



**BAKLIWAL
FOUNDATION**
COLLEGE OF ARTS, COMMERCE & SCIENCE

BCA I YEAR I SEMESTER

BCA1.2 C PROGRAMMING

TABLE OF CONTENTS

Unit	Topics	Page No.
1	Programming Structure	4
	Problem Solving Techniques	6
	Development Tools	6
2	C Character Set	15
	Operators & Expressions	18
	Typedef	25
	Type Conversion	25
	Constants	28
	Strings	30
	Enum Data types	33
	Operator Precedence and Associativity	34
	Library Functions	35
	Control Structure	43
	Iteration Statements	49
	Jump Statements	53
3	Arrays	57
	Strings	61
	Function	66
	Arrays with Function	76
	Storage Classes	79
4	Structure	86
	Union	94
	Pointer	96
	File Handling	106

UNIT I

Introduction to C Language

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called ***Programming in C***, and the title that covered ANSI C was called ***Programming in ANSI C***. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is **reliable**, **simple** and **easy** to use. often heard today is – “C has been already superceded by languages like C++, C# and Java.

Program

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. So a computer program is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's instruction set. And the approach or method that is used to solve the problem is known as an algorithm.

So for as programming language concern these are of two types.

- 1) Low level language
- 2) High level language

Low level language:

Low level languages are machine level and assembly level language. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to

implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understood by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

High level language:

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Three types of translator are there – Compiler Interpreter Assembler

Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

Integrated Development Environments (IDE)

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows usto easily manage large software programs, edit files in windows, and compile, link,run, and debug programs.

On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++.

1.1 PRORAMMING STRUCTURE

Behind all of the software we use on a daily basis, there's a code being run with all sorts of terms and symbols. Surprisingly, it can often be broken down into three simple

programming structures called sequences, selections, and loops. These come together to form the most basic instructions and algorithms for all types of software.

1.1.1 SEQUENCE

A sequence is a series of actions that is completed in a specific order. Action 1 is performed, then Action 2, then Action 3, etc., until all of the actions in the sequence have been carried out.

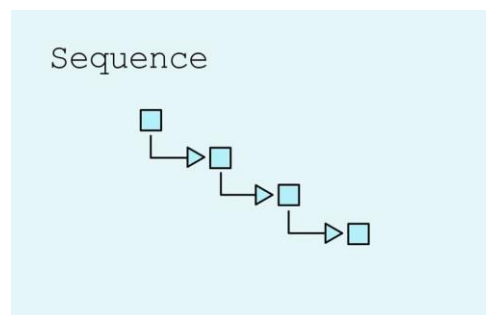


Fig 1.1 A sequence of events

A sequence we do every day is a morning routine. You might wake up, drink some water, take a shower, eat breakfast, and so on. Everyone's routine is different, but they're all made up of a sequence of various actions.

1.1.2 SELECTION

Selections are a bit different. Instead of following a specific order of events, they ask a question in order to figure out which path to take next.

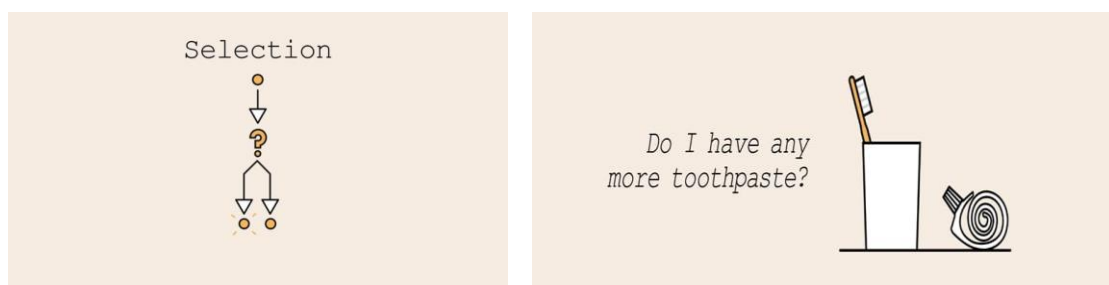


Fig 1.2 Selection between tasks

Let's say you go to brush your teeth, and you find that you're out of toothpaste. You'd then ask, "Do I have any more toothpaste?" If the answer is no, then you would add it to your shopping list. But if the answer is yes, you would just use the toothpaste. This is really all a selection is doing: answering a question based on what it finds.

1.1.3 ITERATION AND MODULAR

Iteration is the process of repeating steps. For example, a very simple algorithm for eating breakfast cereal might consist of these steps: put cereal in bowl. add milk to cereal. spoon cereal and milk into mouth.

Modular programming is the process of subdividing a computer program into separate sub-programs. Every module has an implementation part that hides the code and any other private implementation detail the clients modules should not care of.

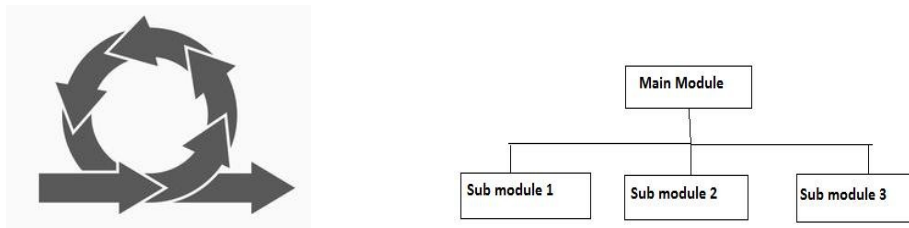


Fig 1.3 Representation of Iteration and Modulation

1.2 PROBLEM SOLVING TECHNIQUES

Steps to Solve a Problem With the Computer

problem solving through programming in c

Step 1: Understanding the Problem:

Here we try to understand the problem to be solved in totally. Before with the next stage or step, we should be absolutely sure about the objectives of the given problem.

Step 2: Analyzing the Problem:

After understanding thoroughly the problem to be solved, we look at different ways of solving the problem and evaluate each of these methods.

The idea here is to search for an appropriate solution to the problem under consideration. The end result of this stage is a broad overview of the sequence of operations that are to be carried out to solve the given problem.

Step 3: Developing the solution:

Here, the overview of the sequence of operations that was the result of the analysis stage is expanded to form a detailed step by step solution to the problem under consideration.

Step 4: Coding and Implementation:

The last stage of problem-solving is the conversion of the detailed sequence of operations into a language that the computer can understand

1.3 DEVELOPMENT TOOLS

1.3.1 ALGORITHM

Algorithm “why do we need to study algorithm”? If you want to be a computer professional, there are both practical and theoretical reasons to study algorithm. From a practical point of view, you should know the standard set of important algorithms from different areas of computing; In addition, you should be able to design new

algorithms and analyze their efficiency. From the theoretical standpoint the study of algorithms, sometimes called **algorithms**.

Another reason to study algorithms is the usefulness in developing analytical skills. After that, algorithms can be seen as special kinds of solutions for the problems, but precisely defined procedures for getting answers. Consequently specific algorithm design techniques can be interpreted and involved course the precision inherently imposed by algorithmic thinking limits the kinds of problems than can be solved with an algorithm.

In the 1930s, before the advent of computers, mathematicians worked very actively to formalize and study the notation of simple instructions were given for solving a problem or computer as a solution. Various formal modes of computation were desired and investigated. Much of the emphasis in the early work in this field computability theory was on describing and characterizing those problems that could be solved algorithmically and on exhibiting some problems that could not be solved. One of the important negative results was insolvability of the “halting problem”. The halting problem is to determine whether an arbitrary given algorithm (or computer program) will eventually halt (rather than, say get into an infinite loop) while working on a given input.

An algorithm is a sequence of unambiguous instructions for solving a problem that is a sequence of computational steps that transform them input into the output.

An algorithm has the following properties:

1. **Input:** The algorithms get input.
2. **Output:** The algorithms produce output.
3. **Definiteness:** Each instruction to represent with clear & unambiguous.
4. **Finiteness:** The algorithm terminates; that is it terminates after finite number of steps.
5. **Correctness:** The produced output by the algorithm is correct.

EXAMPLE

Consider the given algorithm to find the largest of three numbers x, y and z.

Step 1: Start the program

Step 2: Read the value x, y and z

Step 3: To compare $((x > y) \text{ and } (x > z))$ then

print x

else if $(y > z)$

```
        print y
    else
        print z
```

Step 4: Stop the program

Thus in the given above algorithm, consider the first step is to start the program, the second step to read the values of x, y and z and after that compare the first number with second number and also compare the first number with third number, if the condition is true then print the value of first number else compare the second number with third number if the condition is satisfied then display the second number otherwise display the third number.

1.3.2 FLOWCHARTS

Definition Of Flow Chart

A flow chart is a graphical or symbolic representation of a process. Each step in the process is represented by a different symbol and contains a start description of the process.

Advantages Of Flowchart

The advantages of flowchart are,

1. *Communication*: Flowchart is used for better way of communication in all connections.
2. *Effective Analysis*: By using flowchart, problems are analyzed in better manner.
3. *Proper Documentation*: Flowchart serves as good program documentation.
4. *Efficient Coding*: The flowcharts act as a guideline during the system analysis and development phase.
5. *Proper Debugging*: The flowchart helps in debugging process.
6. *Efficient Program Maintenance*: The maintenance of operating program Is an easy way for drawing the flowchart.

Limitations Of Flowchart

Some of the limitations of flowchart are,

1. *Complex Logic*: The program logic is very complicated; in this manner drawing flowchart is difficult

2. *Alterations And Modifications:* Alteration and modifications cannot be made and hence flowchart is very complex process
3. *Reproduction:* In flowchart, symbols cannot be typed, hence it becomes a problem

1.3.3 PSEUDOCODES

It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program's flow, but excludes underlying details.

- “Pseudo” means initiation or false.
- “Code” means the set of statements or instructions written in a programming language.
- Pseudocode is also called as “Program Design Language [PDL]”.
- Pseudocode is a Programming Analysis Tool, which is commonly used for planning the program logic.
- Pseudocode is written in normal English and cannot be understood by the computer.
- Set of instructions that mimic programming language instructions
- An informal high-level description of the operating principle of a computer program. It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.

Rules for writing Pseudocode

1. Write one statement per line
2. Capitalize initial keywords (READ, WRITE, IF, WHILE, UNTIL).
3. Indent to show hierarchy.
4. End multiline structure.
5. Keep statements language.

Advantages

- It can be done easily in any word processor.
- It can be easily modified as compared to flowchart.
- Its implementation is very useful in structured design elements.
- It can be written easily.
- It can be read and understood easily.

- Converting a pseudocode to programming language is very easy compared with converting a flowchart to programming language.

Disadvantage

- It is not visual.
- We do not get a picture of the design.
- There is no standardized style or format.
- For a beginner, it is more difficult to follow the logic or write pseudocode as compared to flowchart.

1.3.3.1 Definition

The pseudocode in C is an informal way of writing a program for better human understanding. It is written in simple English, making the complex program easier to understand. Pseudocode cannot be compiled or interpreted.

1.3.3.2 Characteristics

The characteristics of pseudocode are:

Pseudocode is an informal way of expressing ideas and algorithms during the development process.

Pseudocode uses simple and concise words and symbols to represent the different operations of the code.

Pseudocode can be used to express both complex and simple processes

Example

To write pseudocode for to prepare a student mark sheet processing using control statement

Step 1: Start the program

Step 2: Initialize the variables rollno, stud_name, m1, m2, total, Average and result

Step 3: Read the values for rollno, stud_name, m1, m2

Step 4: To calculate total=m1+m2

Step 5: To calculate average=total/2

1.3.4 DEVELOPING ALGORITHM

The first step in algorithm problem solving is to understand the complexity of the problem given. The fundamental importance of both algorithms and data structures for

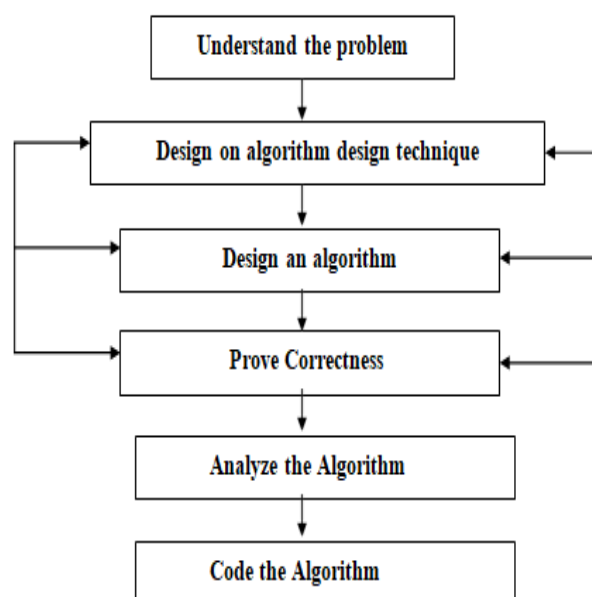
computer programming is very useful. An algorithm design techniques(or “strategy” or “paradigm”) is a general approach for solving problems algorithmically that is applicable to a variety of problems from different area of computing. There are two methods used for designing an algorithm.

1. Pseudocode
2. Flowchart

A pseudocode is a mixture of a natural language is usually more precise than a natural language and its usage often yields more succinct algorithm description. Pseudocode for the statement used such as for, if and while and also used for □ assignment operation two slashes for // comments.

In the second approach for specifying algorithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithms steps. Once an algorithm has been specified, you have to prove its correctness. That is you have to prove that the algorithm yields a required result for every legitimate input in finite amount of time.

After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithmic efficiency: time efficiency and space efficiency. Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs. Another desirable characteristic of an algorithm is simplicity and generality. Two issues have; in generality of the problem the algorithm solves and the range of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms. Consider for example, the problem for determining whether two integers are relatively prime. Most algorithms are designed to be ultimately implemented as a computer programs.



STUDY OF ALGORITHM

There are four distinct areas to know about algorithms includes,

1. HOW TO DEVISE ALGORITHMS

To create an algorithm, it may never fully optimized. One of the major goals of an algorithm is to use variety of designing techniques to yield good algorithms. By using dynamic programming techniques, it is used to devise an algorithm (i.e. devise means divided) in a good manner.

2. HOW TO VALIDATE ALGORITHMS

After devising the algorithm, the next field is validating the algorithms. Thus computer will produce correct result for all possible legal inputs.

3. HOW TO ANALYSE ALGORITHMS

Analysis of algorithm is an important part of computer system. It is used to determine the amount of resources such as time and storage necessary to execute. An algorithm can be given in many ways. For example, it can be written down in English or French or any other natural language.

4. HOW TO TEST A PROGRAM

Testing a program consists of two phases: debugging and profiling (or performance measurement). Debugging means to identify errors in a program. Performance measurement is the process of executing a correct program, it measure time and space it takes to compute the results.

Pseudocode is nothing but the actual code of computer languages such as C, C++ and Java. Let us differentiate between the algorithm and program. Algorithm is a sequence of unambiguous instructions but program means it represents more precise and concise notations called a program.

Another way to represent pseudocode is one of the tools that can be used to write a preliminary plan that can be developed into a computer program. Pseudocode is a generic way of describing an algorithm without use of any specific programming language syntax.

EXAMPLE

To write pseudocode for sum of three numbers using sequence structure

Step 1: Start the program

Step 2: Initialize the variables sum, number1, number2, number3 of type integer

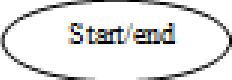
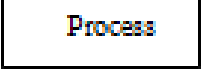
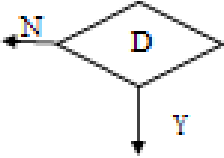


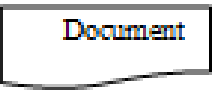

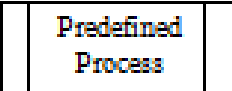
Step 3: Read number1, number2 and number3

Step 4: Calculate $\text{sum} = \text{number1} + \text{number2} + \text{number3}$

Step 5: Print sum

Step 6: End the program

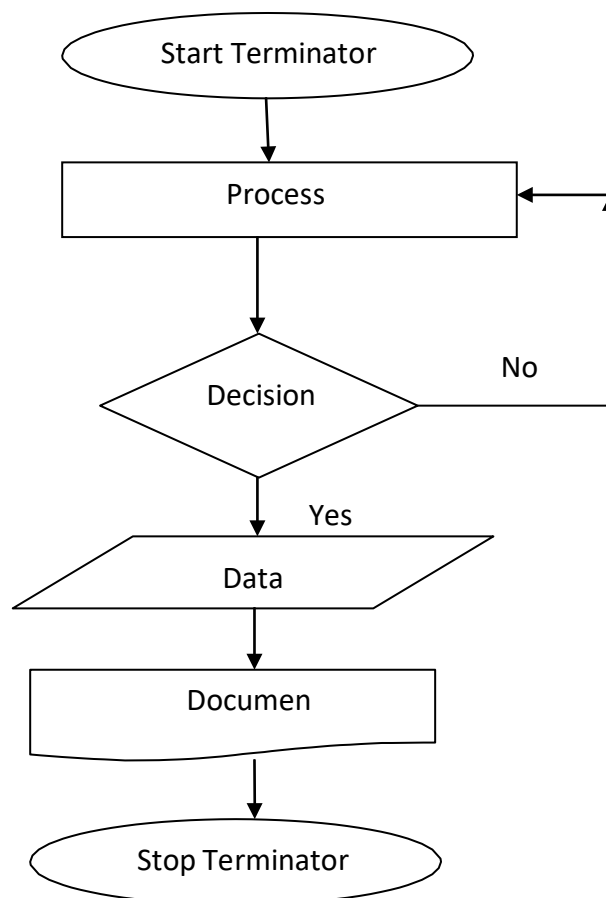
1.3.5 DRAWING FLOWCHARTS

S. No	SYMBOL	SYMBOL NAME	FUNCTION
1		Oval/Terminator	Start/end
2		Box/Process	Activity in the Process
3		Diamond/Decision	Indicating a Decision Point
4		Circle/Connector	The Particular Step is Connected to Another Page
5		Triangle/Data	Data Input/Output(I/O) for a Process
6		Document	Indicating a Document or Report
7		Arrow/Connector	Direction of Process Flow
8		Predefined Process	Invoke a Subroutine or Interrupt Program

The following symbols that are commonly used in flowcharts they are,

1. **OVAL/TERMINATOR:** Ovals indicates both the **start and end** of the process.
2. **BOX/PROCESS:** A Rectangular flow shape indicates the **activity in the process.**
3. **DIAMAND/DECISION:** Diamonds indicates the **decision point**, such as yes/no or on/off or go/not go.
4. **CIRCLE/CONNECTOR:** A **circle** indicates the particular step is connected to another page or part of the flowchart.
5. **TRIANGLE/DATA:** A triangle indicates **data input or output (I/O)** for a process.
6. **DOCUMENT:** A document is used to indicate a **document or report.**
7. **FLOW LINE/ARROW/CONNECTOR:** An Arrow indicates to show the direction that the process flows.
8. **PREDEFINED PROCESS:** A **predefined process** is used to indicate a subroutine or interrupt program.

The following flow chart shows the symbolic representation of flow diagram.



UNIT II

2.1 C CHARACTER SET

A character denotes any alphabet, digit or special symbol used to represent information.

Valid alphabets, numbers and special symbols allowed in C are

Alphabets	A, B,, Y, Z a, b,, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * () _ - + = \ { } [] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

2.2 TOKENS

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

```
printf("Hello, World! \n");
```

The individual tokens are:

```
printf(  
"Hello, World! \n"
```

2.3 IDENTIFIERS

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

- 1) name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.
- 2) first characters should be alphabet or underscore
- 3) name should not be a keyword

4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

Identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

2.4 KEYWORDS

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords**. These words are predefined and always written in lower case or small letter. These keywords can't be used as a variable name as it assigned with fixed meaning.

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

2.5 VARIABLES

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax: `int a; char c; float f;`

Variable initialization

When we assign any initial value to variable during the declaration, is called initialization of variables. When variable is declared but contain undefined value then it is called garbage value. The variable is initialized with the assignment operator such as

Data type Variable name = Constant;

Example: `int a=20;`

Or `int a;`

`a=20;`

2.6 DATA TYPES

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time. C has the following 4 types of data types

Basic built-in data types: int, float, double, char

Enumeration data type: enum

Derived data type: pointer, array, structure, union

Void data type: void

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character 6, or a semicolon similarly A variable declared char can only store character type value.

2.7 QUALIFIERS

There are two types of type qualifier in c

Size qualifier: short, long

Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char	1	-128 to 127
	Unsigned char	1	0 to 255
int	int or signed int	2	-32768 to 32767
	unsigned int	2	0 to 65535
	short int or signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
float	Float	4	-3.4E-38 to 3.4E+38
double	Double	8	1.7E-308 to 1.7E+308
	Long double	10	3.4E-4932 to 1.1E+4932

2.8 OPERATORS and EXPRESSIONS

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational for example:-

`int z = x + y` // arithmetic expression

`a > b` // relational

`a == b` // logical

`func(a, b)` // function call

Expressions consisting entirely of constant values are called *constant expressions*. So, the expression

`121 + 17 - 110` is a constant expression because each of the terms of the expression is a constant value. But if `i` were declared to be an integer variable, the expression

`180 + 2 - i` would not represent a constant expression.

Operator

This is a symbol use to perform some operation on variables, operands or with the constant. Some operator required 2 operand to perform operation or Some required single operation.

Several operators are there those are, arithmetic operator, assignment, increment , decrement, logical, conditional, comma, size of , bitwise and others.

2.8.1 ARITHMETIC OPERATORS

This operator used for numeric calculation. These are of either Unary arithmetic operator, Binary arithmetic operator. Where Unary arithmetic operator required only one operand such as +, -, ++, --, !, tiled. And these operators are addition, subtraction, multiplication, division. Binary arithmetic operator on other hand required two operand and its operators are +(addition), -(subtraction),

*(multiplication), /(division), %(modulus). But modulus cannot applied with floating point operand as well as there are no exponent operator in c.

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Unary (+) and Unary (-) is different from addition and subtraction.

When both the operand are integer then it is called integer arithmetic and the result is always integer. When both the operand are floating point then it is called floating arithmetic and when operand is of integer and floating point then it is called mix type or mixed mode arithmetic . And the result is in float type.

2.8.2 RELATIONAL OPERATORS

It is used to compare the value of two expressions depending on their relation. Expression that contains a relational operator is called a relational expression.

Here the value is assigned according to true or false value.

$(a \geq b) \text{ || } (b > 20)$

$(b > a) \text{ \&\& } (e > b)$

$0(b \neq 7)$

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

2.8.3 LOGICAL OPERATORS

Operator used with one or more operand and returns either value zero (for false) or one (for true). The operand may be constant, variables or expressions. And the expression that combines two or more expressions is termed as logical expression. C has three logical operators

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

Where logical NOT is a unary operator and other two are binary operator. Logical AND gives result true if both the conditions are true, otherwise result is false. And logical OR gives result false if both the conditions are false, otherwise result is true.

2.8.4 BIT-WISE OPERATORS

Bitwise operators permit a programmer to access and manipulate data at the bit level. These operators can operate on integer and character values but not on float and double. In bitwise operators, the function `showbits()` is used to display the binary representation of any integer or character value.

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

In one's complement, all 0s change to 1 and all 1s change to 0. In the bitwise OR, its value would be obtained by 0 to 2 bits. As the bitwise OR operator is used to set on a particular bit in a number. It operates on 2 operands and the operands are compared on a bit-by-bit basis. And hence both the operands are of the same type.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100

	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

2.8.5 INCREMENT and DECREMENT OPERATORS

The Unary operator ++, --, is used as increment and decrement which acts upon single operand. Increment operator increases the value of variable by one

Similarly decrement operator decreases the value of the variable by one. And these operator can only be used with the variable, but can't use with expression and constant as ++6 or ++(x + y + z). It again categories into prefix post fix. In the prefix the value of the variable is incremented 1st, then the new value is used, where as in postfix the operator is written after the operand (such as m++,m--).

EXAMPLE

```
let y=12;
z= ++y;
y= y+1;
z= y;
```

Similarly in the postfix increment and decrement operator is used in the operation. And then increment and decrement is performed.

EXAMPLE

```
let x= 5;
y= x++;
y=x;
x= x+1;
```

2.8.6 ASSIGNMENT OPERATOR

A value can be stored in a variable with the use of assignment operator. The assignment operator (=) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side should be variable or constant or any expression. When variable on the left hand side is occur on the right hand side then we can avoid by writing the compound statement. For example,

```
int x= y;
```

```
int Sum = x+y+z;
```

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2

$\wedge=$	bitwise exclusive OR and assignment operator	$C \wedge= 2$ is same as $C = C \wedge 2$
$ =$	bitwise inclusive OR and assignment operator	$C = 2$ is same as $C = C 2$

2.8.7 CONDITIONAL OPERATORS

It sometimes called as ternary operator. Since it required three expressions as operand and it is represented as $(? , :)$.

SYNTAX

$exp1 ? exp2 : exp3$

Here $exp1$ is first evaluated. It is true then value return will be $exp2$. If false then $exp3$.

EXAMPLE

```
void main()
{
    int a=10, b=2
    int s= (a>b) ? a:b; printf("value is:%d");
}
```

Output:

Value is:10

2.8.8 SPECIAL OPERATORS

Comma Operator

Comma operator is use to permit different expression to be appear in a situation where only one expression would be used. All the expression are separator by comma and are evaluated from left to right.

EXAMPLE

```
int i, j, k, l;

for(i=1,j=2; i<=5; j<=10; i++;j++)
```


Sizeof Operator

Size of operator is a Unary operator, which gives size of operand in terms of byte that occupied in the memory. An operand may be variable, constant or data type qualifier.

Generally it is used make portable program(program that can be run on different machine) . It determines the length of entities, arrays and structures when their size are not known to the programmer. It is also use to allocate size of memory dynamically during execution of the program.

EXAMPLE

```
main( )  
  
{  
  
    int sum; float f;  
  
    printf( "%d%d" ,size of(f), size of (sum) );  
  
    printf("%d%d", size of(235 L), size of(A));  
  
}
```

2.9 TYPEDEF

The C programming language provides a keyword called typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

After this type definitions, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example:

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

```
typedef unsigned char byte;
```

You can use typedef to give a name to user defined data type as well.

2.10 TYPE CONVERSIONS

A type cast is basically a conversion from one type to another. There are two types of type conversion:

Implicit Type Conversion



Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int -> unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

// An example of implicit conversion

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

```
int main()
```

```
{
```

```
    int x = 10; // integer x
```

```

char y = 'a'; // character c

// y implicitly converted to int. ASCII

// value of 'a' is 97

x = x + y; // x is implicitly converted to float

float z = x + 1.0;

printf("x = %d, z = %f", x, z);

return 0;

}

```

Output:

x = 107, z = 108.000000

Explicit Type Conversion–

This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type. Type indicated the data type to which the final result is converted.

The syntax in C:

(type) expression

// C program to demonstrate explicit type casting

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

```
int main()
```

```
{
```

```
    double x = 1.2; // Explicit conversion from double to int
```

```
    int sum = (int)x + 1;
```

```
    printf("sum = %d", sum);
```

```
    return 0;
```

```
}
```

Output:

sum = 2

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.

2.11 CONSTANTS

Constant is any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a constant. A constant is an entity that doesn't change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. C constants can be divided into two major categories:

- Primary Constants
- Secondary Constants

These constants are further categorized as

- Numeric constant
- Character constant
- String constant

Numeric constant: Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is -32768 to 32767.

For a 32-bit compiler the range would be even greater. Mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant**. An integer constants are whole number which have no decimal point. Types of integer constants are:

Decimal constant: 0 ----- 9(base 10)

Octal constant: 0 ----- 7(base 8)

Hexa decimal constant: 0----9, A-----F(base 16)

In decimal constant first digit should not be zero unlike octal constant first digit

must be zero (as in 076, 0127) and in hexadecimal constant first two digit should be 0x/ 0X (such as 0x24, 0x87A). By default type of integer constant is integer but if the value of integer constant is exceeds range then value represented by integer type is taken to be unsigned integer or long integer. It can also be explicitly mention integer and unsigned integer type by suffix l/L and u/U.

Real constant is also called floating point constant. To construct real constant we must follow the rule of ,

-real constant must have at least one digit.

-It must have a decimal point.

-It could be either positive or negative.

-Default sign is positive.

-No commas or blanks are allowed within a real constant. Ex.: +325.34
426.0

-32.76

To express small/large real constant exponent (scientific) form is used where number is written in mantissa and exponent form separated by e/E. Exponent can be positive or negative integer but mantissa can be real/integer type, for example $3.6 \times 10^5 = 3.6e+5$. By default type of floating point constant is double, it can also be explicitly defined it by suffix of f/F.

Character constant

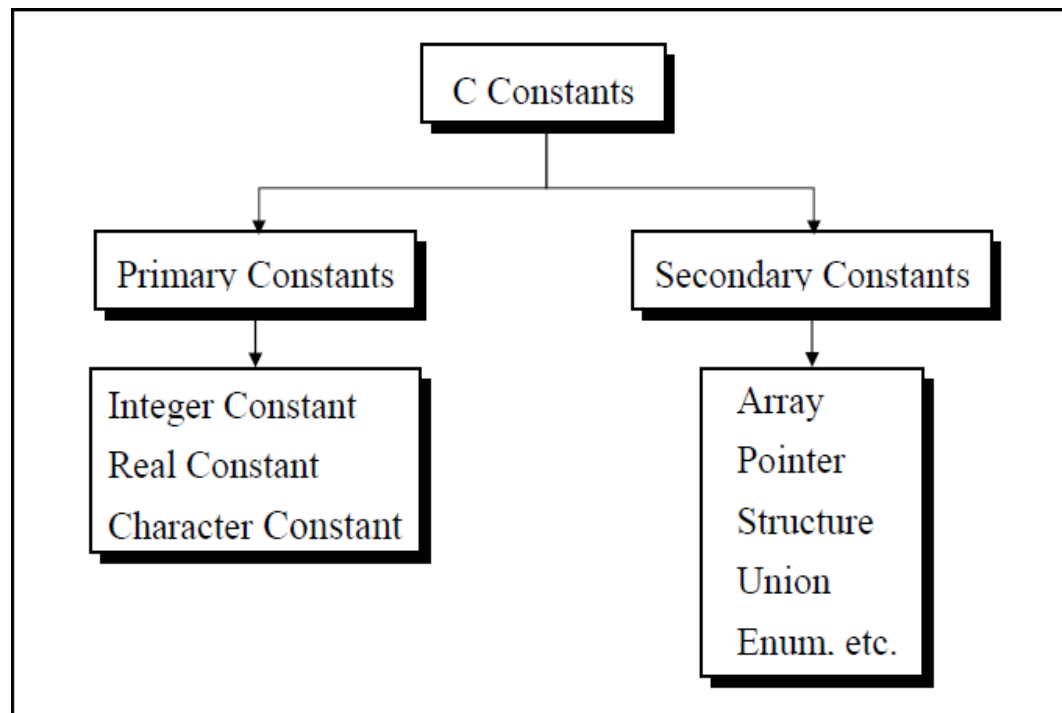
Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as '9', 'c', '\$', ' ' etc. Every character constant has a unique integer like value in machine's character code as if machine using ASCII (American standard code for information interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:

- A Z ASCII value (65-90)
- a z ASCII value (97-122)
- 0----9 ASCII value (48-59)
- ; ASCII value (59)

String constant

Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String

constant has zero, one or more than one character and at the end of the string null character(\0) is automatically placed by compiler.



Some examples are “,sarathina” , “908”, “3”, ” , “A” etc.

In C although same characters are enclosed within single and double quotes it represents different meaning such as “A” and ‘A’ are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

2.12 DECLARING SYMBOLIC CONSTANTS

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constant are generally defined at the beginning of the program as

#define name value

Here name generally written in upper case for example,

#define MAX 10

#define CH ‘b’

#define NAME “sony”

2.13 CHARACTER STRINGS

The string in C programming language is actually a one-dimensional array of characters which isterminated 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 above defined string in 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 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 result something as follows:

Greeting message: Hello

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

- **strcpy**s1, s2;

Copies string s2 into string s1.

- **strcat**s1, s2;

Concatenates string s2 onto the end of string s1.

- **strlen s1;**

Returns the length of string s1.

- **strcmp s1, s2;**

Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

- **strchr s1, ch;**

Returns a pointer to the first occurrence of character ch in string s1.

- **strstr s1, s2;**

Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

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

int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

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

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

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

    return 0;
}
```


When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

2.14 ENUMERATED DATA TYPES

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
enum State {Working = 1, Failed = 0};
```

The keyword 'enum' is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
// The name of enumeration is "flag" and the constant are the values of the flag. By
```

```
// default, the values of the constants are as follows: constant1 = 0, constant2 = 1,
```

```
// constant3 = 2 and so on.
```

```
enum flag{constant1, constant2, constant3, ..... };
```

Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
```

```
// defined as the variable of type week.
```

```
enum week{Mon, Tue, Wed};
```

```
enum week day;
```

```
// Or
```

```
enum week{Mon, Tue, Wed}day;
```

```
// An example program to demonstrate working of enum in C
```

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

```
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
```

```
int main()
```

```
{
```

```

enum week day;

day = Wed;

printf("%d",day);

return 0;

}

```

Output:

2

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Interesting facts about initialization of enum.

1. **Two enum names can have same value.** For example, in the following C program both ‘Failed’ and ‘Freezed’ have same value 0.
2. **If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0.** For example, in the following C program, sunday gets value 0, monday gets 1, and so on.
3. **We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.**
4. **The value assigned to enum names must be some integral constant,** i.e., the value must be in range from minimum possible integer value to maximum possible integer value.
5. All enum constants must be unique in their scope.

2.15 OPERATOR PRECEDENCE and ASSOCIATIVITY

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

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

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

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

2.16 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()*, the *<stdio.h>* should be included.

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.

Example: Square root using `sqrt()` function

Suppose, you want to find the square root of a number. To compute the square root of a number, you can use the `sqrt()` library function. The function is defined in the *math.h* header file.

```
#include <stdio.h>

#include <math.h>

int main()
{
    float num, root;

    printf("Enter a number: ");

    scanf("%f", &num);

    // Computes the square root of num and stores in root

    root = sqrt(num);

    printf("Square root of %.2f = %.2f", num, root);

    return 0;
}
```

When you run the program, the output will be:

Enter a number: 12

Square root of 12.00 = 3.46

Library Functions in Different Header Files

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

2.16.1 MATHS

There is also a list of math functions available, that allows you to perform mathematical tasks on numbers.

To use them, you must include the math.h header file in your program:

```
#include <math.h>
```

Square Root

To find the square root of a number, use the sqrt() function:

Example

```
printf("%f", sqrt(16));
```

Round a Number

The `ceil()` function rounds a number upwards to its nearest integer, and the `floor()` method rounds a number downwards to its nearest integer, and returns the result:

Example

```
printf("%f", ceil(1.4));  
printf("%f", floor(1.4));
```

Power

The `pow()` function returns the value of x to the power of y (xy):

Example

```
printf("%f", pow(4, 3));
```

Other Math Functions

A list of other popular math functions (from the `<math.h>` library) can be found in the table below:

Function	Description
<code>abs(x)</code>	Returns the absolute value of x
<code>acos(x)</code>	Returns the arccosine of x
<code>asin(x)</code>	Returns the arcsine of x
<code>atan(x)</code>	Returns the arctangent of x
<code>cbrt(x)</code>	Returns the cube root of x
<code>cos(x)</code>	Returns the cosine of x
<code>exp(x)</code>	Returns the value of E^x
<code>sin(x)</code>	Returns the sine of x (x is in radians)
<code>tan(x)</code>	Returns the tangent of an angle

2.16.2 STRING HANDLING FUNCTIONS

These string handling functions are defined in `string.h` header file. Hence you need to include this header file whenever you use these string handling functions in your C program.

Following are the most commonly used string handling functions in C. All these functions take either character pointer or character arrays as arguments.

strlen() is used to find the length of a string. strlen() function returns the length of a string. It returns an integer value.

Example:

```
char *str = "Learn C Online";

int strLength;

strLength = strlen(str); //strLength contains the length of the string i.e. 14
```

In the above example, we have declared a string named str and assigned a value – “Learn C Online” to it. In the next line, I have declared an int variable named strLength which will be used to store the length of the string named str. The next line uses the in-built function strlen() which accepts one parameter i.e., a string named str.

strlen() function will return the length (int type) of the string which be stored in the variable strLength. After execution of this line, variable strLength should hold integer value 14.

strcpy()

strcpy() function is used to copy one string to another. strcpy() function accepts 2 parameters. The first parameter is the destination string i.e. the string variable used to copy the string into. The second parameter is the source string i.e. the string variable or string value that needs to be copied to the destination string variable.

The Destination_String should be a variable and Source_String can either be a string constant or a variable.

Syntax:

```
strcpy(Destination_String,Source_String);
```

Example:

```
char *Destination_String;

char *Source_String = "C String functions";

strcpy(Destination_String,Source_String);

printf("%s", Destination_String);
```

Output:

C String functions

In the above example, I have declared 2 strings. The first string variable is named “Destination_String” and the second string is named “Source_String” with a string value assigned to it.

In the next line, I am using strcpy() function with 2 parameters.

This in-built string function will copy the value of Source_String into Destination_String.

The variable Destination_String will now hold the value – “C String functions”.

strncpy()

strncpy() is used to copy only the left-most n characters from source to destination. The Destination_String should be a variable and Source_String can either be a string constant or a variable.

Syntax:

```
strncpy(Destination_String, Source_String, no_of_characters)
```

strcat()

strcat() is used to concatenate two strings. The Destination_String should be a variable and Source_String can either be a string constant or a variable.

Syntax:

```
strcat(Destination_String, Source_String);
```

Example:

```
char *Destination_String = "Learn ";  
char *Source_String = "String functions";  
strcat(Destination_String, Source_String);  
puts( Destination_String);
```

Output:

Learn String functions

strncat()

strncat() is used to concatenate only the leftmost n characters from the source with the destination string.

The Destination_String should be a variable and Source_String can either be a string constant or a variable.

Syntax:

```
strncat(Destination_String, Source_String, no_of_characters);
```


Example:

```
char *Destination_String="Visit ";  
  
char *Source_String = "Learn C Online is a great site";  
  
strncat(Destination_String, Source_String,14);  
  
puts( Destination_String);
```

Output:

Visit Learn C Online

strcmp()

strcmp() function is used to compare two strings. strcmp() function does a case-sensitive comparison between two strings. The Destination_String and Source_String can either be a string constant or a variable.

Syntax:

```
int strcmp(string1, string2);
```

This function returns an integer value after comparison. The value returned is 0 if two strings are equal.

If the first string is alphabetically greater than the second string then, it returns a positive value.

If the first string is alphabetically less than the second string then, it returns a negative value

Example:

```
char *string1 = "C Programming";  
  
char *string2 = "C Programming";  
  
int ret;  
  
ret=strcmp(string1, string2);  
  
printf("%d",ret);
```

Output:

0

strncmp()

strncmp() is used to compare only left most 'n' characters from the strings.

Syntax:

```
int strncmp(string1, string2, no_of_chars);
```

This function returns an integer value after comparison.

The value returned is 0 if left most 'n' characters of two strings are equal.

If the left-most 'n' characters of the first string are alphabetically greater than the left-most 'n' characters of second-string then, it returns a positive value.

If the left-most 'n' characters of the first string are alphabetically less than the left-most 'n' characters of second-string then, it returns a negative value

Example:

```
char *string1 = "Learn C Online is a great site";
```

```
char *string2 = "Learn C Online";
```

```
int ret;
```

```
ret=strncmp(string1, string2,7);
```

```
printf("%d",ret);
```

Output: 0

strcmpi()

strcmpi() function is used to compare two strings. strcmp() function does a case insensitive comparison between two strings. The Destination_String and Source_String can either be a string constant or a variable.

Syntax:

```
int strcmpi(string1, string2);
```

This function returns an integer value after comparison.

Example:

```
char *string1 = "Learn C Online";
```

```
char *string2 = "LEARN C ONLINE";
```

```
int ret;
```

```
ret=strcmpi(string1, string2);  
printf("%d",ret);
```

Output:

0

strncmpi()

strncmpi() is used to compare only left most 'n' characters from the strings. strncmpi() function does a case insensitive comparison.

Syntax:

```
int strncmpi(string1, string2,no_of_chars);
```

This function returns an integer value after comparison.

Example:

```
char *string1 = "Learn C Online is a great site";  
char *string2 = "LEARN C ONLINE";  
int ret;  
ret=strncmpi(string1, string2,7);  
printf("%d",ret);
```

Output:

0

2.17 CONTROL STRUCTURE

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program, that is called control statement. Control statement defined how the control is transferred from one part to the other part of the program. There are several control statement like if...else, switch, while, do...while, for loop, break, continue, goto etc.

2.17.1 COMPOUND STATEMENT

A compound statement (also called a "block") typically appears as the body of another statement, such as the if statement. Declarations and Types describes the form and meaning of the declarations that can appear at the head of a compound statement.

Syntax

compound-statement:

{ declaration-listopt statement-listopt }

declaration-list:

declaration

declaration-list declaration

statement-list:

statement

statement-list statement

If there are declarations, they must come before any statements. The scope of each identifier declared at the beginning of a compound statement extends from its declaration point to the end of the block. It is visible throughout the block unless a declaration of the same identifier exists in an inner block.

Identifiers in a compound statement are presumed auto unless explicitly declared otherwise with register, static, or extern, except functions, which can only be extern. You can leave off the extern specifier in function declarations and the function will still be extern.

Storage is not allocated and initialization is not permitted if a variable or function is declared in a compound statement with storage class extern. The declaration refers to an external variable or function defined elsewhere.

Variables declared in a block with the auto or register keyword are reallocated and, if necessary, initialized each time the compound statement is entered. These variables are not defined after the compound statement is exited. If a variable declared inside a block has the static attribute, the variable is initialized when program execution begins and keeps its value throughout the program. See Storage Classes for information about static.

This example illustrates a compound statement:

```
if ( i > 0 )  
{  
    line[i] = x;  
    x++;  
    i--;  
}
```

In this example, if i is greater than 0, all statements inside the compound statement are executed in order.

2.17.2 SELECTION STATEMENT

The selection statement means the statements are executed depends – upon a condition. If a condition is true, a true block (a set of statements) is executed, otherwise a false block is executed. This statement is also called decision statement or selection statement because it helps in making decision about which set of statements are to be executed.

Types: Two way branching, Multiway branching

2.17.2.1 IF STATEMENT

Statement execute set of command like when condition is true and its syntax is

if (condition)

Statement;

The statement is executed only when condition is true. If the if statement body is consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

void main()

{

int n;

printf (“ enter a number:”);

scanf(“%d”,&n);

if (n>10)

printf(“ number is greater”);

}

Output:

Enter a number:12

Number is greater

2.17.2.2 IF – ELSE STATEMENT

It is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded as true & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

if (condition)

```

{
Statement1;
Statement2;
}
else
{
Statement1;
Statement2;
}

```

Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be a if statement with in an else statement.

Example:- /* To check a number is even or odd */

```

void main()
{
int n;
printf ("enter a number: ");
scanf ("%d", &n);
if (n%2==0)
printf ("even number");
else
printf ("odd number");
}

```

Output:

Enter a number:121

odd number

2.17.2.3 NESTED IF

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```
if (condition)
{
    if (condition)
        Statement1;
    else
        statement2;
}
else
    Statement3;
```

If....else LADDER

In this type of nesting there is an if else statement in every else part except the last part. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is :-

```
if (condition)
    Statement1;
else if (condition)
    statement2;
else if (condition)
    statement3;
else
    statement4;
```

This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested “else if” would not executed. But it has disadvantage over if else statement that, in if else statement whenever the condition is true, other condition are not checked. While in this case, all condition are checked.

2.17.2.4 SWITCH

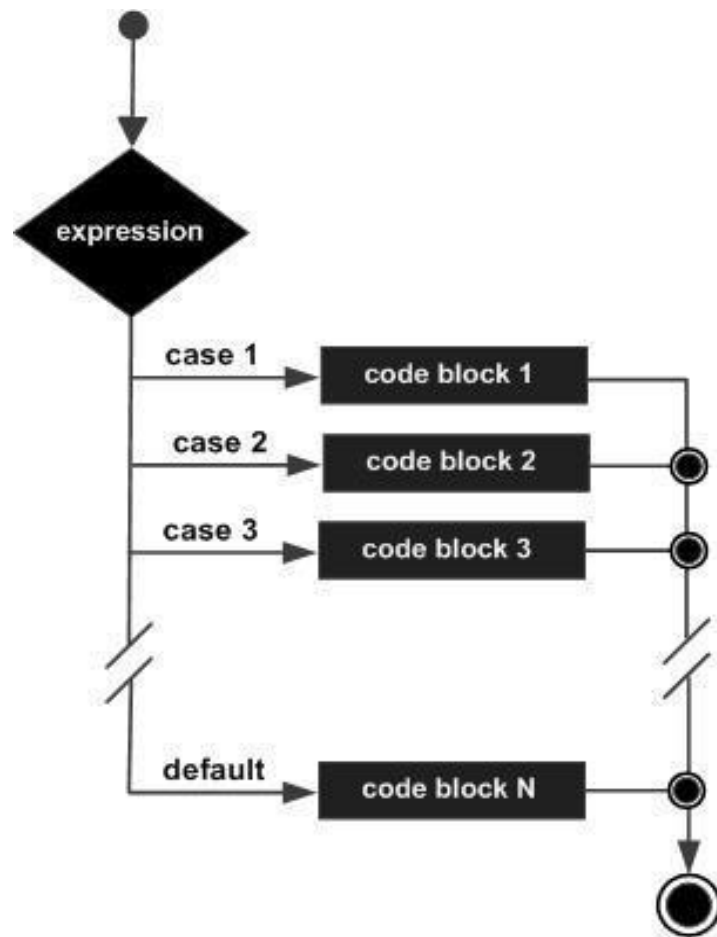
A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Syntax

```
switch(expression) {  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    /* you can have any number of case statements */  
    default : /* Optional */  
        statement(s);  
}
```

2.18 ITERATION STATEMENT

Loop:-it is a block of statement that performs set of instructions. In loops

Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

2.18.1 FOR LOOP

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)
```

```
{
```

```
Statement;
```

```
}
```

Or

```
for(initialized counter; test counter; update counter)
{
    Statement;
}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Example:-

```
void main()
{
    int i;
    for(i=1;i<10;i++)
    {
        printf(" %d ", i);
    }
}
```

Output:-1 2 3 4 5 6 7 8 9

2.18.2 WHILE LOOP

The test condition may be any expression .when we want to do something a fixed no of times but not known about the number of iteration, in a program then while loop is used.

Here first condition is checked if, it is true body of the loop is executed else, If condition is false control will be come out of loop.

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

Example:

Syntax:

while(condition)

{

Statement 1;

Statement 2;

}

Or

while(test condition)

Statement;

```
/* wap to print 5 times welcome to C */
#include<stdio.h>

void main()
{
    int p=1;
    While(p<=5)
    {
        printf("Welcome to C\n");
        P=p+1;
    }
}
```

Output: Welcome to C

Welcome to C

Welcome to C

Welcome to C

Welcome to C

2.18.3 DO WHILE LOOP

This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Example:-

#include<stdio.h>

void main()

{

int X=4;

do

Syntax:-

Do

{

Statement;

}

while(condition);

```

{
printf("%d",X);

X=X+1;

}while(X<=10);

Printf(" ");

}

```

Output: 4 5 6 7 8 9 10

Here firstly statement inside body is executed then condition is checked. If the condition is true again body of loop is executed and this process continue until the condition becomes false. Unlike while loop, semicolon is placed at the end of while.

There is minor difference between while and do while loop, while loop test the condition before executing any of the statement of loop. Whereas do while loop test condition after having executed the statement at least one within the loop.

If initial condition is false while loop would not executed it's statement on other hand do while loop executed it's statement at least once even If condition fails for first time. It means do while loop always executes at least once.

Notes:

Do while loop used rarely when we want to execute a loop at least once.

2.18.4 NESTED LOOPS

When a loop is written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

```

void main()

{

int i,j;

for(i=0;i<2;i++)

for(j=0;j<5; j++)

printf("%d %d", i, j);

}

```

Output: i=0

j=0 1 2 3 4

i=1

j=0 1 2 3 4

2.18.5 JUMP STATEMENTS

Jump Statements interrupt the normal flow of the program while execution and jump when it gets satisfied given specific conditions. The main uses of jump statements are to exit the loops like for, while, do-while also switch case and executes the given or next block of the code, skip the iterations of the loop, change the control flow to specific location, etc. There are 4 types of Jump statements in C language. – Break Statement, Continue Statement, Goto Statement, Return Statement.

2.18.5.1 BREAK

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in which it appears and it is generally written with condition. It is written with the keyword as break. When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to the first statement after the loop. This break statement is usually associated with if statement.

Example :

```
void main()
{
    int j=0;
    for(;j<6;j++)
        if(j==4)
            break;
}
```

Output:

0 1 2 3

2.18.5.2 CONTINUE

Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through the beginning of the loop. It is usually associated with the if statement. It is useful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

Example:-

```
void main()
{
    int n;
    for(n=2; n<=9; n++)
    {
        if(n==4)
            continue;
        printf("%d", n);
    }
    printf("out of loop");
}
```

Output: 2 3 5 6 7 8 9 out of loop

2.18.5.3 GOTO

The goto statement allows us to transfer control of the program to the specified label.

Syntax of goto Statement

```
goto label;
```

... ..

label:

statement;

The label is an identifier. When the goto statement is encountered, the control of the program jumps to label: and starts executing the code.

Example: goto Statement

```
// Program to calculate the sum and average of positive numbers
```

```
// If the user enters a negative number, the sum and average are displayed.
```

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

```
int main() {
```

```
    const int maxInput = 100;
```

```
    int i;
```

```
    double number, average, sum = 0.0;
```

```
    for (i = 1; i <= maxInput; ++i) {
```

```
        printf("%d. Enter a number: ", i);
```

```
        scanf("%lf", &number);
```

```
        // go to jump if the user enters a negative number
```

```
        if (number < 0.0) {
```

```
            goto jump;
```

```
        }
```

```
        sum += number;
```

```
    }
```

```
jump:
```

```
    average = sum / (i - 1);
```

```
    printf("Sum = %.2f\n", sum);
```

```
    printf("Average = %.2f", average);
```

```
    return 0;
```

```
}
```

Run Code

Output

1. Enter a number: 3

2. Enter a number: 4.3

3. Enter a number: 9.3

4. Enter a number: -2.9

Sum = 16.60

Average = 5.53

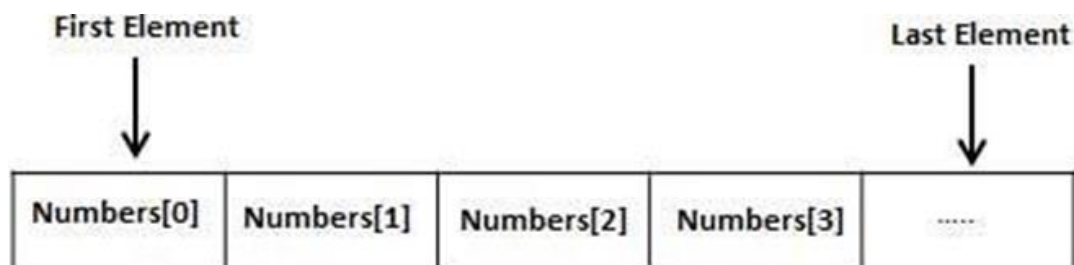
UNIT III

3.1 ARRAYS

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



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];
```

Now balance is a variable array which is sufficient to hold up-to 10 double numbers.

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 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```

#include <stdio.h>

int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */ int i,j;

    /* initialize elements of array n to 0 */ for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */ for ( j = 0; j < 10; j++ )
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}

```

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

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

3.1.1 NEED

Arrays are very much useful in a case where many variables of the same (data) types need to be stored and processed. For example, suppose we have to write a program in which can accept salary of 50 employees. If we solve this problem by making use of variables, we need 50 variables to store employee's salary. Remembering and managing these 50 variables is not an easy task and it will make the program a complex and lengthy program. This problem can be solved by declaring 1 array having 50 elements; one for employee's salary. Now we only have to remember the name of that 1 array.

3.1.2 TYPES

3.1.2.1 SINGLE DIMENSIONAL ARRAYS

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

```
double balance[5] = { 1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets []. Following is an example to assign a single element of the array:

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, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th i.e. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

3.1.2.2 TWO DIMENSIONAL ARRAYS

The simplest form of the 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 think as a table which will have x number of rows and y number of columns. A 2-dimentional array **a**, which contains three rows and four columns can be shown as below:

	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 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 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 2-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 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```

#include <stdio.h>

int main ()
{
    /* an array with 5 rows and 2 columns*/

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

    int i, j;

    /* output each array element's value */for ( i = 0; i
    < 5; i++ )
    {
        for ( j = 0; j < 2; j++ )
        {
            printf("a[%d][%d] = %d\n", i,j, a[i][j] );
        }
    }

    return 0;
}

```

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

```

a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6

```

3.1.3 STRINGS

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

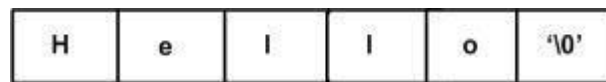
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

```
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 above-defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print 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 result something as follows:

```
Greeting message: Hello
```

3.1.3.1 STRING MANIPULATION

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

S.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.

2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <stdio.h>

#include <string.h>

int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

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

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

    /* total length of str1 after concatenation */
```

```

len = strlen(str1);

printf("strlen(str1) : %d\n", len );

return 0;

}

```

When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) : Hello
```

3.1.3.2 ARRAYS OF STRINGS

A String can be defined as a one-dimensional array of characters, so an array of strings is two –dimensional array of characters.

Syntax

```
char str_name[size][max];
```

Syntax

```
char str_arr[2][6] = { {'g','o','u','r','i','\0'}, {'r','a','m','\0'} };
```

Alternatively, we can even declare it as

Syntax

```
char str_arr[2][6]={"gouri", "ram"};
```

From the given syntax there are two subscripts first one is for how many strings to declare and the second is to define the maximum length of characters that each string can store including the null character. C concept already explains that each character takes 1 byte of data while allocating memory, the above example of syntax occupies $2 * 6 = 12$ bytes of memory.

Example

```
char str_name[8] = {'s','t','r','i','n','g','s','\0'};
```

By the rule of initialization of array, the above declaration can be written as

```
char str_name[] = "Strings";
```

0 1 2 3 4 5 6 7 Index

Variables 2000 2001 2002 2003 2004 2005 2006 2007 Address

This is a representation of how strings are allocated in memory for the above-declared string in C.

Following are the examples:

Example:

```
#include <stdio.h>

int main()
{
    char name[10];

    printf("Enter the name: ");

    fgets(name, sizeof(name), stdin);

    printf("Name is : ");

    puts(name);

    return 0;
}
```

Output:

Now for two-dimensional arrays, we have the following syntax and memory allocation. For this, we can take it as row and column representation (table format).

```
char str_name[size][max];
```

In this table representation, each row (first subscript) defines as the number of strings to be stored and column (second subscript) defines the maximum length of the strings.

```
char str_arr[2][6] = { {'g','o','u','r','i','\0'}, {'r','a','m','\0'} };
```

Alternatively, we can even declare it as

Syntax:

```
char str_arr[2][8]={"gouri", "ram"};
```

3.1.3.3 EVALUATION ORDER

Given an array arr[] of string type which consists of strings “+”, “-” and Numbers. Find the sum of the given array.

Examples :

Input : arr[] = {"3", "+", "4", "-", "7", "+", "13"}

Output : Value = 13

The value of expression 3+4-7+13 is 13.

Input : arr[] = { "2", "+", "1", "-8", "+", "13" }

Output : Value = 8

3.2 FUNCTION

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 usually is so **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, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like 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 language consists of a function header and a function body. Here are all the parts of a function:

Return Type: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or

argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */  
  
int max(int num1, int num2)  
{  
    /* local variable declaration */  
  
    int result;  
  
    if (num1 > num2) result  
        = num1;  
  
    else  
  
        result = num2;  
  
    return result;  
}
```

3.2.1 FUNCTION COMPONENTS

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(), following is the function declaration:

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

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <stdio.h>

/* function declaration */

int max(int num1, int num2);

int main ()
{
    /* local variable definition */int a = 100;
    int b = 200;int ret;

    /* calling a function to get max value */ret = max(a,
    b);

    printf( "Max value is : %d\n", ret );return 0;
}

/* function returning the max between two numbers */

int max(int num1, int num2)
{
    /* local variable declaration */

    int result;

    if (num1 > num2) result
        = num1;
```

```

else

    result = num2;

return result;

}

```

The max() function is left along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are **two ways** that arguments can be passed to a function:

Function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses call by value method to pass arguments. In general, this means that **code within a function cannot alter the arguments used to call the function**. Consider the function swap() definition as follows.

```

/* function definition to swap the values */

void swap(int x, int y)

{

    int temp;

    temp = x; /* save the value of x */

    x = y;    /* put y into x */

    y = temp; /* put x into y */

    return;

}

```

Now, let us call the function swap() by passing actual values as in the following example:

```
#include <stdio.h>

/* function declaration */

void swap(int x, int y);

int main ()
{
    /* local variable definition */

    int a = 100;

    int b = 200;

    printf("Before swap, value of a : %d\n", a );

    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */

    swap(a, b);

    printf("After swap, value of a : %d\n", a );

    printf("After swap, value of b : %d\n", b );

    return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

Function call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes

made to the parameter affect the passed argument.

To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as **pointer types** as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
/* function definition to swap the values */  
  
void swap(int *x, int *y)  
{  
    int temp;  
  
    temp = *x;    /* save the value at address x */  
  
    *x = *y;      /* put y into x */  
  
    *y = temp;    /* put x into y */  
  
    return;  
}
```

Let us call the function swap() by passing values by reference as in the following example:

```
#include <stdio.h>  
  
/* function declaration */  
  
void swap(int *x, int *y);  
  
int main ()  
{  
  
    /* local variable definition */  
  
    int a = 100;  
  
    int b = 200;  
  
    printf("Before swap, value of a : %d\n", a );  
  
    printf("Before swap, value of b : %d\n", b );  
  
    /* calling a function to swap the values.  
  
    * &a indicates pointer to a ie. address of variable a and
```

```

    * &b indicates pointer to b ie. address of variable b.

    */

    swap(&a, &b);

    printf("After swap, value of a : %d\n", a );

    printf("After swap, value of b : %d\n", b );

    return 0;

}

```

Result:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

3.2.2 RETURN DATA TYPE

The return statement returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Syntax:

```
return[expression];
```

There are various ways to use return statements. Few are mentioned below:

Methods not returning a value: In C/C++ one cannot skip the return statement, when the methods are of return type. The return statement can be skipped only for void types.

Not using return statement in void return type function: When a function does not return anything, the void return type is used. So if there is a void return type in the function definition, then there will be no return statement inside that function (generally).

Syntax:

```
void func()

{

    .

    .

}
```

Example:

```
// C code to show not using return

// statement in void return type function

#include <stdio.h>

// void method

void Print()

{

    printf("Welcome to C Programming");

}

// Driver method

int main()

{

    // Calling print

    print();

    return 0;

}
```

Output:

Welcome to C Programming

Using return statement in void return type function: Now the question arises, what if there is a return statement inside a void return type function? Since we know that, if there is a void return type in the function definition, then there will be no return statement inside

that function. But if there is a return statement inside it, then also there will be no problem if the syntax of it will be:

Correct Syntax:

```
void func()
{
    return;
}
```

This syntax is used in function just as a jump statement in order to break the flow of the function and jump out of it. One can think of it as an alternative to “break statement” to use in functions.

3.2.3 PARAMETER PASSING

Function parameters, so called **formal parameters**, are treated as local variables within that function and they will take preference over the global variables.

Following is an example:

```
#include <stdio.h>

/* global variable declaration */

int a = 20;

int main ()

    /* local variable declaration in main function */

    int a = 10;

    int b = 20;

    int c = 0;

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

    printf ("value of c in main() = %d\n",

        return 0;

}

/* function to add two integers */

int sum(int a, int b)

{
```

```

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

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

return a + b;
}

```

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

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

3.2.4 RETURN BY REFERENCE

Functions can be declared to return a reference type. There are two reasons to make such a declaration:

The information being returned is a large enough object that returning a reference is more efficient than returning a copy.

The type of the function must be an l-value.

The referred-to object will not go out of scope when the function returns.

Just as it can be more efficient to pass large objects to functions by reference, it also can be more efficient to return large objects from functions by reference. Reference-return protocol eliminates the necessity of copying the object to a temporary location prior to returning.

Reference-return types can also be useful when the function must evaluate to an l-value. Most overloaded operators fall into this category, particularly the assignment operator. Overloaded operators are covered in Overloaded Operators.

3.2.5 DEFAULT ARGUMENTS

We must follow four rules when using default arguments. Some rules may seem difficult at first, but after a few simple examples, you will see that they are pretty easy to understand and remember.

Arguments without a default value may not be defined to the right of an argument with a default.

When calling a function with default arguments, an argument may not be specified to the right of an argument whose default is accepted.

If the function definition and the function declaration are not the same, then the default values appear in function prototypes (perhaps in a .h file).

If a function with default arguments is also overloaded, all possible ways that the function can be called with and without default values must be distinct from any and all overloaded versions.

3.2.6 RECURSIVE FUNCTIONS

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

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

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

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

3.2.7 ARRAYS WITH FUNCTIONS

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

Example 1: Pass Individual Array Elements

```
#include <stdio.h>  
  
void display(int age1, int age2) {  
    printf("%d\n", age1);  
    printf("%d\n", age2);  
}  
  
int main() {  
    int ageArray[] = {2, 8, 4, 12};
```

```
// pass second and third elements to display()

display(ageArray[1], ageArray[2]);

return 0;

}
```

Output

8

4

Here, we have passed array parameters to the display() function in the same way we pass variables to a function.

```
// pass second and third elements to display()

display(ageArray[1], ageArray[2]);
```

We can see this in the function definition, where the function parameters are individual variables:

```
void display(int age1, int age2) {

    // code

}
```

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

```
result = calculateSum(num);
```

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

```
float calculateSum(float num[]) {

    ... ..

}
```

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

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

Example 3: Pass two-dimensional arrays

```
#include <stdio.h>

void displayNumbers(int num[2][2]);
```

```

int main() {
    int num[2][2];

    printf("Enter 4 numbers:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            scanf("%d", &num[i][j]);
        }
    }

    // pass multi-dimensional array to a function
    displayNumbers(num);

    return 0;
}

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

```

Output

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

Notice the parameter `int num[2][2]` in the function prototype and function definition:

```
// function prototype
```

```
void displayNumbers (int num[2][2]);
```

This signifies that the function takes a two-dimensional array as an argument. We can also pass arrays with more than 2 dimensions as a function argument.

When passing two-dimensional arrays, it is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified.

For example,

```
void displayNumbers (int num[][2]) {
```

```
    // code
```

```
}
```

3.2.8 STORAGE CLASSES

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:

1. **auto:** This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword `auto` is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the `auto` variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside. They are assigned a garbage value by default whenever they are declared.

Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

- extern:** Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function / block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this link.
- static:** This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.
- register:** This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free registration is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free registration is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the

program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:
Syntax:

```
storage_class var_data_type var_name;
```

Functions follow the same syntax as given above for variables. Have a look at the following C example for further clarification:

```
// A C program to demonstrate different storage classes
```

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

```
// declaring the variable which is to be made extern
```

```
// an initial value can also be initialized to x
```

```
int x;
```

```
void autoStorageClass()
```

```
{
```

```
    printf("\nDemonstrating auto class\n\n");
```

```
    // declaring an auto variable (simply
```

```
    // writing "int a=32;" works as well)
```

```
    auto int a = 32;
```

```
    // printing the auto variable 'a'
```

```
    printf("Value of the variable 'a'"
```

```
        " declared as auto: %d\n",
```

```
        a);
```

```
    printf("-----");
```

```
}
```

```
void registerStorageClass()
```

```
{
```

```
    printf("\nDemonstrating register class\n\n");
```

```

// declaring a register variable
register char b = 'G';

// printing the register variable 'b'
printf("Value of the variable 'b'"
      " declared as register: %d\n",
      b);
printf("-----");
}

void externStorageClass()
{
    printf("\nDemonstrating extern class\n\n");

    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;

    // printing the extern variables 'x'
    printf("Value of the variable 'x'"
          " declared as extern: %d\n",
          x);

    // value of extern variable x modified
    x = 2;

    // printing the modified values of
    // extern variables 'x'
    printf("Modified value of the variable 'x'"
          " declared as extern: %d\n",

```

```

        x);

printf("-----");
}

void staticStorageClass()
{
    int i = 0;

    printf("\nDemonstrating static class\n\n");

    // using a static variable 'y'

    printf("Declaring 'y' as static inside the loop.\n"
           "But this declaration will occur only"
           " once as 'y' is static.\n"
           "If not, then every time the value of 'y' "
           "will be the declared value 5"
           " as in the case of variable 'p'\n");

    printf("\nLoop started:\n");

    for (i = 1; i < 5; i++) {

        // Declaring the static variable 'y'

        static int y = 5;

        // Declare a non-static variable 'p'

        int p = 10;

        // Incrementing the value of y and p by 1

        y++;

        p++;

        // printing value of y at each iteration

        printf("\nThe value of 'y', "
               "declared as static, in %d "

```

```

        "iteration is %d\n",
        i, y);
// printing value of p at each iteration
printf("The value of non-static variable 'p', "
        "in %d iteration is %d\n",
        i, p);
}
printf("\nLoop ended:\n");
printf("-----");
}
int main()
{
    printf("A program to demonstrate"
           " Storage Classes in C\n\n");
// To demonstrate auto Storage Class
autoStorageClass();
// To demonstrate register Storage Class
registerStorageClass();
// To demonstrate extern Storage Class
externStorageClass();
// To demonstrate static Storage Class
staticStorageClass();
// exiting
printf("\n\nStorage Classes demonstrated");
return 0;
}

```

Output:

A program to demonstrate Storage Classes in C

Demonstrating auto class

Value of the variable 'a' declared as auto: 32

Demonstrating register class

Value of the variable 'b' declared as register: 71

Demonstrating extern class

Value of the variable 'x' declared as extern: 0

Modified value of the variable 'x' declared as extern: 2

Demonstrating static class

Declaring 'y' as static inside the loop.

But this declaration will occur only once as 'y' is static.

If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'

Loop started:

The value of 'y', declared as static, in 1 iteration is 6

The value of non-static variable 'p', in 1 iteration is 11

The value of 'y', declared as static, in 2 iterations is 7

The value of non-static variable 'p', in 2 iterations is 11

The value of 'y', declared as static, in 3 iterations is 8

The value of non-static variable 'p', in 3 iterations is 11

The value of 'y', declared as static, in 4 iterations is 9

The value of non-static variable 'p', in 4 iterations is 11

Loop ended:

Storage Classes demonstrated

UNIT IV

4.1 STRUCTURE

C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

Title

Author

Subject

Book ID

4.1.1 DECLARATION

'struct' keyword is used to create a structure. Following is an example.

```
struct address
```

```
{  
    char name[50];  
    char street[100];  
    char city[50];  
    char state[20];  
    int pin;  
};
```

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
```

```
struct Point
```

```
{  
    int x, y;  
} p1; // The variable p1 is declared with 'Point'
```

```
// A variable declaration like basic data types

struct Point

{
    int x, y;
};

int main()

{
    struct Point p1; // The variable p1 is declared like a normal variable
}
```

4.1.2 DEFINITION

To define a structure, you **must use the struct statement**. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

```
struct [structure tag]

{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as

int i; or float f; or any other valid variable definition. At the end of the structure's definition, before

the final semicolon, you can specify one or more structure variables but it is optional. Here is the

way you would declare the Book structure:

```
struct Books
```

```

{
char title[50];

char author[50];

char subject[100];

int book_id;

} book;

```

4.1.3 ACCESSING STRUCTURE MEMBERS

To access any member of a structure, we use the **member access operator** . . The **member**

access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure

type. Following is the example to explain usage of structure:

```

#include <stdio.h>

#include <string.h>

struct Books

{

char title[50];

char author[50];

char subject[100];

int book_id;

};

int main( )

{ struct Books Book1; /* Declare Book1 of type Book */

struct Books Book2; /* Declare Book2 of type Book */

/* book 1 specification */

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

```



```

strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* book 2 specification */

strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */

printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* print Book2 info */

printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}

```

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

```

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial

```

4.1.4 INITIALIZATION

Structure members cannot be initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
    int x = 0; // COMPILER ERROR: cannot initialize members here
    int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members can be initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
    int x, y;
};

int main()
{
    // A valid initialization. member x gets value 0 and y
    // gets value 1. The order of declaration is followed.
    struct Point p1 = {0, 1};
}
```

4.1.5 NESTING OF STRUCTURES

C provides us the feature of nesting one structure within another structure by using which, complex data types are created.

For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee

into a separate structure and nest the structure address into the structure employee. Consider the following program.

```
#include<stdio.h>

struct address
{
    char city[20];
    int pin;
    char phone[14];
};

struct employee
{
    char name[20];
    struct address add;
};

void main ()
{
    struct employee emp;

    printf("Enter employee information?\n");

    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);

    printf("Printing the employee information....\n");

    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

The structure can be nested in the following ways.

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
    int dd;
    int mm;
    int yyyy;
};

struct Employee
{
    int id;
    char name[20];
    struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it cannot be used in multiple data structures. Consider the following example.

```

struct Employee
{
    int id;
    char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}empl;

```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy

C Nested Structure example

Let's see a simple example of the nested structure in C language.

```

#include <stdio.h>

#include <string.h>

struct Employee
{
    int id;
    char name[20];
    struct Date
    {

```

```

        int dd;

        int mm;

        int yyyy;

    }doj;
}e1;

int main( )
{
    //storing employee information

    e1.id=101;

    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array

    e1.doj.dd=10;

    e1.doj.mm=11;

    e1.doj.yyyy=2014;

    //printing first employee information

    printf( "employee id : %d\n", e1.id);

    printf( "employee name : %s\n", e1.name);

    printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.m
m,e1.doj.yyyy);

    return 0;

}

```

Output:

employee id : 101

employee name : Sonoo Jaiswal

employee date of joining (dd/mm/yyyy) : 10/11/2014

4.2 UNION

4.2.1 INTRODUCTION

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be

defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but **unlike structures, they share the same memory location.**

Let's understand this through an example.

```
struct abc
```

```
{  
  
    int a;  
  
    char b;  
  
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type int and 'b' of type character. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the **difference is that union keyword is used** for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

4.2.2 DIFFERENCE BETWEEN STRUCTURE AND UNION

	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.

4.3 POINTERS

4.3.1 INTRODUCTION

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.

```
int n = 10;
```

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

Declaring a pointer

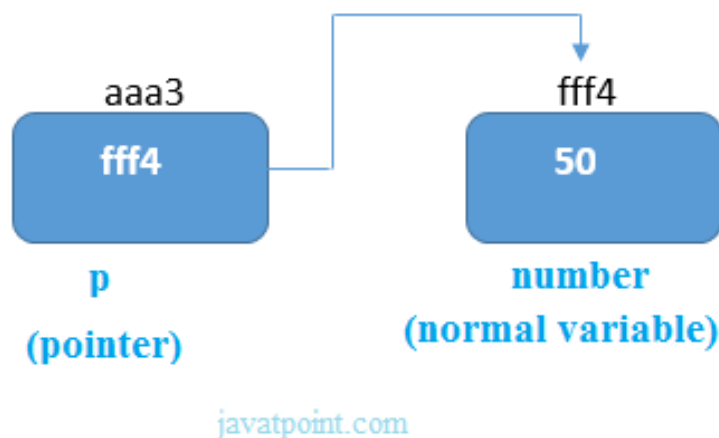
The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

```
int *a; // pointer to int
```

```
char *c; // pointer to char
```

Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (indirection operator), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.


```
#include<stdio.h>

int main(){

int number=50;

int *p;

p=&number;//stores the address of number variable

printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.

printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.

return 0;

}
```

Output

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

Pointer to array

```
int arr[10];

int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.
```

Pointer to a function

```
void show (int);

void(*p)(int) = &display; // Pointer p is pointing to the address of a function
```

Pointer to structure

```
struct st {

    int i;

    float f;

}ref;

struct st *p = &ref;
```

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.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

4.3.2 ADDRESS OPERATOR

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

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

```
int main(){
```

```
int number=50;
```

```
printf("value of number is %d, address of number is %u",number,&number);
```

```
return 0;
```

```
}
```

Output

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero)

Address Operator(&)

The Address Operator in C also called a pointer. This address operator is denoted by “&”. This & symbol is called an ampersand. This & is used in a unary operator. The purpose of this address operator or pointer is used to return the address of the variable. Once we declared a pointer variable, we have to initialize the pointer with a valid memory address; to get the memory address of the variable ampersand is used. When we use the ampersand symbol as a prefix to the variable name & and it gives the address of that variable. An address of the operator is used within C that is returned to the memory address of a variable. These addresses returned by the address of the operator are known as pointers because they “point” to the variable in memory.

4.3.3 POINTER VARIABLES

Pointer Variables

A computer memory location has an address and holds a content. The address is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data. It is entirely up to the programmer to interpret the meaning of the data, such as integer, real number, characters or strings. To ease the burden of programming using numerical address and programmer-interpreted data, early programming languages (such as C) introduce the concept of variables. A variable is a named location that can store a value of a particular type. Instead of numerical addresses, names (or identifiers) are attached to certain addresses. Also, types (such as int, double, char) are associated with the contents for ease of interpretation of data. Each address location typically hold 8-bit (i.e., 1-byte) of data. A 4-byte int value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses. To store a 32-bit address, 4 memory locations are required.

The image below shows how to assign a pointer variable.

```
int *p;
```

```
p = &a;
```

4.3.4 VOID POINTERS

Void pointers

The void pointers refer to the pointers that have no data type associated with them. As the name suggests, the void pointer indicates that the pointer is basically empty- and thus, it is capable of holding any data type address in the program. Then, these void pointers that have addresses, can be further typecast into other data types very easily.

The allocation of memory also becomes very easy with such void pointers. It is because they make the functions flexible enough to be allocated with bytes and memories appropriately. Let us look at the syntax for the void pointer in C.

Syntax

```
void *name_of_pointer;
```

Here, the void keyword acts as the pointer type, and it is followed by the pointer name- to which the pointer type points and allocates the address location in the code. The declaration of a pointer happens with the name and type of pointer that supports any given data type. Let us take a look at an example to understand this better.

```
void *ptr
```

Here, the pointer is expecting a void type- not int, float, etc. It is pointed by the ptr pointer here that includes the * symbol. It denotes that this pointer has been declared, and it will be used for dereferencing in the future.

Why use a Void Pointer in C?

As we all know, the address that is assigned to any pointer must have the same data type as we have specified in the declaration of the pointer. For instance, when we are declaring the float pointer, this float pointer won't be able to point to an int variable or any other variable. In simpler words, it will only be able to point to the float type variable. We thus use a pointer to void to overcome this very problem.

The pointer to void can be used in generic functions in C because it is capable of pointing to any data type. One can assign the void pointer with any data type's address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting. So, it reduces complications in a code.

The Basic Algorithm

Here is how the void pointers work in a program. Consider this as the pseudocode for the program discussed ahead.

Begin

Declaring v of int data type.

Initializing v = 7.

Declaring w of float data type.

Initializing w = 7.6.

Declaring the pointer u as void.

Initializing the u pointer to v.

Print “The Integer variable in the program is equal to = ”.

Printing the value of v using the pointer u.

Initializing the u pointer to w.

Print “The Float variable in the program is equal to = ”.

Printing the value of w using the pointer u.

End.

Example Code

Let us take a look at an example of how we use the above void pointers in a code.

```
#include<stdlib.h>

int main() {

int v = 7;

float w = 7.6;

void *u;

u = &v;

printf(“The Integer variable in the program is equal to = %d”, *( (int*) u) );

u = &w;

printf(“\nThe Float variable in the program is equal to = %f”, *( (float*) u) );

return 0;

}
```

The output obtained out of the program would be:

The Integer variable in the program is equal to = 7

The Float variable in the program is equal to = 7.600000

4.3.5 POINTER ARITHMETIC

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

```
ptr++
```

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. **This operation will move the pointer to the next memory location without impacting the actual value at the memory location.** If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = {10, 100, 200};

    int i, *ptr;

    /* let us have array address in pointer */

    ptr = var;

    for ( i = 0; i < MAX; i++) {

        printf("Address of var[%d] = %x\n", i, ptr );

        printf("Value of var[%d] = %d\n", i, *ptr );

        /* move to the next location */

        ptr++;

    }

    return 0;

}
```

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

Address of var[0] = bf882b30

Value of var[0] = 10

Address of var[1] = bf882b34

Value of var[1] = 100

Address of var[2] = bf882b38

Value of var[2] = 200

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = { 10, 100, 200 };

    int i, *ptr;

    /* let us have array address in pointer */

    ptr = &var[MAX-1];

    for ( i = MAX; i > 0; i-- ) {

        printf("Address of var[%d] = %x\n", i-1, ptr );

        printf("Value of var[%d] = %d\n", i-1, *ptr );

        /* move to the previous location */

        ptr--;

    }

    return 0;}
```

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

Address of var[2] = bfedbcd8

Value of var[2] = 200

Address of var[1] = bfedbcd4

Value of var[1] = 100

Address of var[0] = bfedbcd0

Value of var[0] = 10

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example – one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] –

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = { 10, 100, 200 };

    int i, *ptr;

    /* let us have address of the first element in pointer */

    ptr = var;

    i = 0;

    while ( ptr <= &var[MAX - 1] ) {

        printf("Address of var[%d] = %x\n", i, ptr );

        printf("Value of var[%d] = %d\n", i, *ptr );

        /* point to the next location */

        ptr++;

        i++;

    }

    return 0;
```



```
}
```

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

Address of var[0] = bfdcb20

Value of var[0] = 10

Address of var[1] = bfdcb24

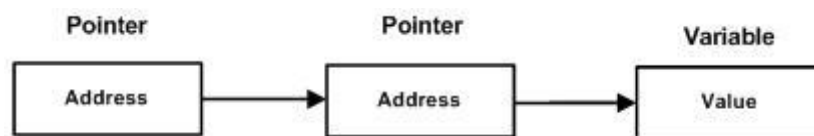
Value of var[1] = 100

Address of var[2] = bfdcb28

Value of var[2] = 200

4.3.6 POINTERS TO POINTERS

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int –

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

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

```
int main () {
```

```
    int var;
```

```
    int *ptr;
```

```
    int **pptr;
```

```
    var = 3000;
```

```
    /* take the address of var */
```

```
    ptr = &var;
```

```

/* take the address of ptr using address of operator & */

pptr = &ptr;

/* take the value using pptr */

printf("Value of var = %d\n", var );

printf("Value available at *ptr = %d\n", *ptr );

printf("Value available at **pptr = %d\n", **pptr);

return 0;}

```

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

Value of var = 3000

Value available at *ptr = 3000

Value available at **pptr = 3000

4.4 FILE HANDLING

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

4.4.1 HIERARCHY OF FILE STREAM CLASSES

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file `iostream.h`. Figure given below shows the hierarchy of these classes.

`ios` class is topmost class in the stream classes hierarchy. It is the base class for `istream`, `ostream`, and `streambuf` class.

`istream` and `ostream` serves the base classes for `iostream` class. The class `istream` is used for input and `ostream` for the output.

Class ios is indirectly inherited to iostream class using istream and ostream. To avoid the duplicity of data and member functions of ios class, it is declared as virtual base class when inheriting in istream and ostream as

```
class istream: virtual public ios
```

```
{  
};
```

```
class ostream: virtual public ios
```

```
{  
};
```

The _withassign classes are provided with extra functionality for the assignment operations that's why _withassign classes.

Facilities provided by these stream classes.

The ios class: The ios class is responsible for providing all input and output facilities to all other stream classes.

The istream class: This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as get, getline, read, ignore, putback etc..

4.4.2 OPENING & CLOSING A FILE - Modes

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

```
FILE *fopen( const char * filename, const char * mode );
```

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

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

```

void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}

```

Output

The content of the file will be printed.

```

#include;

void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;

```

```
printf("%c",ch);

}

fclose (fp );

}
```

Closing File: fclose()

The `fclose()` function is used to close a file. The file must be closed after performing all the operations on it. The syntax of `fclose()` function is given below:

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the `fclose()` function.

```
fclose(fptr);
```

Here, `fptr` is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprintf()` and `fscanf()` expects a pointer to the structure `FILE`.

Example 1: Write to a text file

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    int num;

    FILE *fptr;

    // use appropriate location if you are using MacOS or Linux

    fptr = fopen("C:\\program.txt","w");

    if(fptr == NULL)
    {
        printf("Error!");

        exit(1);
    }
}
```

```

}

printf("Enter num: ");

scanf("%d",&num);

fprintf(fptr,"%d",num);

fclose(fptr);

return 0;

}

```

This program takes a number from the user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

```

#include <stdio.h>

#include <stdlib.h>

int main()
{
    int num;

    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt","r")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.

        exit(1);
    }

    fscanf(fptr,"%d", &num);

    printf("Value of n=%d", num);

    fclose(fptr);

    return 0;

}

```


This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from Example 1, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in a similar way.

4.4.3 TESTING FOR ERRORS

It is quite common that errors may occur while reading data from a file in C++ or writing data to a file. For example, an error may arise due to the following:

- When trying to read a file beyond indicator.
- When trying to read a file that does not exist.
- When trying to use a file that has not been opened.
- When trying to use a file in an inappropriate mode i.e., writing data to a file that has been opened for reading.
- When writing to a file that is write-protected i.e., trying to write to a read-only file.

Failure to check for errors then the program may behave abnormally therefore an unchecked error may result in premature termination for the program or incorrect output.

Below are some [Error handling](#) functions during [file operations](#) in C/C++:

ferror():

In C/C++, the library function **ferror()** is used to check for the error in the stream. Its prototype is written as:

```
int ferror (FILE *stream);
```

The **ferror()** function checks for any error in the stream. It returns a value zero if no error has occurred and a non-zero value if there is an error. The error indication will last until the file is closed unless it is cleared by the **clearerr()** function.

Below is the program to implement the use of **ferror()**:

```
// C program to illustrate the
```

```
// use of ferror()
```

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

```
#include <stdlib.h>
```

```

// Driver Code

int main()
{
    FILE* fp;

    // If a file is opened which does
    // not exist, then it will be an
    // error and corresponding errno
    // value will be set

    char feedback[100];

    int i;

    fp = fopen("GeeksForGeeks.TXT", "w");
    if (fp == NULL) {
        printf("\n The file could "
            "not be opened");
        exit(1);
    }

    printf("\n Provide feedback on "
        "this article: ");

    fgets(feedback, 100, stdin);

    for (i = 0; i < feedback[i]; i++)
        fputc(feedback[i], fp);

    // Error writing file
    if (ferror(fp)) {
        printf("\n Error writing in file");
        exit(1);
    }
}

```

```

// Close the file pointer

fclose(fp);

}

```

After executing this code “Provide feedback on this article:“ will be displayed on the screen and after giving some feedback “Process exited after some seconds with return value 0” will be displayed on the screen.

clearerr():

The function **clearerr()** is used to clear the end-of-file and error indicators for the stream. Its prototype can be given as:

```
void clearerr(FILE *stream);
```

The **clearerr()** clears the error for the stream pointed by the stream. The function is used because error indicators are not automatically cleared. Once the error indicator for a specific stream is set, operations on the stream continue to return an error value until **clearerr()**, **fseek()**, **fsetpos()**, or **rewind()** is called.

Below is the program to implement the use of **clearerr()**:

```

// C program to illustrate the

// use of clearerr()

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

// Driver Code

int main()

{

    FILE* fp;

    char feedback[100];

    char c;

    fp = fopen("file.txt", "w");

```

```

c = fgetc(fp);

if (ferror(fp)) {
    printf("Error in reading from "
        " file : file.txt\n");
}

clearerr(fp)

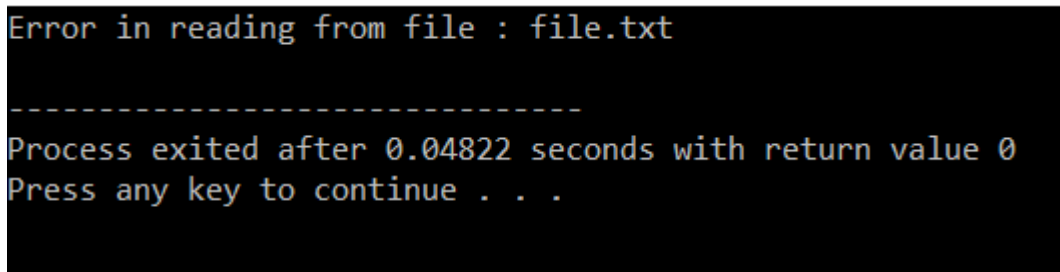
if (ferror(fp)) {
    printf("Error in reading from "
        "file : file.txt\n");
}

// close the file

fclose(fp);
}

```

Output:



```

Error in reading from file : file.txt
-----
Process exited after 0.04822 seconds with return value 0
Press any key to continue . . .

```

The function **perror()** stands for print error. In case of an error, the programmer can determine the type of error that has occurred using the **perror()** function.

When **perror()** is called, then it displays a message describing the most recent error that occurred during a library function call or system call. Its prototype can be given as:

```
void perror (char*msg);
```

- The **perror()** takes one argument which points to an optional user-defined message the message is printed first followed by a colon and the implementation-defined message that describes the most recent error.
- If a call to **perror()** is made when no error has actually occurred then 'No error' will be displayed.

- The most important thing to remember is that a call to **perror()** and nothing is done to deal with the error condition, then it is entirely up to the program to take action. For example, the program may prompt the user to do something such as terminate the program.
- Usually, the program's activities will be determined by checking the value of **errno** and the nature of the error.
- In order to use the external constant **errno**, you must include the header file **ERRNO.H**

Below is the program given below illustrates the use of **perror()**. Here, assume that the file "**file.txt**" does not exist.

```
// C program to illustrate the
// use of perror()

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

// Driver Code

int main()
{
    FILE* fp;

    // First rename if there is any file
    rename("file.txt", "newfile.txt");

    // Now try to open same file
    fp = fopen("file.txt", "r");

    if (fp == NULL) {
        perror("Error: ");
        return (-1);
    }

    // Close the file pointer
```

```

fclose(fp);

return (0);
}

```

Output:

```

Error: : No such file or directory

-----
Process exited after 0.1122 seconds with return value 4294967295
Press any key to continue . . .

```

4.4.4 FILE POINTERS AND THEIR MANIPULATION

- File pointer is a pointer which is used to handle and keep track on the files being accessed. A new data type called “FILE” is used to declare file pointer. This data type is defined in stdio.h file. File pointer is declared as FILE *fp. Where, ‘fp’ is a file pointer.
- fopen() function is used to open a file that returns a FILE pointer. Once file is opened, file pointer can be used to perform I/O operations on the file. fclose() function is used to close the file.

Example 1: Write to a text file

```

#include <stdio.h>

#include <stdlib.h>

int main()
{
    int num;

    FILE *fptr;

    // use appropriate location if you are using MacOS or Linux

    fptr = fopen("C:\\program.txt","w");

    if(fptr == NULL)
    {

```

```

    printf("Error!");
    exit(1);
}
printf("Enter num: ");
scanf("%d",&num);
fprintf(fptr,"%d",num);
fclose(fptr);
return 0;
}

```

This program takes a number from the user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num;
    FILE *fptr;
    if ((fptr = fopen("C:\\program.txt", "r")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    fscanf(fptr,"%d", &num);
    printf("Value of n=%d", num);
}

```

```

fclose(fp);

return 0;

}

```

This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from Example 1, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in a similar way.

4.4.5 SEQUENTIAL ACCESS

In this type of data is kept sequentially. If we want to read the last record of the file we need to read all the records before that record. It takes more time.

C imposes no file structure

- No notion of records in a file
- Programmer must provide file structure
- Creating a File
- FILE *cfPtr;
- Creates a FILE pointer called cfPtr
- cfPtr = fopen("clients.dat", "w");
- Function fopen returns a FILE pointer to file specified
- Takes two arguments – file to open and file open mode
- If open fails, NULL is returned

4.4.6 RANDOM ACCESS

Random accessing of files in C language can be done with the help of the following functions –

ftell ()

rewind ()

fseek ()

ftell ()

It returns the current position of the file ptr.

The syntax is as follows –

```
int n = ftell (file pointer)
```

For example,

```
FILE *fp;
```

```
int n;
```

```
_____
```

```
_____
```

```
_____
```

```
n = ftell (fp);
```

Note – ftell () is used for counting the number of characters which are entered into a file.

```
rewind ( )
```

It makes file ptr move to beginning of the file.

The syntax is as follows –

```
rewind (file pointer);
```

For example,

```
FILE *fp;
```

```
-----
```

```
-----
```

```
rewind (fp);
```

```
n = ftell (fp);
```

```
printf ("%d", n);
```

Output

The output is as follows –

0 (always).

```
fseek ( )
```

It is to make the file ptr point to a particular location in a file.

The syntax is as follows –

`fseek(file pointer, offset, position);`

Offset

The no of positions to be moved while reading or writing.

It can be either negative (or) positive.

Positive - forward direction.

Negative – backward direction.

Position

It can have three values, which are as follows –

0 – Beginning of the file.

1 – Current position.

2 – End of the file.

Example

`fseek (fp,0,2)` - fp moved 0 bytes forward from the end of the file.

`fseek (fp, 0, 0)` – fp moved 0 bytes forward from beginning of the file

`fseek (fp, m, 0)` – fp moved m bytes forward from the beginning of the file.

`fseek (fp, -m, 2)` – fp moved m bytes backward from the end of the file.

Errors

The errors related to `fseek ()` function are as follows –

`fseek (fp, -m, 0);`

`fseek(fp, +m, 2);`

4.4.7 COMMAND LINE ARGUMENTS

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using `main()` function arguments where `argc` refers to the number of arguments passed, and `argv[]` is a pointer array which points to

each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>

int main( int argc, char *argv[] ) {

    if( argc == 2 ) {

        printf("The argument supplied is %s\n", argv[1]);

    }

    else if( argc > 2 ) {

        printf("Too many arguments supplied.\n");

    }

    else {

        printf("One argument expected.\n");

    }

}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$/a.out testing
```

The argument supplied is testing

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$/a.out testing1 testing2
```

Too many arguments supplied.

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$/a.out
```

One argument expected

It should be noted that `argv[0]` holds the name of the program itself and `argv[1]` is a pointer to the first command line argument supplied, and `*argv[n]` is the last argument. If no arguments are supplied, `argc` will be one, and if you pass one argument then `argc` is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes '. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes –

```
#include <stdio.h>

int main( int argc, char *argv[] ) {

    printf("Program name %s\n", argv[0]);

    if( argc == 2 ) {

        printf("The argument supplied is %s\n", argv[1]);

    }

    else if( argc > 2 ) {

        printf("Too many arguments supplied.\n");

    }

    else {

        printf("One argument expected.\n");

    }

}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$/a.out "testing1 testing2"
```

Program name ./a.out

The argument supplied is testing1 testing2