(value < 0)
{
printf (“Illegal value\n'');
break; /*Terminatetheloop*/
}
if(value > 100)
{
printf(“Invalid value\n'');
continue; /* Skip to start loop again */
}
} /* end while value != 0 */
}
```

Continue statement:

The continue statement forces the next iteration of the loop to take place, skipping any code in between. But the break statement forces for termination.

Example 1:

/* Program to print the even numbers below 100 */

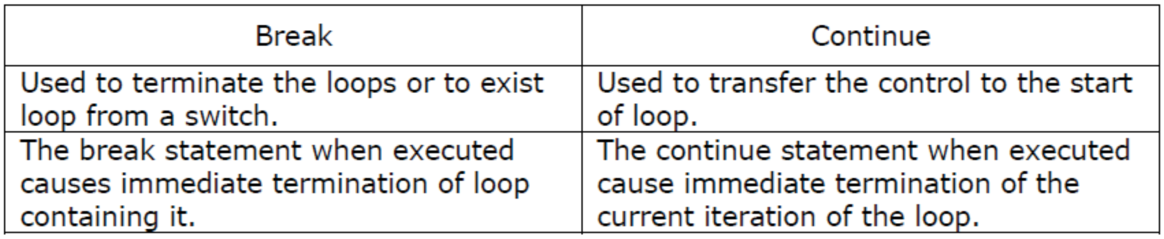
```
#include<stdio.h>
void main()
{
int x;
for(x = 1; x < 10; x++)
{
if(x % 2)
continue;
printf (“%d\t”, x)
}
}
```

An odd number causes continue to execute and the next iteration to occur, by passing the printf () statement. A continue statement is used within a loop (i.e for, while, do – while) to end an iteration in while and do-while loops, a continue statement will cause control to go directly to the conditional test and then continue the looping process. In the case of for, first the increment part of the loop is performed, next the conditional test is executed and finally the loop continues.

Example 2:

```
main()
{
char ch;
while (1)
{
ch = getchar();
if(ch==EOF)
break;
if (iscntrl (ch))
continue;
else
printf ("\n not a control character");
}
```

Distinguishing between break and continue statement:



Break	Continue
Used to terminate the loops or to exist loop from a switch.	Used to transfer the control to the start of loop.
The break statement when executed causes immediate termination of loop containing it.	The continue statement when executed cause immediate termination of the current iteration of the loop.

The exit () function:

Just as we can break out of a loop, we can break out of a program by using the standard library function exit(). This function causes immediate termination of the entire program, forcing a return to the operation system.

The general form of the exit() function is:

```
void exit (int return_code);
```

The value of the return_code is returned to the calling process, which is usually the operation system. Zero is generally used as a return code to indicate normal program termination.

Example:

```
Void menu(void)
{
charch;
printf(“B: Breakfast“);
printf(“L: Lunch“);
printf(“D: Dinner”);
printf(“E: Exit”);
printf(“Enter your choice: “);
do
{
ch = getchar();
switch (ch)
{
case‘B’:
printf (“time for breakfast”);
break;
case‘L’:
printf(“timeforlunch”);
break;
case‘D’:
printf(“timefordinner”);
break;
case‘E’:
exit (0); /* return to operating system */
}
} while (ch != ‘B’ && ch != ‘L’ && ch != ‘D’);
}
```

LOOPING AND ITERATION:

Looping is a powerful programming technique through which a group of statements is executed repeatedly, until certain specified condition is satisfied. Looping is also called a repetition or iterative control mechanism.

C provides three types of loop control structures. They are:

- for statement
- while statement
- do-while statement

The for statement:

The for loop statement is useful to repeat a statement/s a known number of times. The general syntax is as follows:

for (initialization; condition; operation)
statement;

The **initialization** is generally an assignment statement that is used to set the loop control variable.

The **condition** is an expression(relational/logical/arithmetic/bitwise ....) that determines when the loop exists.

The **Operation** defines how the loop control variable changes each time the loop is repeated.

We must separate these three major sections by semicolon.

The for loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the for. The control flow of the for statement is as follows:

Example 1:

```
// printing all odd and even numbers between 1 to 5
int x;
main ()
{
for (x=1; x <=5 ; x++)
{
if( x % 2 == 0 )
printf( “ %d is EVEN \n”,x);
else
printf(“ %d isODD \n”,x);
}
}
```

Output to the screen:

```
1 is ODD
2 is EVEN
3 is ODD
4 is EVEN
5 is EVEN
```

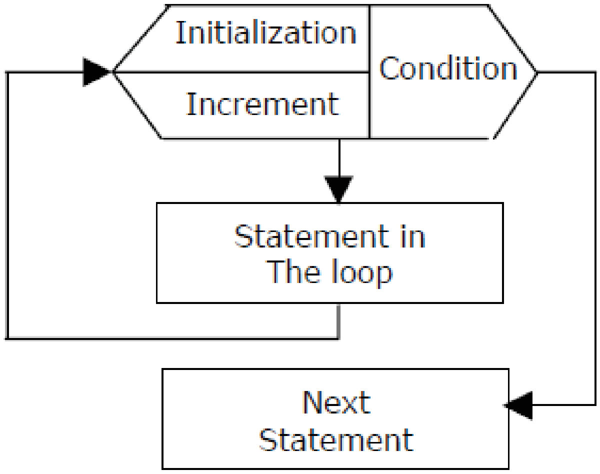
Example 2:

```
// sum the squares of all the numbers between 1 to 5
main()
{
int x, sum = 0;
for (x = 1; x <= 5; x ++)
{
sum = sum + x * x;
}
printf ("\n Sum of squares of all the numbers between 1 to 5 = %d ", sum);
}
```

Output to the screen:

```
Sum of squares of all the numbers between 1 to 5 = 55
```

Flowchart Segment - for Statement:



The **comma (,) operator** is used to extend the flexibility of the for loop. It allows the general form to be modified as follows:

```
for (initialization_1, initialization_2; condition; operation_1, operation_2)
statement;
```

All the following are legal for statements in C. The practical application of such statements is not important here, we are just trying to illustrate peculiar features that may be useful:

1. for (x=0; ((x>3) && (x<9)); x++)
2. for (x=0,y=4; ((x>3) && (y<9)); x++, y+=2)
3. for (x=0, y=4, z=4000; z; z/=10)

The second example shows that multiple expressions can be separated by a , (comma).

Example:

```
main()
{
int j ;
double degC, degF;
clrscr ();
printf (“\n Table of Celsius and Fahrenheit degrees \n\n”);
printf (“Celsius Degree \t Fahrenheit Degree \n“)
degC = -20.0;
for (j = 1; j <= 6; j++)
{
degC = degC+ 20.0;
degF = (degC * 9.0/5.0) + 32.0;
printf (“\n %7.2lf\t\ %7.2lf “, degC, degF);
}
}
```

Output:

```
Table of Celsius and Fahrenheit degrees
Celsius Degree                    Fahrenheit Degree
0.00                                  32.00
20.00                                 68.00
40.00                                 104.00
60.00                                 140.00
80.00                                 176.00
100.00                                212.00
```

2.4.2. Nested for loop:

Nested loops consist of one loop placed inside another loop. An example of a nested for loop is:

```
    for (initialization; condition; operation)
    {
    for (initialization; condition; operation)
    {
    statement;
    }
    statement;
    }
```

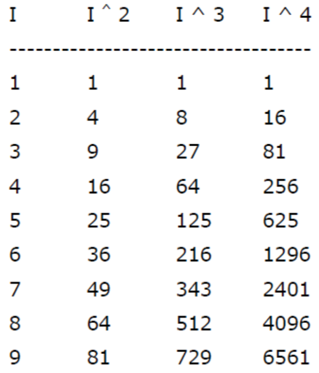
In this example, the inner loop runs through its full range of iterations for each single iteration of the outer loop.

Example:

Program to show table of first four powers of numbers 1 to 9.

```
#include <stdio.h >
void main()
{
int i, j, k, temp;
printf("I\tI^2\tI^3\tI^4 \n");
printf("--------------------------------\n");
for ( i = 1; i < 10; i ++) /* Outer loop */
{
for (j = 1; j < 5; j ++) /* 1st level of nesting */
{
temp=1;
for(k = 0; k < j; k ++)
temp=temp*I;
printf ("%d\t", temp);
}
printf ("\n");
}
}
```

Output to the screen:



```
I       I ^ 2    I ^ 3    I ^ 4
----------------------------------
1       1        1        1
2       4        8        16
3       9        27       81
4       16       64       256
5       25       125      625
6       36       216      1296
7       49       343      2401
8       64       512      4096
9       81       729      6561
```

Infinite for loop:

We can make an endless loop by leaving the conditional expression empty as given below:

```
for( ; ; )
printf("This loop will run for ever");
```

To terminate the infinite loop the break statement can be used anywhere inside the body of the loop. A sample example is given below:

```
for(; ;)
{
ch = getchar ();
if(ch=='A')
break;
}
printf("You typed an A");
```

This loop will run until the user types an A at the keyboard.

for with no bodies:

A C-statement may be empty. This means that the body of the for loop may also be empty. There need not be an expression present for any of the sections. The expressions are optional.

Example 1:

```
/* The loop will run until the user enters 123 */
for( x = 0; x != 123; )
scanf("%d", &x);
```

This means that each time the loop repeats, 'x' is tested to see if it equals 123, but no further action takes place. If you type 123, at the keyboard, however the loop condition becomes false and the loop terminates.

The initialization sometimes happens when the initial condition of the loop control variable must be computed by some complex means.

Example 2:

```
/* Program to print the name in reverse order. */
    #include<conio.h>
    #include<string.h>
    #include<stdio.h>
    void main()
    {
    char s[20];
    int x;
    clrscr ();
    printf ("\nEnter your name: ");
gets (s);
x = strlen (s);
for ( ; x > 0 ; )
{
--x;
printf ("%c\t", s[x]);
}
}
```

Output to the screen:

```
Enter your name: KIRAN
N A R I K
```

The while statement:

The second loop available in ‘C’ is while loop.

The general format of while loop is:

```
while (expression)
statement
```

A while statement is useful to repeat a statement execution as long as a condition remains true or an error is detected. The while statement tests the condition before executing the statement.

The condition, can be any valid C languages expression including the value of a variable, a unary or binary expression, an arithmetic expression, or the return value from a function call.

The statement can be a simple or compound statement. A compound statement in a while statement appears as:

```
while (condition)
{
statement1;
statement2;
}
```

With the if statement, it is important that no semicolon follow the closing parenthesis, otherwise the compiler will assume the loop body consists of a single null statement.

This usually results in an infinite loop because the value of the condition will not change within the body of the loop.

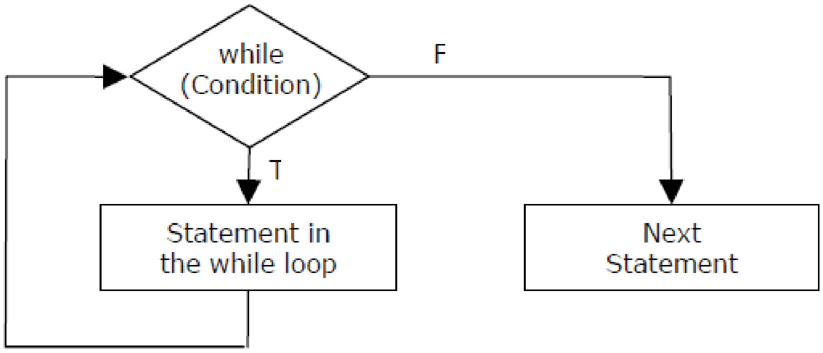
Example:

```
main()
{
int j = 1;
double degC, degF;
clrscr ();
printf (“\n Table of Celsius and Fahrenheit degrees \n\n”);
printf (“Celsius Degree \t Fahrenheit Degree \n“)
degC = -20.0;
while (j <= 6)
{
degC = degC+ 20.0;
degF = (degC * 9.0/5.0) + 32.0;
printf (“\n %7.2lf\t\ %7.2lf “, degC, degF);
j++;
}
}
```

Output:

```
Table of Celsius and Fahrenheit degrees
Celsius Degree                  Fahrenheit Degree
0.00                                32.00
20.00                               68.00
40.00                               104.00
60.00                               140.00
80.00                               176.00
100.00                              212.00
```

Flowchart Segment - while Statement:



Because the while loop can accept expressions, not just conditions, the following are all legal:

```
while(x--);
while(x = x+1);
while(x += 5);
```

Using this type of expression, only when the result of x--, x=x+1, or x+=5, evaluates to 0 will the while condition fail and the loop be exited.

We can go further still and perform complete operations within the while *expression*:

```
while(i++ < 10);
```

The counts i up to 10.

```
while((ch = getchar()) !=`q')
putchar(ch);
```

This uses C standard library functions: getchar () to reads a character from the keyboard and putchar () to writes a given char to screen. The while loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read.

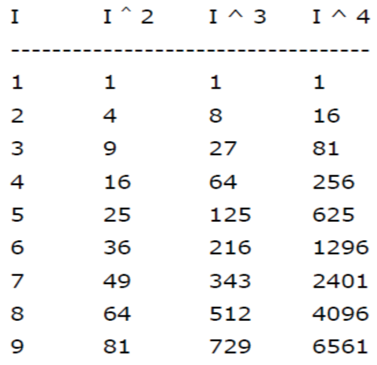
Nested while:

Example:

Program to show table of first four powers of numbers 1 to 9.

```
#include <stdio.h >
void main()
{
int i, j, k, temp;
printf("I\tI^2\tI^3\tI^4 \n");
printf("--------------------------------\n");
i = 1;
while (i < 10) /* Outer loop */
{
j=1;
while (j < 5) /* 1st level of nesting */
{
temp=1;
k=1;
while (k < j)
{
temp = temp * i;
k++;
}
printf ("%d\t", temp);
j++;
}
printf ("\n");
i++;
}
}
```

Output to the screen:



```
I       I ^ 2   I ^ 3   I ^ 4
-----------------------------------
1       1       1       1
2       4       8       16
3       9       27      81
4       16      64      256
5       25      125     625
6       36      216     1296
7       49      343     2401
8       64      512     4096
9       81      729     6561
```

The do-while statement:

The third loop available in C is do – while loop.

The general format of do-while is:

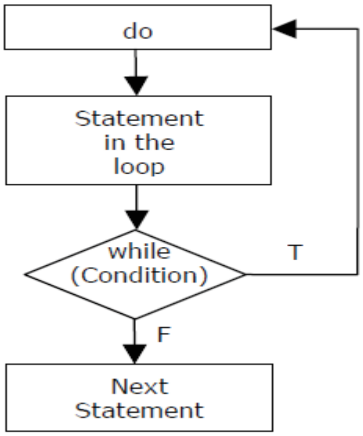
```
        do
        statement;
        while (expression);
```

Unlike for and while loops, which tests the condition at the top of the loop. The do – while loop checks its condition at the bottom of the loop. This means that the do – while loop always executes first and then the condition is tested. Unlike the while construction, the do – while requires a semicolon to follow the statement's conditional part.

If more than one statement is to be executed in the body of the loop, then these statements may be formed into a compound statement as follows:

```
do
{
statement1;
statement2;
} while (condition);
```

Flowchart Segment of do-while Statement:



Example 1:

```
        # include <stdio.h>
        main()
        {
        do
        {
        printf("x = %d\n", x--);
        } while(x > 0);
        }
```

Output to the screen:

```
        X = 3
```

```
X = 2
X = 1
```

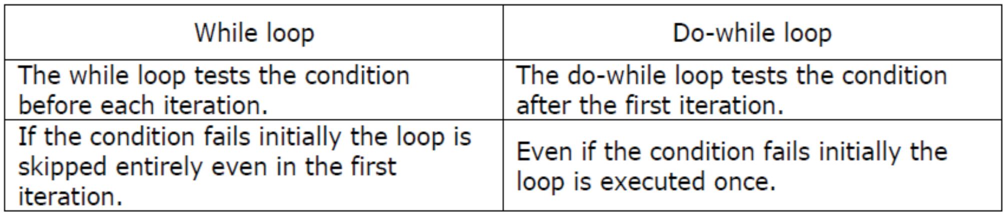
Example 2:

```
#include <stdio.h>
void main()
{
char ch;
printf("T: Train\n");
printf("C: Car\n");
printf("S: Ship\n");
do
{
printf("\nEnter your choice: ");
fflush(stdin);
ch = getchar();
switch(ch)
{
case 'T':
printf("\nTrain");
break;
case 'C':
printf("\nCar");
break;
case 'S':
printf("\nShip");
break;
default:
printf("\n Invalid Choice");
}
} while(ch == 'T' || ch == 'C' || ch == 'S');
}
```

Output to the screen:

```
T: Train
C: Car
S: Ship
Enter your choice: T
Train
```

Distinguishing between while and do-while loops:



While loop	Do-while loop
The while loop tests the condition before each iteration.	The do-while loop tests the condition after the first iteration.
If the condition fails initially the loop is skipped entirely even in the first iteration.	Even if the condition fails initially the loop is executed once.

UNIT-III: FUNCTIONS, ARRAYS AND STRINGS

Functions: Definition and declaration of functions- Function proto type-return statement-types of functions and Built in functions. **User defined functions:** Introduction - Need for user defined functions – Components of functions. **Arrays:** Introduction - Defining an array - Initializing an array - One dimensional array – Multi dimensional array. **Strings:** Introduction – Declaring, initializing string and Handling strings – String handling functions. **Pointers**: Features of Pointers – Declaration of Pointers – Advantages of Pointers.

Functions:

Definition: A function is a sub-Program or) self contained program that is defined for performing a specific task, it is a reusable block of code that gets executed on calling it. It can be treated as sub program. The concept of reusability is achieved using functions; every function in "c" language should return a value.

In other words, 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.

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.

Function prototype

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

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

Syntax of function prototype

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

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

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

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

The Return Statement

- The **return** statement exits the called function and returns control back to the calling function.
  - Once a return statement is executed, no further instructions within the function are executed.
- A single return value (of the appropriate type) may be returned.
  - Parentheses are allowed but not required around the return value.
  - A function with a void return type will not have a return value after the return statement.

- More than one return statement may appear in a function, but only one will ever be executed by any given function call.
    - (All returns other than the last need to be controlled by logic such as "if" blocks.)
- If a function does not contain a return statement, most compilers will add one automatically at the end of the routine, and may generate a warning message. The return value, if any, is undefined in this case.
- "main( )" is technically a function, and should return 0 upon successful completion, or a non-zero value otherwise. This is ignored by many programmers, but some compilers will issue warning messages if main() does not contain a return statement.

Based on function definition, Functions are classified into two types. They are:

1. Built-in/Library Functions/Predefined Functions
2. User defined Functions

Built-in/Library Functions/Pre defined Functions: These are the functions which are already predefined to the c-compiler by the developers of 'C'.

Library functions are the built in function that are already defined in the C library. The prototype of these functions are written in header files. So we need to include respective header files before using a library function. For example, the prototype of math functions like *pow(), sqrt()*, etc is present in math.h, the prototype of *exit(), malloc(), calloc()* etc is in stdlib.h and so on.

User defined Functions: The functions which are written by the user as per his requirement are called as user defined functions.

hose functions that are defined by user to use them when required are called user-defined function. Function definition is written by user. *main()* is an example of user-defined function.

Components of Functions:

Programs using functions will contain the 3 major components.

1. Function Prototype Declaration
2. Function Definition
3. Function call

Function Prototype Declaration:

Before the function is defined in the program the function name and its details should be provided to the compiler and it can be done by declaring the function above the main function and such statement is known as function prototype.

Syntax: return_type function_name(arguments/parameters list);

The prototype of function consists of 3 parts:

1. Function name
2. Argument list
3. Return type of the function

The function name specifies the name of the function and tells the compiler that, it can use the function with this name.

The arguments in the function specify the type and no. of arguments that are to be used within the function. The arguments are also known as parameters.

The return type specifies the type of value to be returned by the function.

The function prototype declaration should always terminate with semicolon.

Ex: void display();

Here display is the function name and it does not contain any arguments and it does not return anything that is the reason we written void as the return type.

int add(int,int);

Here add is the function name and it contains two arguments of integer type and the return type is int.

Function Definition:

It is the actual function that contains the programming statements to perform a specific task. The programming statements should be written in between the {   } braces. The function definition is also called as the body of the function.

Syntax:

```
return_type    function_name(argument list)
{
      ----
      ----  statement(s);
      ----
}
```

Function call:

Defining a function does not do anything. A function performs its tasks (or) operations when it is executed. To execute the function it should be called. A function can be called by the name of the function and such statement is known as function call.

While calling the function the required arguments should be passed(if necessary) and if function returns any value then that value should be stored in some variable for further accessing of it.

Syntax: function_name(argument list);

Ex: display();
add(10,20);

A function can be called by two ways. They are:

- Call by value
- Call by reference

Call by value

When a function is called by value, a copy of actual argument is passed to the called function. The copied arguments occupy separate memory location than the actual argument. If any changes done to those values inside the function, it is only visible inside the function. Their values remain unchanged outside it.

Call by reference

In this method of passing parameter, the address of argument is copied instead of value. Inside the function, the address of argument is used to access the actual argument. If any changes is done to those values inside the function, it is visible both inside and outside the function.

User defined Functions are divided into four types:

1. Functions with no arguments and no return value
2. Functions with argument and no return value
3. Functions with no arguments and return value
4. Functions with argument and return value

Function with no argument and no return type:

In this type of functions the functions will not carry any arguments to the called function and called function does not return any values to the function call.

Functions with argument and no return value:

In this type of functions the functions will carry arguments to the called function and called function does not return any values to the function call.

Functions with no arguments and return value:

In this type of functions the functions will not carry any arguments to the called function and called function will return any value to the function call.

Functions with argument and return value:

In this type of functions the functions will carry arguments to the called function and called function will return any value to the function call.

Arrays: Introduction - Defining an array - Initializing an array - One dimensional array – Multi dimensional array.

Introduction:

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

type arrayName [arraySize];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement –

double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

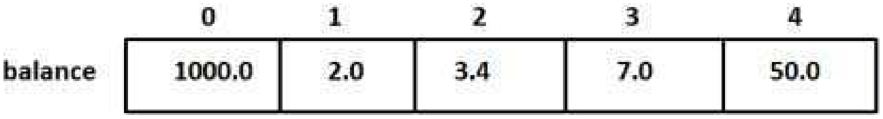
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –



	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable.

One Dimensional Array:

Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another.

an array of 6 elements

Syntax: datatype array_name[size];

datatype: It denotes the type of the elements in the array.

array_name: Name of the array. It must be a valid identifier.

size: Number of elements an array can hold.

here are some example of array declarations:

1 int num[100];
2 float temp[20];
3 char ch[50];

num is an array of type int, which can only store 100 elements of type int.

temp is an array of type float, which can only store 20 elements of type float.

ch is an array of type char, which can only store 50 elements of type char.

Note: When an array is declared it contains garbage values.

The individual elements in the array:

1 num[0], num[1], num[2], ..., num[99]
2 temp[0], temp[1], temp[2], ..., temp[19]
3 ch[0], ch[1], ch[2], ..., ch[49]

We can also use variables and symbolic constants to specify the size of the array.

```
#define SIZE 10

int main()
{
   int size = 10;

   int my_arr1[SIZE]; // ok
   int my_arr2[size]; // not allowed until C99
   // ...
}
```

C programming language allows multidimensional arrays. Here is the general form of a **multidimensional array declaration –**

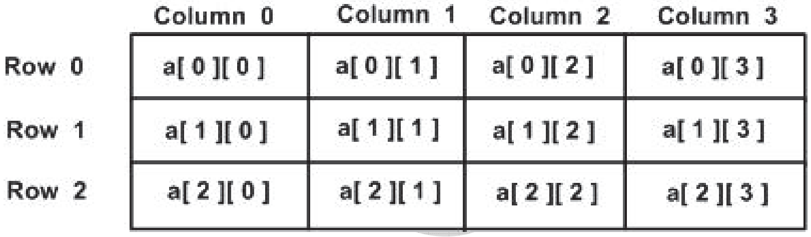
type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array –
int threedim[5][10][4];

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows –
type arrayName [x][y];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows –



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

int val = a[2][3];

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure.

Strings: Introduction – Declaring, initializing string and Handling strings – String handling functions.

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

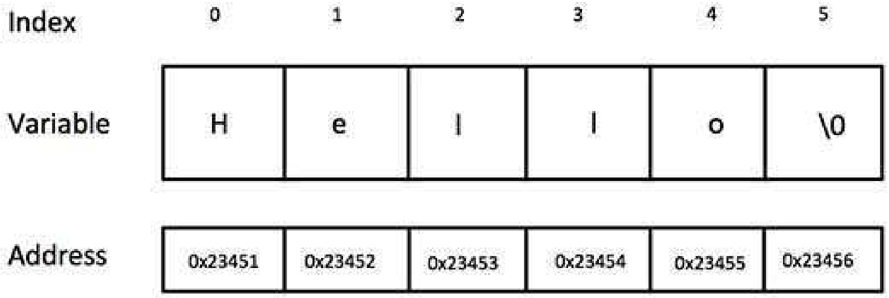
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

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

char greeting[] = "Hello";

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



Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

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

```
#include <stdio.h>

int main () {

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

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

```
Greeting message: Hello
```

Declare and initialize a String

A string is a simple array with char as a data type. 'C' language does not directly support string as a data type. Hence, to display a string in 'C', you need to make use of a character array.

The general syntax for declaring a variable as a string is as follows,

```
char string_variable_name [array_size];
```

The classic string declaration can be done as follow:

```
 char string_name[string_length] = "string";
```

The size of an array must be defined while declaring a string variable because it used to calculate how many characters are going to be stored inside the string variable. Some valid examples of string declaration are as follows,

```
char first_name[15];    //declaration of a string variable
char last_name[15];
```

The above example represents string variables with an array size of 15. This means that the given character array is capable of holding 15 characters at most. The indexing of array begins from 0 hence it will store characters from a 0-14 position. **The C compiler automatically adds a NULL character '\0' to the character array created.**

Let's study the initialization of a string variable. Following example demonstrates the initialization of a string variable,

```
char first_name[15] = "ANTHONY";
char first_name[15] = {'A','N','T','H','O','N','Y','\0'}; // NULL character '\0' is required at end in this declaration
char string1 [6] = "hello";/* string size = 'h'+'e'+'l'+'l'+'o'+"NULL" = 6 */
char string2 [ ] = "world";  /* string size = 'w'+'o'+'r'+'l'+'d'+"NULL" = 6 */
char string3[6] = {'h', 'e', 'l', 'l', 'o', '\0'} ; /*Declaration as set of characters ,Size 6*/
```

In string3, the NULL character must be added explicitly, and the characters are enclosed in single quotation marks.

'C' also allows us to initialize a string variable without defining the size of the character array. It can be done in the following way,

```
char first_name[ ] = "NATHAN";
```

The name of a string acts as a pointer because it is basically an array.

STRING HANDLING FUNCTIONS:

- **strlen()** - calculates the length of a string

The strlen() function calculates the length of a given string.

The strlen() function takes a string as an argument and returns its length. The returned value is of type long int.

It is defined in the <string.h> header file.

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
   char a[20]="Program";
   char b[20]={'P','r','o','g','r','a','m','\0'};

   printf("Length of string a = %ld \n",strlen(a));
   printf("Length of string b = %ld \n",strlen(b));

   return 0;
}
```

Output

```
Length of string a = 7
Length of string b = 7
```

Note that the strlen() function doesn't count the null character \0 while calculating the length.

- **strcpy()** - copies a string to another

The strcpy() function copies the string to the another character array.

strcpy() Function prototype

```
char* strcpy(char* destination, const char* source);
```

The strcpy() function copies the string pointed by source (including the null character) to the character array destination.

The function also returns the copied array.

The strcpy() function is defined in the string.h header file.

Example: C strcpy()

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

int main()
{
   char str1[10]= "awesome";
   char str2[10];
   char str3[10];

   strcpy(str2, str1);
   strcpy(str3, "well");
   puts(str2);
   puts(str3);

   return 0;
}
```

Output

```
awesome
well
```

It is important to note that the destination array should be large enough to copy the array. Otherwise, it may result in undefined behavior.

- **strcmp()** - compares two strings

The strcmp() function compares two strings and returns 0 if both strings are identical.

C strcmp() Prototype

```
int strcmp (const char* str1, const char* str2);
```

The strcmp() function takes two strings and returns an integer.

The strcmp() compares two strings character by character.

If the first character of two strings is equal, the next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in the string.h header file.

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
negative	if the ASCII value of the first unmatched character is less than second.
positive integer	if the ASCII value of the first unmatched character is greater than second.

Example: C strcmp() function

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

int main()
{
   char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
   int result;
   // comparing strings str1 and str2
   result = strcmp(str1, str2);
   printf("strcmp(str1, str2) = %d\n", result);
   // comparing strings str1 and str3
   result = strcmp(str1, str3);
   printf("strcmp(str1, str3) = %d\n", result);

   return 0;
}
```

Output

```
strcmp(str1, str2) = 32
strcmp(str1, str3) = 0
```

The first unmatched character between string str1 and str2 is third character. The ASCII value of 'c' is 99 and the ASCII value of 'C' is 67. Hence, when strings str1 and str2 are compared, the return value is 32.

When strings str1 and str3 are compared, the result is 0 because both strings are identical.

- **strcat()** - concatenates two strings

  The function strcat() concatenates two strings.

  In C programming, strcat() concatenates (joins) two strings.

  The strcat() function is defined in <string.h> header file.

C strcat() Prototype

```
char *strcat(char *dest, const char *src)
```

It takes two arguments, i.e, two strings or character arrays, and stores the resultant concatenated string in the first string specified in the argument.

The pointer to the resultant string is passed as a return value.

Example: C strcat() function

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[] = "This is ", str2[] = "programiz.com";

   //concatenates str1 and str2 and resultant string is stored in str1.
   strcat(str1,str2);

   puts(str1);
   puts(str2);

   return 0;
}
```

Output

```
This is programiz.com
programiz.com
```

Pointers: Features of Pointers – Declaration of Pointers – Advantages of Pointers.

Introduction:

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.

Features of Pointers:

1. Pointers save memory space.
2. Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location.
3. Memory is accessed efficiently with the pointers. The pointer assigns and releases the memory as well. Hence it can be said the Memory of pointers is dynamically allocated.
4. Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
5. An array, of any type can be accessed with the help of pointers, without considering its subscript range.
6. Pointers are used for file handling.
7. Pointers are used to allocate memory dynamically.
8. In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Uses of pointers:

1. To pass arguments by reference
2. For accessing array elements
3. To return multiple values
4. Dynamic memory allocation
5. To implement data structures
6. To do system level programming where memory addresses are useful

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. **int** n = 10;

1. **int*** p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

2. **int** *a;//pointer to int

3. **char** *c;//pointer to char

Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
2) We can **return multiple values from a function** using the pointer.
3) It makes you able to **access any memory location** in the computer's memory.

UNIT-IV:

Structures: Features of Structures - Declaring and initialization of Structures –Structure within Structure-Array of Structures- Enumerated data type-**Unions**-Definition and advantages of Unions comparison between Structure & Unions.
Object Oriented Programming: Introduction to Object Oriented Programming – Structure of C++ - Simple Program of C++ - Differences between C & C++

C Structures

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.
For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.
In structure, data is stored in form of **records**.

Defining a structure

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

Syntax:

```
struct [structure_tag]
{
   //member variable 1
   //member variable 2
   //member variable 3
   ...
}[structure_variables];
```

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.
Note: The closing curly brace in the structure type declaration must be followed by a semicolon(;).

Example of Structure

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   // F for female and M for male
   char gender;
};
```

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members.**

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot . operator also called **period** or **member access** operator.

For example:

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

struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

int main()
{
   struct Student s1;

   /*
      s1 is a variable of Student type and
      age is a member of Student
   */
   s1.age = 18;
   /*
      using string function to add name
   */
   strcpy(s1.name, "Viraaj");
   /*
      displaying the stored values
   */
   printf("Name of Student 1: %s\n", s1.name);
   printf("Age of Student 1: %d\n", s1.age);

   return 0;
}
```

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
   float height;
   int weight;
   int age;
};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
or,
struct Patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Array of Structure

We can also declare an array of **structure** variables. in which each element of the array will represent a **structure** variable. **Example :** struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#include<stdio.h>

struct Employee
{
  char ename[10];
  int sal;
};

struct Employee emp[5];
int i, j;
void ask()
{
  for(i = 0; i < 3; i++)
  {
    printf("\nEnter %dst Employee record:\n", i+1);
    printf("\nEmployee name:\t");
    scanf("%s", emp[i].ename);
    printf("\nEnter Salary:\t");
    scanf("%d", &emp[i].sal);
  }
  printf("\nDisplaying Employee record:\n");
  for(i = 0; i < 3; i++)
  {
    printf("\nEmployee name is %s", emp[i].ename);
    printf("\nSlary is %d", emp[i].sal);
  }
}
void main()
{
  ask();
}
```

Nested Structures

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another stucture as member variable.

Example:

```
struct Student
{
  char[30] name;
  int age;
```

```
   /* here Address is a structure */
   struct Address
   {
      char[50] locality;
      char[50] city;
      int pincode;
   }addr;
};
```

Structure as Function Arguments

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example:

```
#include<stdio.h>

struct Student
{
   char name[10];
   int roll;
};

void show(struct Student st);

void main()
{
   struct Student std;
   printf("\nEnter Student record:\n");
   printf("\nStudent name:\t");
   scanf("%s", std.name);
   printf("\nEnter Student rollno.:\t");
   scanf("%d", &std.roll);
   show(std);
}

void show(struct Student st)
{
   printf("\nstudent name is %s", st.name);
   printf("\nroll is %d", st.roll);
}
```

C Unions:

A union is a user-defined type similar to structs in C programming.

How to define a union?

We use the union keyword to define unions. Here's an example:

```
union car
{
 char name[50];
 int price;
};
```

The above code defines a derived type union car.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
 char name[50];
 int price;
};

int main()
{
 union car car1, car2, *car3;
 return 0;
}
```

Another way of creating union variables is:

```
union car
{
 char name[50];
 int price;
} car1, car2, *car3;
```

In both cases, union variables *car1*, *car2*, and a union pointer *car3* of union cartype are created.

Access members of a union

We use the . operator to access members of a union. To access pointer variables, we use also use the -> operator.

In the above example,

- To access *price* for car1, car1.price is used.
- To access *price* using car3, either (*car3).price or car3->price can be used.

Advantages & Disadvantages of Unions:

Advantages of union

Here, are pros/benefits for using union:

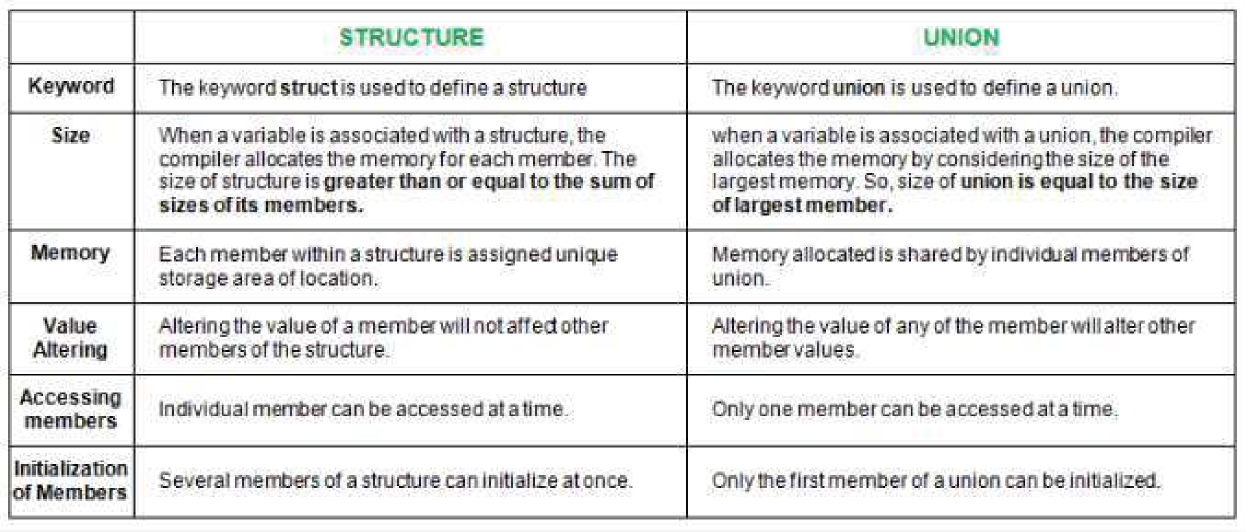
- It occupies less memory compared to structure.
- When you use union, only the last variable can be directly accessed.
- Union is used when you have to use the same memory location for two or more data members.
- It enables you to hold data of only one data member.
- Its allocated space is equal to maximum size of the data member.

Disadvantages of union

Here, are cons/drawbacks for using union:

- You can use only one union member at a time.
- All the union variables cannot be initialized or used with varying values at a time.
- Union assigns one common storage space for all its members.

Difference between unions and structures



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Why this difference in the size of union and structure variables?

Here, the size of *sJob* is 40 bytes because

- the size of name[32] is 32 bytes
- the size of salary is 4 bytes
- the size of workerNo is 4 bytes

However, the size of *uJob* is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (name[32]), is 32 bytes.

Object Oriented Programming:

Introduction:

C++ is an extension of the C programming language, which means that all of the C library functions can be used in a C++ application. C++ was finally standardized in June 1998, but its history can be traced back almost 20 years. This document will begin with how C++ has evolved over the years and introduce some of the language's features. Since C++ is an object-oriented programming language, it is important to understand the concepts of object-oriented programming. The remainder of this document will discuss object-oriented programming, C++ classes and how they are implemented, introduce some new keywords, and mention some basic I/O differences between C and C++.

Evolution of C++

C++ was originally known as “C with Classes.” Bjarne Stroustrup from AT&T Laboratories developed the language in 1980. Bjarne needed to add speed to simulations that were written in Simula-67. Since C was the fastest procedural language, he decided to add classes, function argument type checking and conversion, and other features to it. Around the 1983/1984 time frame, virtual functions and operator overloading were added to the language, and it was decided that “C with Classes” be renamed to C++. The language became available to the public in 1985 after a few refinements were made. Templates and exception handling were added to C++ in 1989. The Standard Template Library (STL) was developed by Hewlett-Packard in 1994, and was ultimately added to the draft C++ standard. The final draft was accepted by the X3J16 subcommittee in November 1997, and received final approval from the International Standards Organization (ISO) in June 1998 to officially declare C++ a standard.

Programming Paradigms

There are two programming paradigms:

- Procedure-Oriented
- Object-Oriented

Examples of procedure-oriented languages include: C, Pascal, FORTRAN

Examples of object-oriented languages include: C++, SmallTalk, Eiffel.

A side-by-side comparison of the two programming paradigms clearly shows how object-oriented programming is vastly different from the more conventional means of programming:

Procedure-Oriented Programming	Object-Oriented Programming
• Top Down/Bottom Up Design	• Identify objects to be modeled
• Structured programming	• Concentrate on what an object does
• Centered around an algorithm	• Hide how an object performs its tasks
• Identify tasks; how something is done	• Identify an object's behavior and attributes

Structure of C++:

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

- **Object –** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class –** A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
- **Methods –** A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables –** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

C++ Program Structure

Let us look at a simple code that would print the words *Hello World*.

```
#include <iostream>
using namespace std;

// main() is where program execution begins.
int main() {
  cout << "Hello World"; // prints Hello World
  return 0;
}
```

Let us look at the various parts of the above program –

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.

- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
- The next line '**// main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.
- The next line **cout << "Hello World";** causes the message "Hello World" to be displayed on the screen.
- The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

Semicolons and Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements −

```
x = y;
y = y + 1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example −

```
{
  cout << "Hello World"; // prints Hello World
  return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

```
x = y;
y = y + 1;
add(x, y);
```

is the same as

```
x = y; y = y + 1; add(x, y);
```

A SIMPLE PROGRAM:

C++ "Hello World!" Program

```
// Your First C++ Program

#include <iostream>

int main() {
   std::cout << "Hello World!";
   return 0;
}
```

Output

```
Hello World!
```

STORAGE CLASSES IN C++

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

- auto
- register
- static
- extern
- mutable

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 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++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream>

// Function declaration
void func(void);

static int count = 10; /* Global variable */

main() {
  while(count--) {
    func();
  }

  return 0;
}

// Function definition
void func( void ) {
  static int i = 5; // local static variable
  i++;
  std::cout << "i is " << i ;
  std::cout << " and count is " << count << std::endl;
}
```

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

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 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 as all it does is point 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 be used in other files also, then *extern* will be used in another file to give 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.cpp

```
#include <iostream>
int count ;
extern void write_extern();

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

Second File: support.cpp

```
#include <iostream>

extern int count;

void write_extern(void) {
  std::cout << "Count is " << count << std::endl;
}
```

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows –

```
$g++ main.cpp support.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows –

```
$./write
5
```

The mutable Storage Class

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.

Difference between C and C++

Similarities between C and C++ are:

- Both the languages have a similar syntax.
- Code structure of both the languages are same.
- The compilation of both the languages is similar.
- They share the same basic syntax. Nearly all of C's operators and keywords are also present in C++ and do the same thing.
- C++ has a slightly extended grammar than C, but the basic grammer is the same.
- Basic memory model of both is very close to the hardware.
- Same notions of stack, heap, file-scope and static variables are present in both the languages.

Differences between C and C++ are:

C++ can be said a superset of C. Major added features in C++ are Object-Oriented Programming, Exception Handling and rich C++ Library.

Below is the table of differences between C and C++:

C	C++
C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.

C	C++
C does no support polymorphism, encapsulation, and inheritance which means that C does not support object oriented programming.	C++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language.
C is a subset of C++.	C++ is a superset of C.
C contains 32 keywords.	C++ contains 52 keywords.
For the development of code, C supports procedural programming.	C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.
Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in form of an object in C++.
C does not support information hiding.	Data is hidden by the Encapsulation to ensure that data structures and operators are used as intended.
Built-in data types is supported in C.	Built-in & user-defined data types is supported in C++.
C is a function driven language because C is a procedural programming language.	C++ is an object driven language because it is an object oriented programming.
Function and operator overloading is not supported in C.	Function and operator overloading is supported by C++.
C is a function-driven language.	C++ is an object-driven language
Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Namespace features are not present inside the C.	Namespace is used by C++, which avoid name collisions.
Header file used by C is stdio.h.	Header file used by C++ is iostream.h.

C	C++
Reference variables are not supported by C.	Reference variables are supported by C++.
Virtual and friend functions are not supported by C.	Virtual and friend functions are supported by C++.
C does not support inheritance.	C++ supports inheritance.
Instead of focusing on data, C focuses on method or process.	C++ focuses on data instead of focusing on method or procedure.
C provides malloc() and calloc() functions for dynamic memory allocation, and free()for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
Direct support for exception handling is not supported by C.	Exception handling is supported by C++.
scanf() and printf() functions are used for input/output in C.	cin and cout are used for input/output in C++.

Unit – V: Classes and Objects

Data Members-Member Functions - Object Oriented Concepts-Class-Object- Encapsulation-Abstraction - Polymorphism (Function Overloading and Operator Overloading) Inheritance (Inheritance Forms & Types)

DATA MEMBERS AND MEMBER FUNCTIONS IN C++ PROGRAMMING

"Data Member" and **"Member Functions"** are the new names/terms for the members of a class, which are introduced in C++ programming language.

The variables which are declared in any class by using any fundamental data types (like int, char, float etc) or derived data type (like class, structure, pointer etc.) are known as **Data Members**. And the functions which are declared either in private section of public section are known as **Member functions**.

There are two **types of data members/member functions in C++**:

1. Private members
2. Public members

1) Private members

The members which are declared in private section of the class (using private access modifier) are known as private members. Private members can also be accessible within the same class in which they are declared.

2) Public members

The members which are declared in public section of the class (using public access modifier) are known as public members. Public members can access within the class and outside of the class by using the object name of the class in which they are declared.

Consider the example:

```
class Test
{
        private:
                int a;
                float b;
                char *name;

                void getA() { a=10; }
                ...;

        public:
                int count;
                void getB() { b=20; }

                ...;
};
```

Here, a, b, and name are the private data members and count is a public data member.
While, getA() is a private member function and getB() is public member functions.

C++ program that will demonstrate, how to declare, define and access data members an member functions in a class?

```
#include <iostream>
#include <string.h>
using namespace std;

#define MAX_CHAR 30

//class definition
class person
{
    //private data members
    private:
        char name [MAX_CHAR];
        int age;

    //public member functions
    public:
        //function to get name and age
        void get(char n[], int a)
        {
            strcpy(name , n);
            age = a;
        }

        //function to print name and age
        void put()
        {
            cout<< "Name: " << name <<endl;
            cout<< "Age: " <<age <<endl;
        }
};

//main function
int main()
{
    //creating an object of person class
    person PER;

    //calling member functions
    PER.get("Manju Tomar", 23);
    PER.put();

    return 0;
}
```

Output

```
Name: Manju Tomar
Age: 23
```

As we can see in the program, that private members are directly accessible within the member functions and member functions are accessible within in main() function (outside of the class) by using period (dot) operator like object_name.member_name;

OBJECT ORIENTED CONCEPTS:

An ***abstract data type*** (ADT) is a user-defined data type where objects of that data type are used through provided functions without knowing the internal representation. For example, an ADT is analogous to, say an automobile transmission. The car's driver knows how to operate the transmission, but does not know how the transmission works internally.

The ***interface*** is a set of functions within the ADT that allow access to data.

The ***implementation*** of an ADT is the underlying data structure(s) used to store data. It is important to understand the distinction between a ***class*** and an ***object***. The two terms are often used interchangeably, however there are noteworthy differences. Classes will be formally introduced later in this document, but is mentioned here due to the frequent use of the nomenclature in describing OOP.

Classes & Objects

An object is a basic unit in object-oriented programing. An object contains data and methods or functions that operate on that data. Objects take up space in memory.
A class, on the other hand, is a blueprint of the object. Conversely, an object can be defined as an instance of a class. A class contains a skeleton of the object and does not take any space in the memory.

Let us take an **Example** of a car object. A car object named "Maruti" can have data such as color; make, model, speed limit, etc. and functions like accelerate. We define another object "ford". This can have similar data and functions like that of the previous object plus some more additions.

Similarly, we can have numerous objects of different names having similar data and functions and some minor variations.

Thus instead of defining these similar data and functions in these different objects, we define a blueprint of these objects which is a class called Car. Each of the objects above will be instances of this class car.

The differences are summarized below:

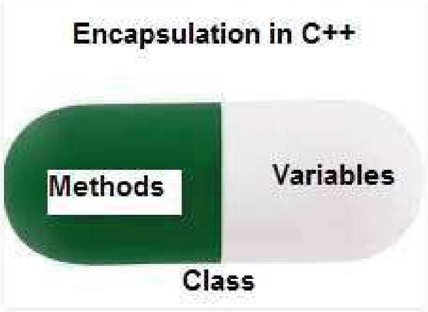
Class	**Object**
• Defines a model	• An instance of a class
• Declares attributes	• Has state
• Declares behavior	• Has behavior
• An ADT	• There can be many ***unique*** objects of the same class

There are four main attributes to object-oriented programming:

- Data Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".



Encapsulation in C++

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

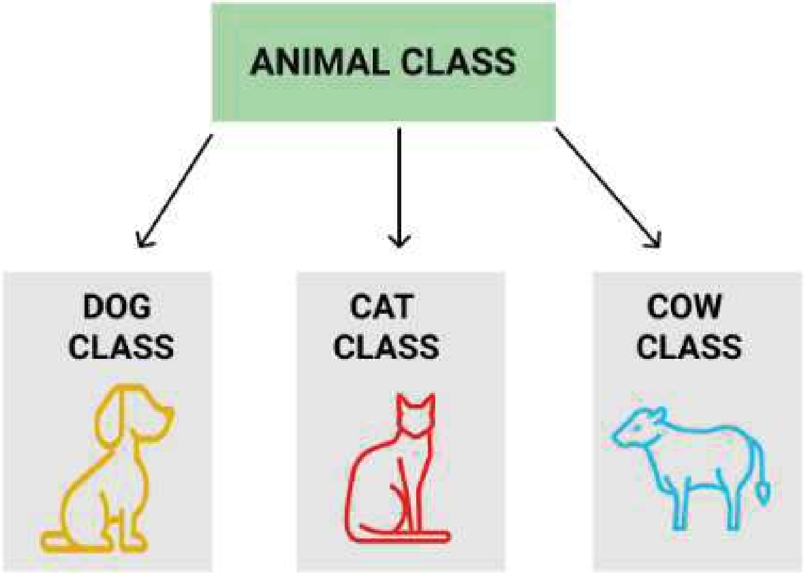
- *Abstraction using Classes*: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

- *Abstraction in Header files*: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

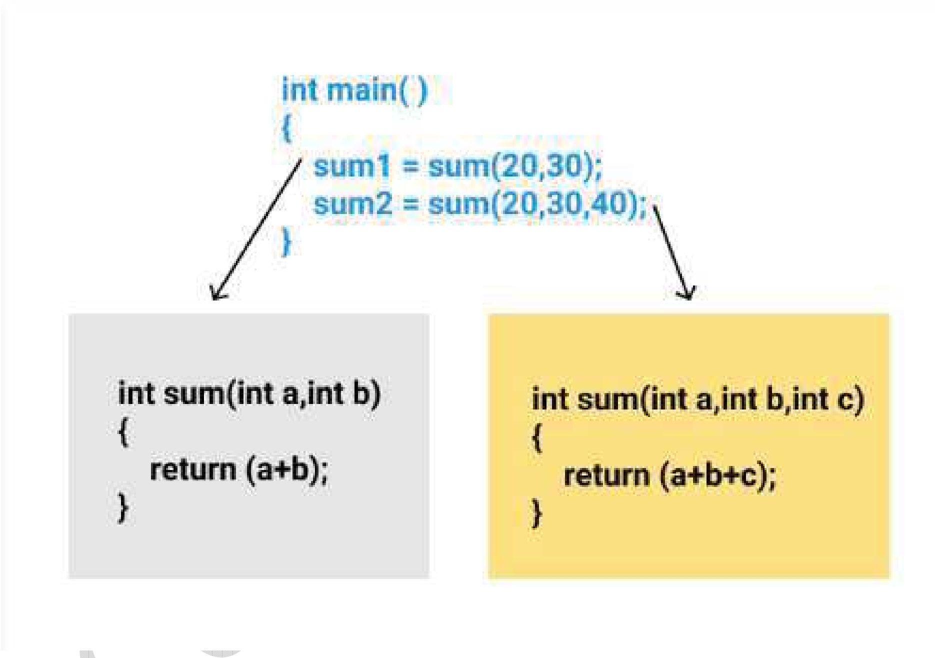
Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.
An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.
C++ supports operator overloading and function overloading.

- *Operator Overloading*: The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading*: Function overloading is using a single function name to perform different types of tasks.

  Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



```
int main( )
{
   sum1 = sum(20,30);
   sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
   return (a+b);
}
```

```
int sum(int a,int b,int c)
{
   return (a+b+c);
}
```