

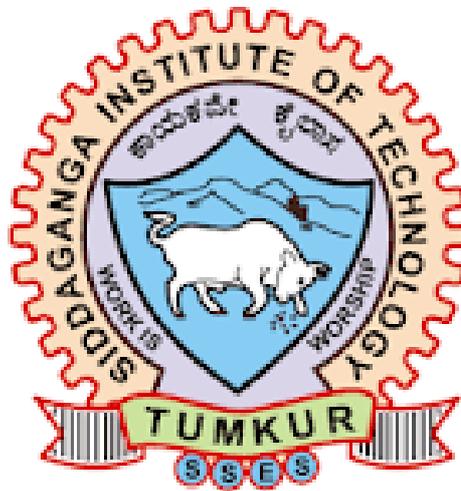
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lecture Notes

Course: C Programming for Problem Solving

Course Code: 1CPS

Faculty: Shwetha A N



SIDDAGANGA INSTITUTE OF TECHNOLOGY TUMKUR-3

An Autonomous Institution, Affiliated to VTU, Belagavi & Recognised by AICTE
and Accredited by NBA, New Delhi

UNIT 1

Introduction to digital computers

Computers have become the basic necessity for any organization. They are used for all sorts of problem solving ranging from simple addition to highly complex calculations like research, Engineering, Simulations, Weather forecasting etc.

A computer is an electronic device that is used for information processing. It accepts data and instructions, stores it in its memory, processes and gives the results to the user. The term compute is derived from Latin word 'Compute' which means to calculate. Therefore, the computer is a calculation machine.

The computers that we see and use in our day to day life are digital computers. They are called so because they use numbers to represent any piece of information. Digital computers work on the principle of programmed instructions which give the computers a purpose and instruct what to do.

Capabilities of a computer

A Computer is capable of processing the following tasks:

- 1] Huge data storage: A computer can store large amount of data and instructions in its memory.
- 2] Input and Output: A computer receives data and instructions as input and display output after execution.
- 3] Processing: It processes the data entered by the user. Processing means performing the necessary operations such as arithmetic or logical operations on the data.

Characteristics of computer

The basic characteristics of computer are

- 1] **High Processing speed:** A computer is an extremely fast information processing device. It carries out all sorts of computations within a fraction of a second. It executes millions of instructions per second.
- 2] **Accuracy:** It gives accurate result for correct input data. Here accuracy means the correctness of processed data. If the input data is erroneous, the output will not be correct.
- 3] **Reliability:** It always gives correct results.
- 4] **Versatility:** Computers are used in all fields like teaching, training, simulations, media and entertainment etc.
- 5] **Diligence:** It does not feel tired. It can be used for hours, days or months.

History of development of computers

1] **Abacus:** This is the first recorded computer. It was invented in china and used by Greeks, Romans and Japanese.

2] **Napier's Bones:** A mathematician John Napier introduced the concept of logarithms. He used a set of bones to perform multiplication. Each bone was carved with numbers on it. These numbers were so carved with numbers on it. These numbers so carved that by keeping them side by side, the product of numbers could be obtained. Hence the name Napier's bones.

3] **Slide rule:** It consists of 2 scales, one of which slides over the other, It was so designed that whenever one scale slides over the other, The alignment of one on the other gave the result of basic arithmetic operations.

4] **Pascale in:** It was made up of counter wheels. This was capable of performing addition, subtraction, multiplication and division.

5] **Rotating wheel calculator:** It was designed based on the principle of counter wheels.

6] **Jacquard's loom:** A textile manufacturer, invented this machine to automatically control the weaving loom.

7] **Differential Engine:** A Professor of mathematics at the Cambridge university, Charles Babbage, invented the differential engine. It was used to calculate various mathematical functions.

8] **Analytical Engine:** Charles Babbage developed the analytical engine. This machine consisted of five functional units such as input unit, memory unit, arithmetic unit, control unit and output unit. The architecture of the modern digital computer resembles the analytical engine and hence Charles Babbage is called the father of computers.

9] **Mark I:** It was capable of performing a sequence of arithmetic operations. The time needed to perform multiplication and division was considerably reduced in this machine.

10] **ENIAC:** It is introduced after revolution in semiconductor technology. It is an acronym for electronic numerical integrator and calculator. This was able to carry out 5000 additions per second.

11] **EDVAC:** John Von Neumann Proposed a new concept of a large internal memory to store instructions and data. This is known as the stored program concept. The first computer developed on the principle of stored program concept was EDVAC. It is an acronym for Electronic Discrete Variable Automatic computer.

12] **UNIVAC- I:** It is an acronym for universal automatic computer I. It was used for scientific and commercial applications.

Generations of computers:

The generations of computers are broadly classified into two types.

1] **Non electronic generation:** These are also referred to as the zero generation computers. They were developed before the semiconductor revolution.

2] **Electronic generation:** Computer developed after 1946 are categorized into five generations. Since they are mainly built with electronic circuitry, They are called the electronic generation computers.

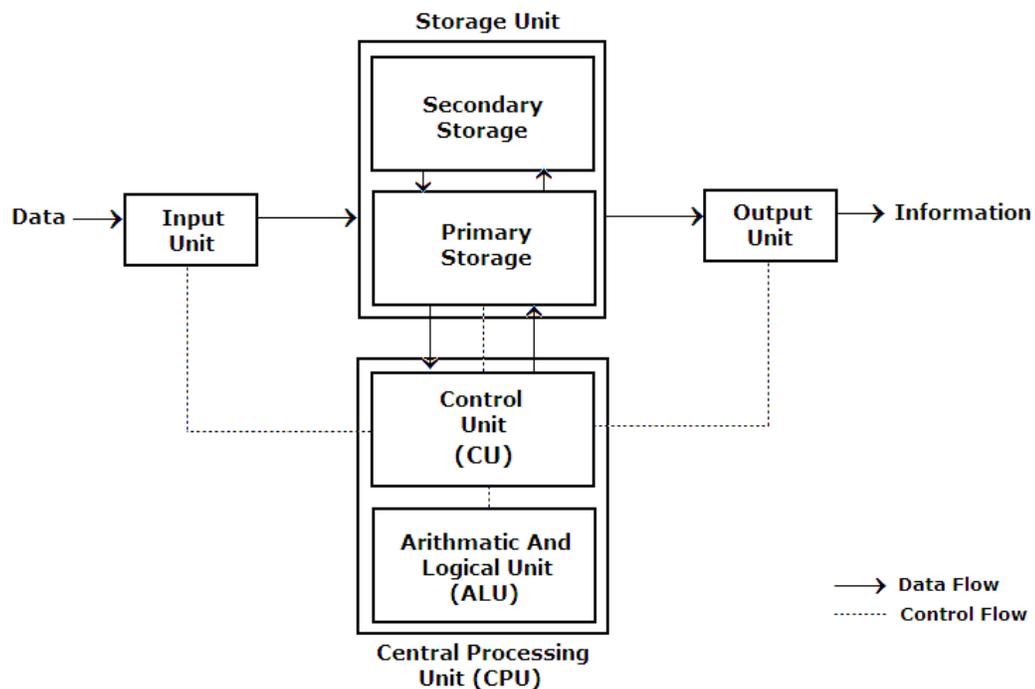
- First generation computers: These were built with vacuum tubes. Their speed was 10^{-3} sec.
- Second generation computers: These were built with diodes and transistors. Their speed was 10^{-6} sec.
- Third generation computers: These were built with integrated circuits(IC's). Their speed was 10^{-9} sec.
- Fourth generation computers: These were built with Large scale integration(LSI). Their speed was 10^{-9} to 10^{-12} sec.
- Fifth generation computers: The development of super computers was the key motivation of the fifth generation computers. These were developed with Super Large scale Integration(SLSI).

Basic Structure of a computer

A computer is a programmable device which performs the following 4 operations:

- 1] Accepting data and information from the user.
- 2] Storing data and information in its memory.
- 3] Processing data and information
- 4] Display the result.

Block diagram of a computer



Input device: This is used to read data and instructions into computer. The different input devices are keyboard, mouse, joystick, light pen etc. Keyboard is the most commonly used input device.

System unit: This is responsible for storing and processing of data and instructions. The system unit consists of CPU and memory devices. The term CPU stands for Central Processing Unit(CPU). The CPU is main unit of computer system. Which performs all arithmetic and logical operations. It is considered as brain of computer system. The data and instructions given by the user are processed in this unit. The CPU consists of Control unit(CU) and Arithmetic and Logic unit(ALU).

The control unit is an important unit in a computer. It controls and coordinates the activities of all the units of a computer system. The functions of control unit are

- 1] Reading data and instructions from memory.
- 2] Understand the instructions.
- 3] Controlling transfer of data and instructions to and from memory.
- 4] Controlling input and output devices.
- 5] Overall supervision of a computer.

The arithmetic and logic unit performs all arithmetic operations such as addition, subtraction, multiplication division and modulus. It also performs logical operations such as AND, OR and NOT.

Memory devices: Memory devices are used to store data and instructions given by the user. The computer memory is measured in terms of bits, bytes and words. A bit is a binary digit, It can store either 0 or 1. A byte is a sequence of 8 bits. A word is a combination of 2 bytes. The computer memory is classified into:

- 1] CPU registers
- 2] Primary memory
- 3] Secondary memory
- 4] Cache memory

The CPU registers hold a limited amount of memory during execution. They are inside the CPU. The main memory is the primary memory, Which holds data and instructions. The main memory is a temporary memory that holds data and instructions till power supply is there. But the secondary memory is a permanent memory, which holds data and instructions as long as user wants. The cache memory is a high speed memory and present in between CPU and memory. Unlike the main memory and secondary memory, the user cannot access the cache memory.

Output devices: Once the data and instructions are processed, the result can be displayed on output devices. The monitor is the most commonly used output device. The other output devices are printer, LCD's, speakers and disks.

Categorization of computers

The computers can be categorized based on

- 1] Principle of working
- 2] Size and capacity
- 3} Processor and storage

Computers based on principle of working

Based on the principle of working , computers are classified as:

- 1] Analog computers
- 2] Digital computers
- 3] Hybrid computers

Analog computers takes input which is continuous in nature such as temperature, pressure and voltage.

Digital computers reads input which is discrete in nature such as graphical data.

Hybrid computers are the combination of both analog and digital computers. The hybrid computers accept both analog and digital data as input.

Factors	Analog computers	Digital computers
Input	Physical variables	Graphical data
Processing	Integration	Arithmetic and logical
Output	Graphical	Numeric and graphical
Memory Unit	Normally not required	Necessary to store data and instructions
Accuracy	Limited	More
Application	Simulation	In solving business and scientific problems

Due to the advancement in the technology, the size , shape, processing power and price of the computers are changing every year. Based on these factors, computer are categorized into:

- 1] Computers for individual users.
- 2] Computers for organizations.

Computers for individual users: Computers for individual users are used by only person at a time. There are 6 types of computers in the individual users category. They are

- 1) Desktop computers
- 2) Workstations
- 3) Notebook computers
- 4) Tablet PC's
- 5) Handheld computers
- 6) Smart phones

Computers for organizations: Computers for organizations can be used by more than 1 person at a time. Such computers are classified as

- 1) Network servers
- 2) Mainframe computers
- 3) Mini computers
- 4) Super computers

The parts of a computer system

There are four parts in a computer system . They are,

- 1) **Hardware:** The mechanical components of a computer system are called hardware components. The example of hardware components are monitor, keyboard, hard disk mouse etc.
- 2) **Software:** Software is a program or a set of programs written to carry out specific task. A program is a set of instructions that performs certain tasks. Software tells the hardware what to do. Software components are more expensive compare to hardware.
- 3) **Data:** Data is the raw information the computers can process. Data by itself doesn't make much sense to the user. It will give a meaning only when it is processed by the computer. A processed data is called information. Data can be text, numbers, audio, video etc.
- 4) **User:** people who use computers are called users. No computer system is 100% autonomous. They need human being in one or other way. So user is considered as part of computer system.

Information processing cycle

Information processing cycle is a process that the computer follows in inputting data, processing, outputting data and storing data. There are four steps in an information processing cycle. They are

- 1) Input
- 2) Processing
- 3) Output
- 4) Storage

Input: Input is nothing but the data and instructions given to computer. In this phase , the computer accepts data and instructions from the user. Input devices are used to accept and pass the same to processing stage.

Processing: In this stage, the data and instructions given by the user are processed. The system unit of a computer is responsible for processing. In system unit, both arithmetic and logical operations are carried out to produce desired results.

Output: Output is nothing but the result. In this stage, the user can display the result. The results may be in the form of text, number, audio, image etc.

Storage: The data and instructions should be stored for future use. The memory unit is responsible for storage.

Algorithms and Flowcharts

Steps involved in problem solving

Problem solving by a computer involves following steps:

1. Problem definition
2. Analysis
3. Design
4. Coding
5. Running the program
6. Debugging
7. Testing
8. Documentation

Problem definition

In this step, the problem solver should understand what is the input need to be given , what is output we get and what is operation need to be performed.

Analysis: The given problem must be analyzed before it is solved. It involves finding type of input we need to give and also finding the type of operation we need to perform, that is whether it is arithmetic or logical operation.

Design: It is to write the blue print of solution. Algorithms ,flowcharts and pseudo code are some of the design techniques.

Algorithm: An algorithm can be defined as a step by step procedure to solve a particular problem. It consists of English like statements. Each statements must be precise and well defined to perform specific operation.

The word algorithm is named by the mathematician name “Abu Jafar Mohammed Ibn Musa Al Khowarizmi” the last two words of name took different pronunciations over the period such as ‘Alkhowarizm’,’Algorism’ and finally it became “Algorithm”.

Characteristics of an algorithm

Every algorithm have five important characteristics:

1] **Input:** Algorithm may accept zero or more inputs.

2] **Output:** It should produce at least one output, output is nothing but the result.

3] **Definiteness:** There should not be only ambiguity in instructions.

4] **Finiteness:** If we convert algorithm to program, the execution of program should complete in fixed time. It should not take infinite time.

5] **Effectiveness:** The operations defined in algorithm should be simple.

Algorithmic Notations

While writing algorithms, the following notations need to be followed.

1] Name of the algorithm: Give meaningful name for an algorithm, which specified the problem to be solved.

2] Step number: Assign number to each instruction of an algorithm.

3] Explanatory comment: It is the description or explanation about an instruction. It should be written with in square brackets.

4] Termination: It specifies the end of the algorithm. It generally a stop statement and it should be the last instruction of algorithm.

Example 1: Write an algorithm to compute the area of circle.

Algorithm: Area of circle

Step 1: Read Radius

Step 2: [compute area]

$$\text{Area} = 3.14 * r * r$$

Step 3: [print the area]

Print "area of circle=", area

Step 4: [End of an algorithm]

Stop

Example 2: Write an algorithm to perform the basic arithmetic operations such as addition, subtraction, multiplication and division.

Algorithm: Arithmetic operations

Step 1: Read A, B

Step 2: [calculate sum, difference, product and quotient]

$$\text{Sum} = A + B$$

$$\text{Difference} = A - B$$

$$\text{Product} = A * B$$

$$\text{Quotient} = A/B$$

Step 3: [print the Contents of sum, difference, product and quotient]

Print “ sum of A and B=”, sum

Print “difference, of A and B=”, difference,

Print “product of A and B=”, product

Print “quotient of A by B=”, quotient

Step 4: [End of an algorithm]

Stop

Example 3: Write an algorithm to find area of rectangle .

Algorithm: Area o rectangle

Step 1: Read l, b

Step 2: [calculate area of rectangle]

Area= l*b

Step 3: [print contents of area]

Print “ area of rectangle=”, area

Step 4: [End of an algorithm]

Stop

Example 4: Write an algorithm to calculate the simple interest and compound interest for the amount deposited (P) for some years (T) for the rate of interest (R).

Algorithm: simple interest and compound interest

Step 1: [read the value of P, R, T]

Read P, R, T

Step 2: [calculate simple interest]

$$SI = \frac{P * R * T}{100}$$

Step 3: [calculate compound interest]

$$CI = P * \left(1 + \frac{R}{100} \right)^T - P$$

Step 4: [print the contents of SI and CI]

Print "simple interest=", SI

Print "compound interest=", CI

Step 5: [End of an algorithm]

Stop

Example 5: Write an algorithm to find largest of two numbers.

Algorithm: Largest of two numbers

Step 1: Read A, B

Step 2: [compare A and B]

If $(A > B)$ then

Print " A is the largest"

else

Print " B is the largest"

Step 3: [End of an algorithm]

Stop

Example 6: Write an algorithm to find largest of 3 numbers.

Algorithm: Largest of three numbers.

Step 1:[read values of A, B, C]

Read A, B, C

Step 2:[compare A and B]

if $(A > B)$ go to step 4

Step 3: [compare B and C]

if $(B > C)$ then

print " B is largest"

else

print " C is largest"

Step 4: [compare A and C]

if (A>C) then

print “ A is largest”

else

print “ C is largest”

Step 5: [End of an algorithm]

Stop

Example 7: Write an algorithm to compute the sum of first N natural numbers and their mean.

Algorithm: sum and mean of natural numbers

Step 1:[read values of N]

Read N

Step 2: [set sum equal to zero]

sum=0

Step 3: [compare the sum of N natural numbers]

For num =1 to N in Step of 1 do

sum= sum+ num

Step 4: [compare mean]

$$\text{Mean} = \frac{\sum i}{n}$$

Step 5: [print contents of sum and mean]

Print “sum of N natural numbers=” sum

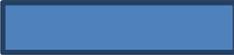
Print “mean of N natural numbers=” mean

Step 6: [End of an algorithm]

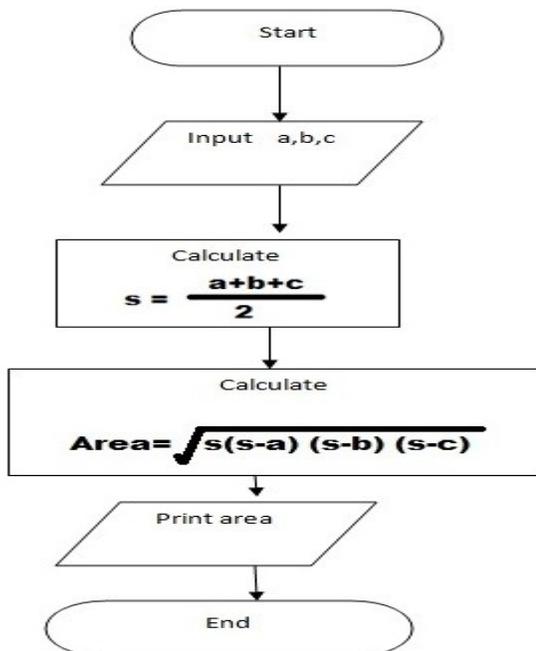
Stop

Flowcharts

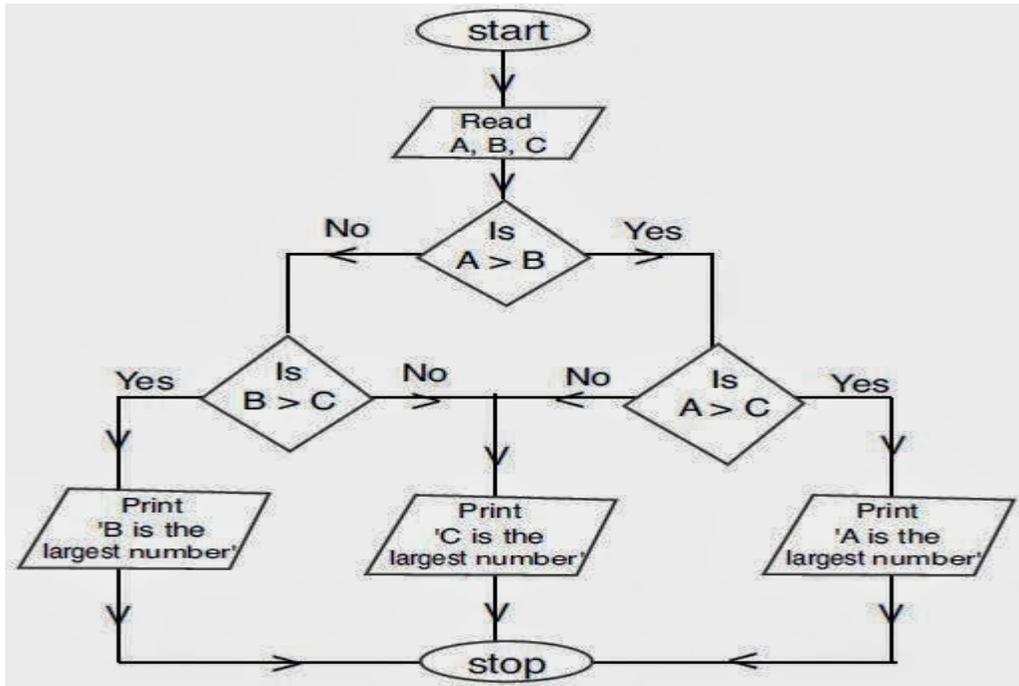
The flowchart can be defined as a diagrammatic representation or graphical representation of an algorithm. It is referred as blue print of an algorithm. Flow charts make use of geometrical figures. They are

Geometric figure	Name	Function
	oval	Start and stop
	Parallelogram	Input and output
	Rectangle	processing
	Diamond	Decision making
	Arrows	Connections
	Small circle	Continuation
	Hexagon	Looping/ Repitation

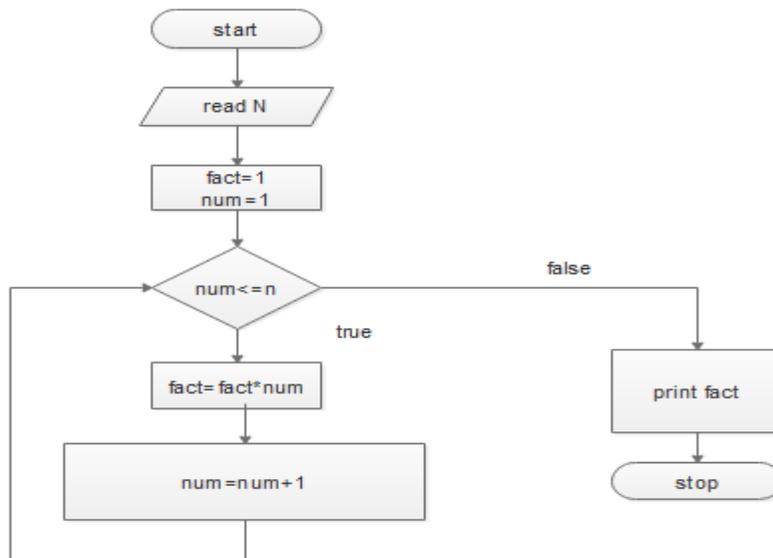
Example 1: Draw a flowchart to find the area of triangle when its three sides are given.



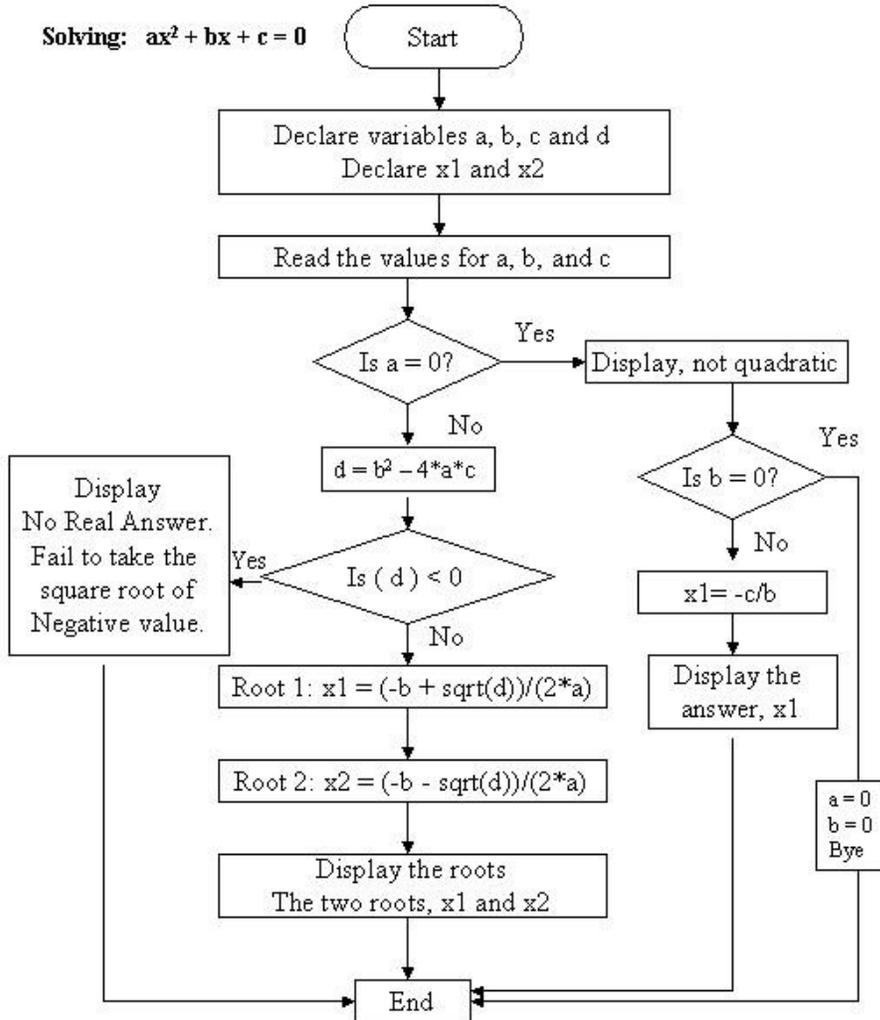
Example 2: Draw a flowchart to find the largest of 3 numbers



Example 3: Draw a flowchart to compute the factorial of a given number.



Example 4: Draw a flowchart to find the roots of a quadratic equation $ax^2+bx+c=0$



Example 5: Draw an algorithm and flowchart to reverse a given number.

Algorithm: Reverse an integer

Step 1:[read the number]

Read N

Step 2: [Initialization]

Temp =N

Step 3: [Repeat until N greater than zero and compute the reverse]

Reverse = 0

Temp =N

While (N>=0)

Digit= N mod 10

Reverse= Reverse*10+digit

N=N/10

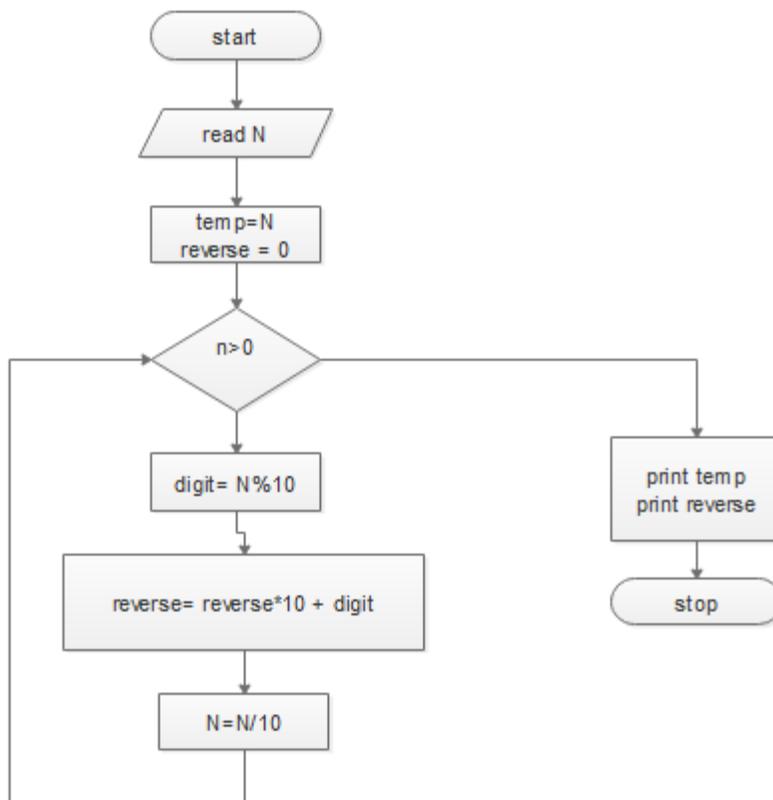
Step 4:[print given number and its reverse]

Print “ given number=”, temp

Print “ Reverse of number=”, reverse

Step 5: [End of an algorithm]

Stop



Pseudo code: A Pseudo code is neither an algorithm nor a program. It consists of English like statements which performs specific operations. In pseudo code, the program is represented in terms of words and phrases, but the syntax is not followed.

The Pseudo code is

- Easy to read
- Easy to understand
- Easy to modify

Example: Write a pseudo code to perform basic arithmetic operations

Read a , b

Sum = a + b

Diff= a – b

Product= a * b

Quotient= a/ b

Print sum, diff, product, quotient

End

Coding: The process of writing program called coding. The computer does not process algorithm or a flow chart, but executes the program. A program is a set of instructions to solve a problem by computer.

Running the program : The program will be executed in the CPU. This phase of problem solving involves 3 steps.

- Store data and instructions
- Understand the instructions
- Perform computations

The user writes program and store it in memory. All the instructions stored in the memory, are read one by one by the CPU, understand it. Then it will perform specified operations. The result of operations is stored in memory again. And also the result is displayed in output devices.

Debugging: Bug is nothing but error or mistake. The process of detecting and correcting errors in the program is called debugging. The term debug was introduced after detecting a bug in Mark-I, by Admiral Gracehopper. She removed the bug from the machine and wrote in a "The Mark - I was debugged today".

Generally programmer do 3 types of errors, they are

- Syntax error
- Logical error
- Run time error

Syntax error: This type of errors will occur when we don't follow syntax while writing programs. On encountering these errors the compiler displays an error message specifying the line number where the error has occurred. It is easy to debug these errors. For example, the syntax of initializing a variable in c is

Variable = expression;

If we miss semicolon in the above statement, then there will be an error.

Logical errors: Logical errors occur during coding process. When the programmer writes a program, he must take care of correct operations to be performed. Even though logical errors are there in a program, the program will get executed, but it will give unexpected results. It is very difficult to debug such as errors, because compiler doesn't display them. We can eliminate such errors through execution of program by giving inputs.

Runtime errors: These errors when we execute instructions which have ambiguity. These errors may occur due to errors in program like divide by zero or due to device errors, keypunch errors, incorrect data input etc. The computer will print these error messages.

Testing: The process of executing the program to test the correctness of outputs is called testing. The program is tested by executing with different sets of data. We can find logical errors using the process.

Documentation: While writing programs, it is a good programming practice to write a brief explanation on the program. This explanation is called comment. It explain how the program works and how to interact with it. Thus, it helps the other programmers to understand the program.

There are 2 types of documentation. They are,

- Internal documentation
- External documentation

Internal documentation: This documentation is a comment statement within a program. It describes the function of the program. These statements are not converted to machine language.

There are 2 types of comments.

- Single line comments: These will start with //
- Multiline comments(block comments): These comments should be written with in /* comment line */

External documentation: This documentation is an executable statement in a program. It may be a message to the user to give inputs. These can be written using output statements. It makes the program more attractive and interactive. Some examples are,

```
printf("enter values of a and b");
```

```
printf("do you want to continue");
```

Introduction to Programming

Importance of C

C became very popular compared to other programming languages introduced before C like BASIC, COBOL, FORTRAN etc. The increasing popularity of C is due to its desirable qualities. They are

- 1) C has rich set of built in functions and operators which can be used to write any complex program.
- 2) Programs written in C are efficient and fast C is many times faster than BASIC.
- 3) They are only 32 keywords in ANSI C.
- 4) C is portable language. This means that C programs written in one computer can be run on another computer with little or no modification.
- 5) C language is well suited for structured programming, this requires the programmer to think of a problem in terms of functions.
- 6) C has the ability to extend itself. A C program is basically a collection of functions that are supported by C library. We can add our own functions to C library.

Basic structure of C programs

The documentation section consists of a set of comment time giving the name of the program the author name and other details,

The link section provides instructions to the compiler to link functions from system library like `stdio.h`, `stdlib.h` etc

In definition section, we can define all symbolic constants

These are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section.

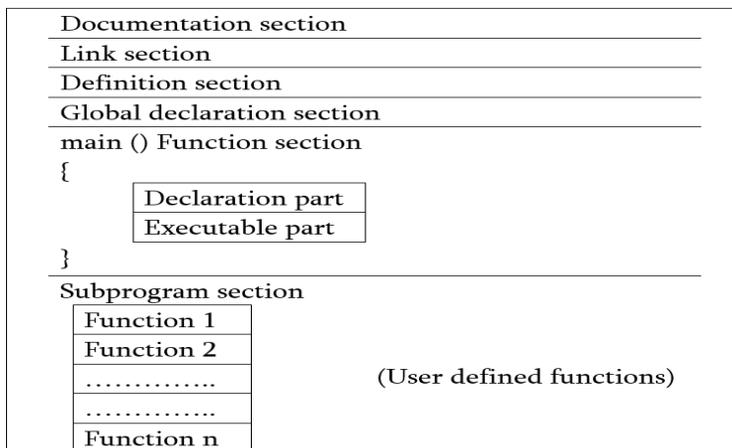


Fig: Basic structure of C program

Every c program must have on main () function section.This section contains two parts, declaration part and executable part. The declaration part is used to declare all the variables which need to be used in executable part. There should be at least one statement in the executable part. These two parts must appear between opening and closing braces. The program execution begins at open brace of main () function and ends at closing brace. The closing brace of main () function is the logical end of program. All statements in declaration and executable part must end with a semicolon.

Executing a C program

Execution of a program in C involves following steps.

1. Creating the program
2. Compiling the program
3. Linking the program with functions that are needed from c library.
4. Executing the program

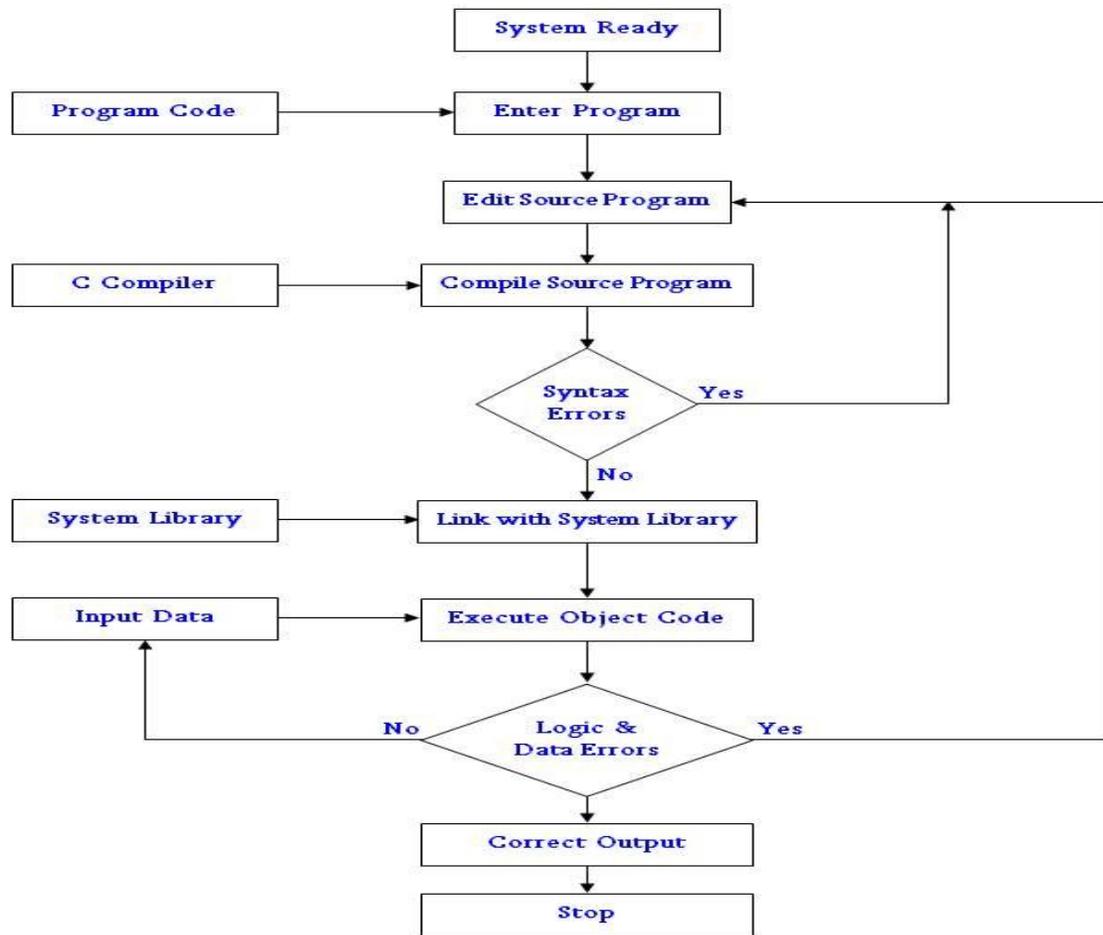


Figure : Process of compiling and running a C program.

Constants variables and data types

Character set

The group of characters which are used to program is called characters set. The characters in C are grouped into following categories.

- 1) Letters(A-Z and a-z)
- 2) Digits (0 to 9)
- 3) Special characters
- 4) White spaces.

Special characters

, comma	vertical bar	^ caret
---------	--------------	---------

. period	/ slash	*asterisk
; semicolon	\ backslash	- Minus sign
: colon	~ tilde	+ plus sign
? question mark	_ underscore	<opening angle bracket(or) less than sign
' apostrophe	\$ dollar sign	> closing angle bracket(or) greater than sign
“ quotation mark	% percent sign	(left paranthesis
! exclamation mark	& ampersand) right paranthesis
[left bracket] right bracket	{ left brace
} right brace	# number sign	

White spaces:- These are used to separate words

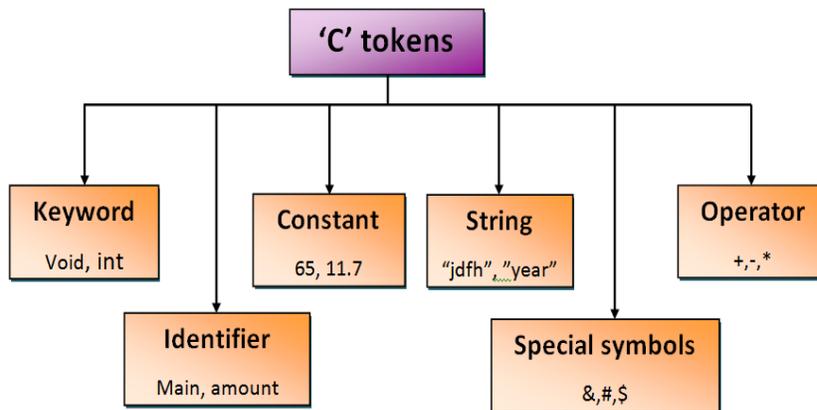
Trigraph characters:- Many non English keyboards do not support all the characters which are mentioned above. So ANSI C introduces the concept of trigraph characters to provide a way to use these characters in a program, which are not available in keyboard.

Each trigraph character starts with ??for example

Trigraph Character	Translation
??=	#
??([
??)]
??<	{
??>	}
??!	
??/	\
??-	~

C tokens

The smallest units of a program are called tokens C token are categorized into six types They are



Keywords and identifiers

Each C word can be categorized as keyword or an identifiers. Keywords are the words which have special meaning and which should not be changed. There are totally 32 keywords in C. They are

Auto	continue	extern	long	sizeof	unsigned
Break	default	float	register	static	double
Case	do	for	return	switch	int
Char	else	goto	short	typedef	struct
Const	enum	if	signed	union	void
Volatile	while				

Identifies are names of variables, functions and arrays.

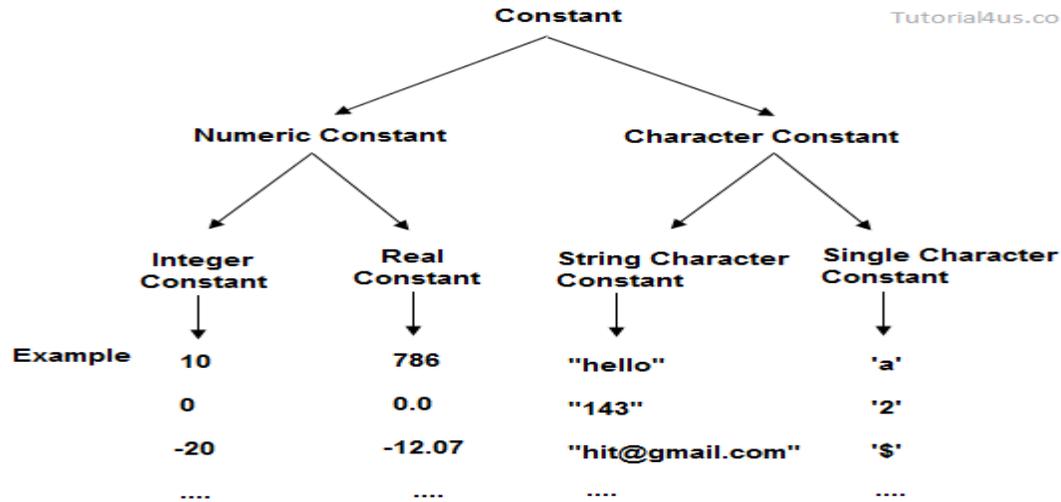
These are user defined names and consists of letters, digits and underscore.

Rules to create identifiers

- 1) First character of variable must be alphabet (or) underscore
- 2) Variable name must consist of only letters, digits and underscore
- 3) Variable name should not exceed 31 characters
- 4) Keywords should be used as variable
- 5) Variables should not contain white spaces.

Constants

Constants are the values which do not change throughout execution of program. C supports several types of constants. They are



Integer Constants

An integer constant is a number which consists of a sequence of digits. Integer constants are into 3 types

- 1) Decimal integer constants
- 2) Octal integer constants
- 3) Hexadecimal integer constants

Decimal integer constants consist of a set of digits, 0 through 9. These constants may be positive or negative. Valid examples of decimal integer constants are

12 -125 12468—96

Embedded spaces, commas and special characters should not be used. For example

12 60 20,000 5\$85

are invalid

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. For example

015 0751 0468

A sequence of digits preceded by 0x or 0X are called hexadecimal integer constants. They can also contain alphabets A to F. The letter A through F represent the number 10 through 15. For example

0x76A 0XAF 0x5D

Real constants

Real constants are numbers which have a fractional part. Real constants can be positive or negative. For example

0.0083 - 0.72 435-36

It is possible to omit digits before decimal points or digits after decimal point for example

215. .95 -.71 .5

A real number can also be expressed as exponential (or) scientific notation. The general syntax of scientific notation is

Mantissa E/e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. Examples are

0.65e4 12e-2 1.5e0 5 3.18 E3 -12E-1

The value 215.65 can also be written as 2.156 5e2

Single character constants

A single character constant contains a single character enclosed within a pair of single quotes. For example

'5' 'x' ';'

Character constants are associated with integer values called ASCII values. For example

Print("%c", '97')

Would print the number 97. Similarly

Print("%c", 'a')

Would print the output as 'a'

String constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, digits, special characters and so on. Examples are

“abcd” , “1987” , “hello world” , “+ - 53”

The character constant ‘x ’is not same as string constants “x” the string constants are not associated with integer values.

Backslash character constants

C supports some special backslash character constants that are used in output functions. They are also called escape sequences. They are

Constants	Meaning
‘\a’	Audible alert (bell)
‘\b’	Back space
‘\f’	Form feed
‘\n’	New line
‘\r’	Carriage return
‘\t’	Horizontal tab
‘\v’	Vertical tab
‘\’	Single quote
‘\”	Double quote
‘\?’	Question mark
‘\’	Backslash
‘\0’	Null character

Variables

A variable is name of memory location where we can store data. A variable may take different values at different times throughout program execution. The programmer should give meaningful name for a variable.

Data Types

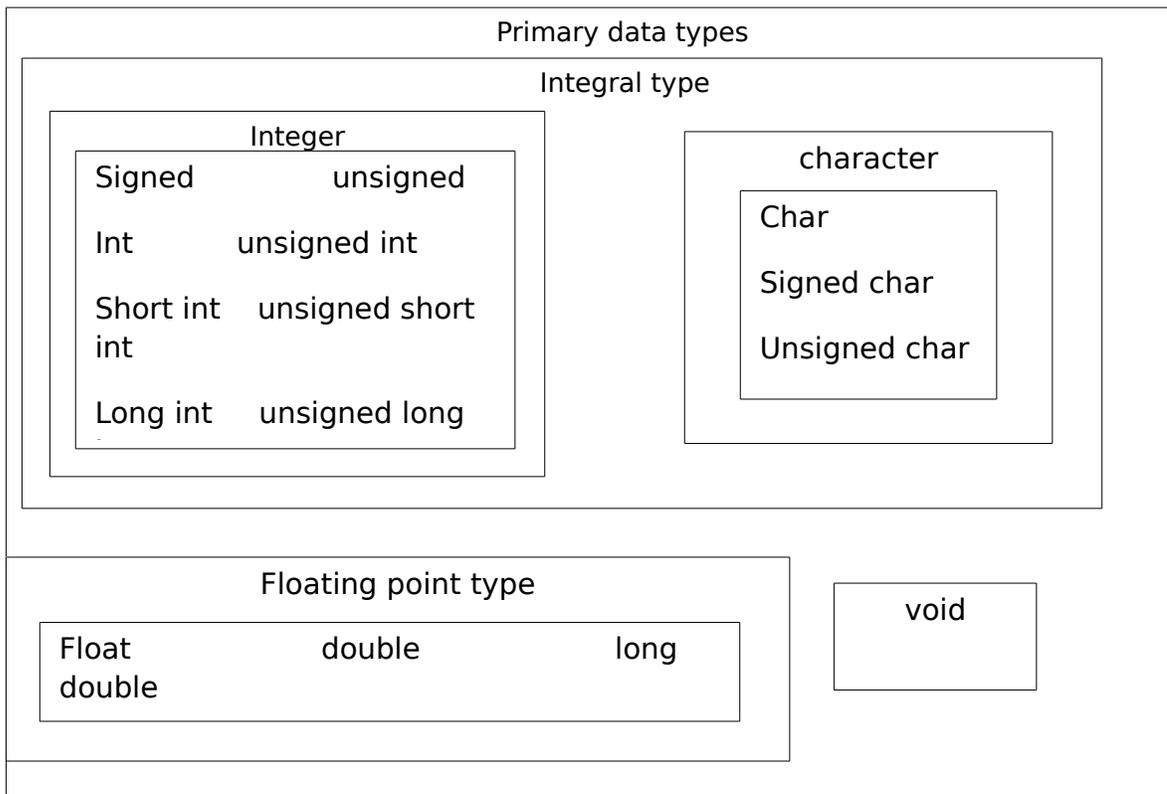
C supports three classes of data types. They are

- 1) Primary data types
- 2) Derived data types
- 3) User defined data types

The primary data types supported by C are

- 1) Int (integer)
- 2) Float (floating point)
- 3) Double (double precision floating point)
- 4) Char (character)
- 5) Void

Data type	Range of values
Char	-128 to 127
Int	-32,768 to 32,767
Float	3.4e-38 to 3.4e +38
Double	1.7e-308 to 1.7e+308



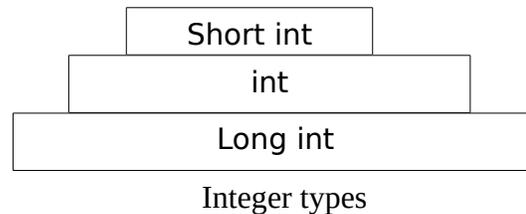
Integer types

Integers are whole numbers. Generally, integers occupy 2 bytes or 4 bytes to store a number, depending on the computer. The integers can be signed or unsigned. If we declare int, by default computer assumes it as signed integer, where we can store both positive and negative numbers. A signed integer uses 1 bit for sign and remaining bits to store number. If an integer takes 2 bytes, then the range of values can be stored are -2^{15} to $2^{15} - 1$ for signed integer.

C has 3 classes of integer storage namely short int, int and long int. Short int takes half the size of int and long int takes double the size of memory to store number. Signed and unsigned are called qualifiers.

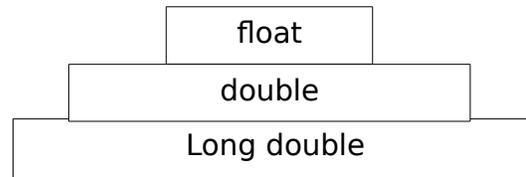
Size and range of data types

Type	Size	Range
Char or signed char	8	-128 to 127
Unsigned char	8	0 to 255
Int or signed int	16	-32,768 to 32,767
Unsigned int	16	0 to 65535
Short int or signed short int	8	-128 to 127
Long int or signed long int	32	-2,147,483,648 to 2,147,483,647
Unsigned short int	8	0 to 255
Unsigned long int	32	0 to 4,294,967,295
float	32	$3.4E - 38$ to $3.4E + 38$
double	64	$1.7E - 308$ to $1.7E + 308$
Long double	80	$3.4E - 4938$ to $1.1E + 4932$



Floating point types

Float uses 4 bytes to store numbers, with 6 digits of precision. Floating numbers are numbers which have fractional part. If the accuracy provided by float is not sufficient, the type double can be used to store number, which uses 8 bytes giving a precision of 14 digits. If the accuracy provided by double is not sufficient, then we can use long double. Long double takes 10 bytes to store a number.



Data types and their keywords

Data type	Keyword
character	char
Unsigned character	Unsigned char
Signed character	Signed char
Signed integer	Signed int/ int
Signed short integer	Signed short int/ short int/ short
Signed long integer	Signed long int/ long int/ long

Unsigned integer	unsigned int/ unsigned
Unsigned short integer	Unsigned short int/ unsigned short t
Unsigned long integer	Unsigned long integer/ Unsigned long
Floating point	float
Double precision floating point	double
Extended double precision floating point	Long double

Character types

A single character can be defined using char data type. Char takes 1 byte of memory. The qualifier signed or unsigned char can be explicitly applied to char. The signed chars have range of values 0 to 255, signed chars have range of values -128 to 127.

Void types

Void is used with functions.

Declaration of variables

Declaration of variable does two things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program. The syntax for declaring a variable is data-type v1,v2,.....vn;

v1,v2,.....vn are name of variables. Variables are separated by commas. A declaration statement must end with semicolon. Examples are

```
int amount;
float SI,CI;
double area;
```

User defined type declaration

typedef is a keyword which is used to give new name to existing data type, the syntax is

```
typedef type identifier;
```

type is name of existing data type, identifier is the new name given to existing data type. For ex

```
typedef int whole_number;
```

```
typedef float fraction;
```

These new names can be used to declare variables like

```
whole_number a, b;
```

```
fraction x, y;
```

typedef cannot create new data type. The main advantage of typedef is that we can give meaningful names to data types which increases the readability of program.

The another user defined data types is enumerated data type, the syntax is

```
enum identifier{ value1, value2,...valuen};
```

The identifier is a user defined enumerated data type. enum is used to give meaningful names to integer constants. For ex

```
enum day{ Monday, Tuesday, Wednesday, Thursday };
```

The compiler automatically assigns integer digits beginning with 0 to all enumeration constants. That is the enumeration constant Monday is assigned with 0, Tuesday is assigned with 1 and so on.

To avoid automatic assignments, we can define like

```
enum day { Monday=10, Tuesday, Wednesday, Thursday };
```

Here the constant Monday is assigned with value 10. The remaining constants are assigned values that increase successively by 1. We can also define like

```
enum day { Monday=10, Tuesday, Wednesday=25, Thursday };
```

then Tuesday will be assigned with 11 and Thursday will be assigned the value 26.

The enum data type is used to declare variables like

```
Day week-day=Monday;
```

Then week-day is assigned assigned with value 10.

Declaration of storage class

Variables in C can have not only data types but also storage class that provides information about lifetime of variables. In C, there are 4 storage classes. They are

1. **Auto:** Local variable can be used only within the function/block in which it is declared.
2. **Static:** These variables will retain values even after completion of function execution.
3. **Extern:** Variable declared I one file can be used in another file using extern storage class.

4. **Register:** Variables stored in registers.

Examples of declaration of variables using storage classes are

```
auto int count;  
extern int amount;  
register float rate;  
static double total;
```

NOTE: static and extern variables are automatically initialized to zero. Auto variables have garbage values before initialization.

Assigning values to variables

Values can be assigned to variable using assignment operator(=). For ex

```
a=10;
```

At the time of declaration itself, a variable can be initialized like

```
int b=25;
```

In single C statements, multiple assignments statements can be written like

```
X=25; y=25;
```

The variables can also be initialized by writing expressions like

```
X=count+2;
```

The expression at RHS will get evaluated first, and that value is assigned to x.

Symbolic constants

We can use constants in a program. These constants may appear in a number of places in program. One example is 3.1412, representing the mathematical constant 'pi', we face two problems in writing programs which contains constant values. They are

- 1] problem in modification of program
- 2] problem in understanding the program

Modifiability

If we want to change value of pi from 3.14 to 3.14159 to improve accuracy, then we need to search whole program for the usage of pi value and then we need to change the value explicitly.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things to different places. For ex, if we use 3.14 value in our program, it may not be value of pi always. So if we give meaningful names to values, then it increases understandability.

The syntax to define symbolic constants are

```
#define symbolic_name constant_value
#define is preprocessor directive. Symbolic_name is name of constant. Constant_value is
value associated with a constant. For ex
#define PI 3.14
```

The following rules need to be followed while defining symbolic constants.

1. Symbolic names have same form as variable names.
2. No blank space between # and define.
3. # must be first character in line
4. Keywords should not be used as symbolic name
5. After definition, symbolic constants should not be assigned with any other values.
6. #define statements must be terminated with semicolon.
7. A blank space is required between #define and symbolic name and also between symbolic name and value.

Declaring a variable as constant

We may want to declare certain variables as constants in a program. It can be done with the const qualifier at the time of declaration. For ex

```
const float interest_rate=0.1;
```

Const is a keyword which tells the compiler that the value of interest_rate should not be modified by program.

Managing Input and Output Operations

The set of functions used for input and output operations are collectively known as standard I/O library. Each program that uses a standard Input/ Output function must contain the statement #include<stdio.h> at the beginning.

Reading a character

The simplest of all input/output operations is reading a character from the standard input unit (keyboard) and writing it to standard output unit (monitor). The character can be read using the function getchar(). The general syntax of getchar() is

Variable_name is valid c name that has been declared as char type. For ex

```
char d;
d = getchar( );
```

getchar() function reads a character from user and stores it in the variable d.

writing a character

Like getchar(), putchar() function is used to write character to output device. The syntax is

```
putchar (variable_name);
```

Variable_name is a char variable contains a character. For ex

```
d='a'  
putchar( d);
```

putchar() will display character 'a' on output screen.

Ex: Read a character from user and print it on to output device.

```
#include<studio.h>
```

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

```
int main( )
```

```
{
```

```
char d;
```

```
printf(" Enter a character/n");
```

```
d = getchar( );
```

```
printf(" The give character is:");
```

```
putchar(d);
```

```
return 0;
```

```
}
```

output:

Enter a character

K

The given character is : K

Formatted input

Formatted input refers to an input data that has been arranged in a particular format. scanf() function is used to read input from user. The general syntax of scanf() is:

scanf("control string", V1,V2,...Vn); The control string specifies the format in which data is to be read and V1,V2,...Vn specify the address of locations where data obtained be stored. control string and variables are separated by commas.

Inputting Integer numbers

The syntax of control string to read an integer is %wd.

The % indicates a conversion specification, w is a width specified and d indicates integer data to be read. For ex

```
scanf(“%d”, &a);
```

& is an address operator. The width can be use as

```
scanf(“%2d%4d”, &a, &b);
```

%2d indicates read 2 digits integer number, %4d indicates read 4 digits integer number. Store first number in address of a and second number in address of b.

An input field may be skipped by specifying * in the place of field width. For ex

```
scanf(“%d %*d %d”, &a, &b);
```

If the input given is 123 456 789 .Then it will assign 7 it will assign 123 to a, 456 will be skipped and it will assign 789 to b.

By default, space is used as separated between two inputs. If we want to change default separator, then we can specify it in control string like

```
scanf(“%d$%d”, &a, &b);
```

For the above, input should be given as 4\$5.

The data type character d may be replaced by l to read long integers and h to read short integers.

Ex: int main()

{	output
int a, b , c ,d;	
printf(“ enter 2 numbers”);	enter 2 numbers
scanf(“%d %d”, &a, &b);	2 3
printf (“a=%d, b=%d”, a, b);	a=2, b=3
printf(“ enter 3 numbers”);	enter 3 numbers
scanf(“%d%*d %d”, &a, &b);	2 3 4
printf (“a=%d, b=%d”, a, b);	a=2 b=4

```

printf(" enter 2 numbers");
scanf("%d$d", &a, &b);
printf ("a=%d, b=%d", a, b);
printf(" enter 2 numbers");
scanf("%2d%4d", &a, &b);
printf ("a=%d, b=%d", a, b);
scanf("%4d%d", &a, &b);
printf ("a=%d, b=%d", a, b);
return 0;
}

```

```

enter 2 numbers
2 $ 3
enter 2 numbers
1234 58
a=12 b=34
123
a=58 b=123

```

Inputting real numbers

The control string used to read float umbers is %f. For ex,

```
scanf("%f %f", &a , &b);
```

If the number needs to be read in scientific rotation, then control string can be used is %e like

```
scanf("%e", &a);
```

If the number to be read is of double type, then the specification should be %lf instead of %f.

Inputting character strings

The %c can be used to read a character. For ex

```
scanf("%c", &d);
```

To read a word, we can use %s. For ex

```
int main( )
```

```
{
```

```
char name[10];
```

```
printf("enter string");
```

```
output
```

```
scanf("%s", name);
```

```
enter string
```

```
printf("name=%s", name);           vinay
return 0;                          name= vinay
}
```

By using %S we cannot read multiple words, because %S consider space as end of input. For that we can write

```
scanf("[a-z]", name);
```

It indicates that read the input which contains any character between a to z and space.

```
scanf("[^$,+]", name);
```

If we use '^' symbol, it indicates, accept input which should not contain any character which are specified in square brackets.

Reading mixed data types

In a single statement, we can read different type of data like

```
scanf("%d %f %c", &a, &b, &c);
```

Commonly used scanf format codes

Code	Meaning
%c	Reads a single character
%d	Reads a decimal integer
%e	Reads float value in scientific rotation
%f	Reads a floating point value
%h	Reads a short integer
%o	Reads an octal integer
%s	Reads a string
%u	Reads an unsigned decimal integer
%x	Reads a hexadecimal integer
%[. .]	Reads a string of words

Formatted output

It is desirable that the outputs are produced in such a way that they are understandable and are in an easy to use form. printf() function is used to print output on monitor. The general syntax of printf () is

```
printf(" control string", V1,V2,V3....Vn);
```

Control string consists of 3 things.

1. Characters (or) information need to be print as it is.
2. Format specifications that define the output format.
3. Escape sequence characters such as \t, \n etc

The control string indicates how many to be printed and what are its types. The variables V1,V2...Vn are printed according to specification of control string. The variables should match in number, type and order with format specifications.

Output of integer numbers

The format specification for printing integer number is %wd

Where w is called width specifier.

Format

output

printf(“%d”, 1286);

1	2	8	6
---	---	---	---

printf(“%6d”, 1286);

		1	2	8	6
--	--	---	---	---	---

printf(“%2d”, 1286);

1	2	8	6
---	---	---	---

printf(“%06d”, 1286);

0	0	1	2	8	6
---	---	---	---	---	---

printf(“%-6d”, 1286);

1	2	8	6		
---	---	---	---	--	--

If the width specifier is less than number of digits of a number, the complete number will be printed. To force left justification, we can use ‘-’ sign. The digit (or) character before width specifier is called fill character and is used to fill empty spaces.

Output of real numbers

The format specification to print real number is %w.pf

The integer w indicates the minimum number of positions that are to be used for the display of value and the integer p indicates the number of digits to be printed after decimal point.

Format

output

Printf(“%f”, 57.7864);

5	7	.	7	8	6	4
---	---	---	---	---	---	---

5	7	.	7	8	6	4
---	---	---	---	---	---	---

Printf(“%7.4f”, 57.7864);

The subprogram section contains all user defined functions. All sections, except the main function section and link section may be absent when they are not required.

Programming style

unlike some other programming languages(COBOL, FORTRAN etc) C is a free form language. That is the C compiler does not care, where on the line we begin typing.

We must develop the habit of writing programs in lower case letters. C program statements are to be written in lowercase letters.

Since C is a free form language, we can group statements on one line. For example

```
A = b;  
X = y + 1;
```

Can also be written as

```
A = b; x = y + 1;
```

The code segment

```
Main()  
{  
  Printf("hello");  
}
```

Can also be written as

```
main( ) { Printf("hello");}
```

however, this style make the program more difficult to understand and should not be used.

UNIT 2

Decision Making And Branching

C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions are there. We may have a number of situations, where we want to change the order of execution of statements based on certain conditions are met. A group of statements until certain conditions are met. This involves kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

In C language, the following statements support decision making capabilities.

1. if statement
2. switch statement
3. conditional operator statement
4. goto statement

these statements are popularly known as decision making statements. Since these statements control the flow of execution, they are also known as control statements.

Decision making with if statement

The if statement is basically a two way decision statement. The if statement may be implemented in different forms. They are

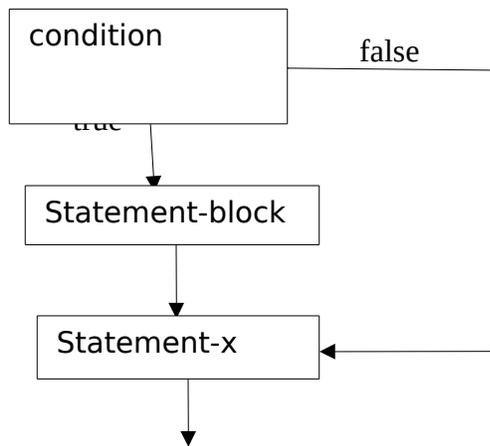
1. simple if statement
2. if else statement
3. nested if else statement
4. else if ladder

Simple if statement

The general form of a simple if statement is

```
if (condition)
{
    Statement- block;
}
Statement-x;
```

The statement-block may be single statement or group of statements. If the test expression or condition is true, then the statements-block will be executed. If the condition is false, then statement-block will be skipped. When the condition is true, both statement- block and statement-x will be executed.



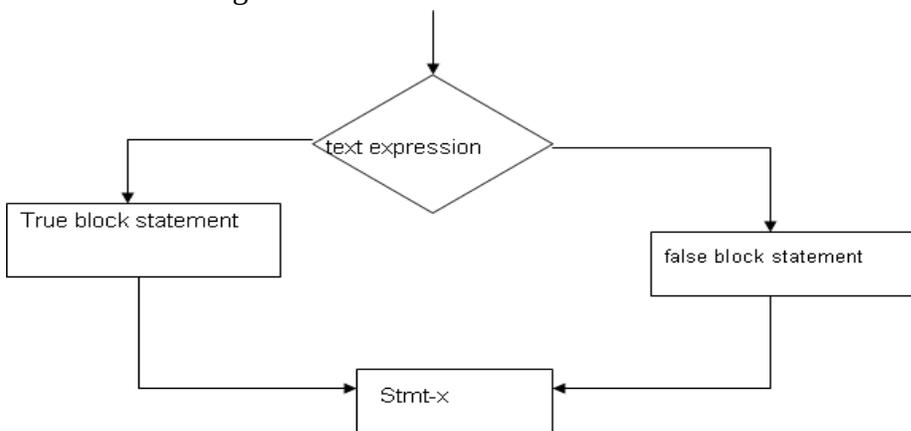
The if else statement

The if else statement is an extension of if statement. The general syntax of if statement is

```

if (condition)
{
    true block statements;
}
else
{
    false block statements;
}
Statement-x;
  
```

If the condition is true, then the true block statements will get executed. If the condition evaluates to false, then the false block statements will get executed. In either case, either true block statements or false block statements will get executed but not both. But in both the cases, statements-x will get executed.



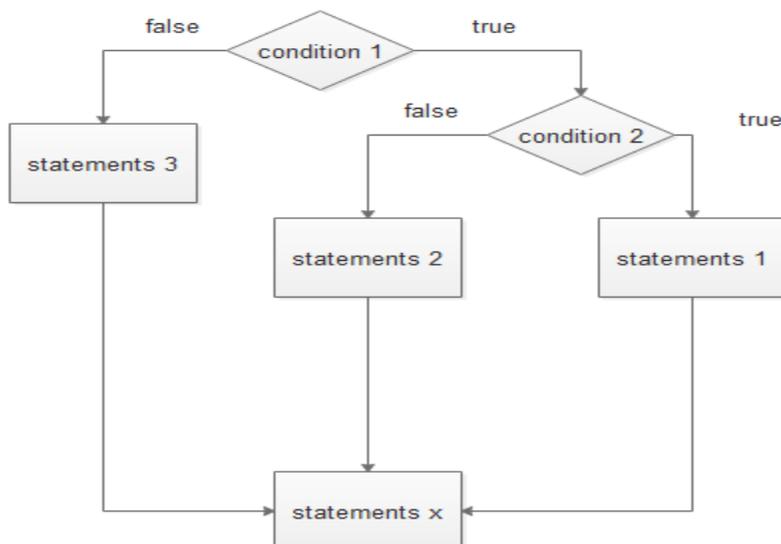
Nested of if else statements

When more conditions need to be evaluated, then we can use nested if else statement. The if else statement written inside another if is called nested if statement. There is no limit on number of levels of nesting. The general syntax of nesting of nesting of if else statement is

```
if (condition)
{
    if (condition 2)
    {
        Statement-1;
    }
    else
    {
        Statement-2;
    }
}
else
{
    Statement-3;
}
Statement-x;
```

If condition1 evaluates to true, then it will evaluate condition2. If condition2 evaluates to true, then statements 1 will get executed. If condition2 evaluates to false, then statements 2 will get executed.

If condition1 evaluates to false, then statements 3 will get executed. In all cases, statements-x will get executed.

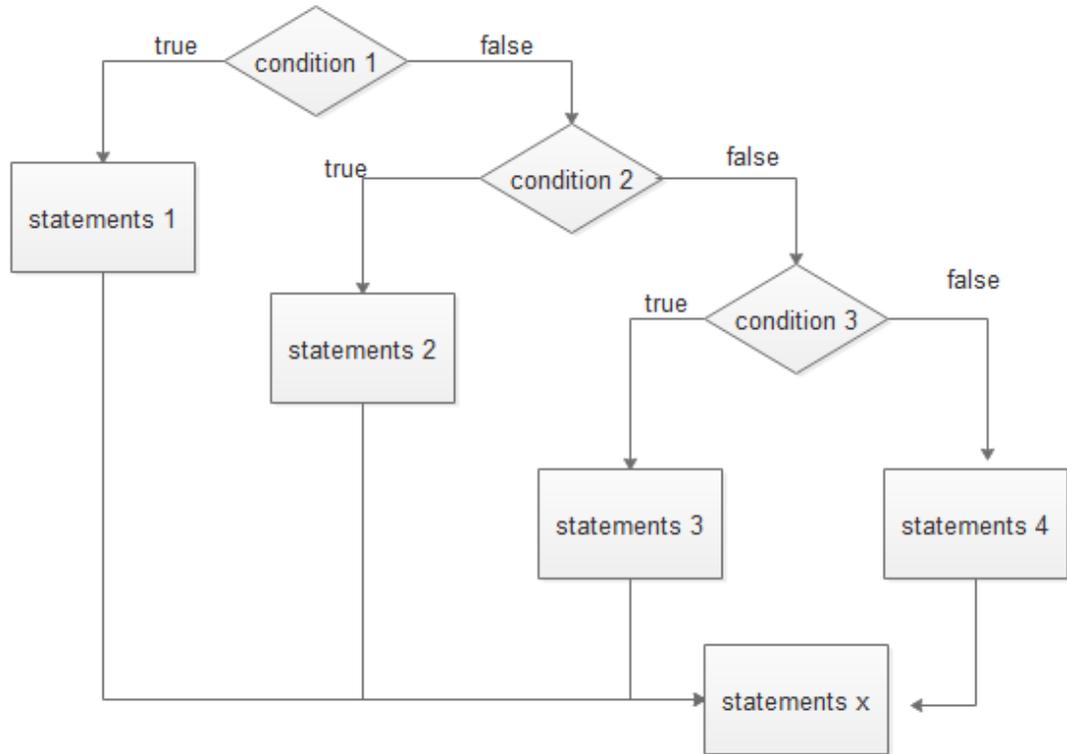


The else if ladder

The other way to evaluate multiple conditions is else if ladder. The general syntax of else if ladder is

```
if (condition 1)
{
Statement-1;
}
else if (condition 2)
{
Statement-2;
}
else if (condition 3)
{
Statement-3;
}
else
{
Statement-4;
```

}Statement-x;



If condition1 evaluates to true, then statements 1 will get executed. If condition2 evaluates to true, then statements 2 will get executed. If condition3 evaluates to true, then statements3 will get executed.

If condition1 evaluates to false, then condition2 will get evaluated. If condition2 evaluates to false, then condition3 will get evaluated. If condition3 evaluates to false, then statement 4 will get executed. In all cases, statement-x will get executed.

Switch statement

In a program, one output of many alternatives need to be selected, then we can use else if ladder. But the complexity of such a program increases when the number of alternatives increases. Fortunately, C has built in multi way decision statement known as a switch.

The switch statement tests the value of given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

The general syntax of switch statement is

```
Switch (expression)
{
Case value-1:
    block-1
```

```

        break;
Case value-2:
        block-2
        break;
.....
.....
default:
        default-block
        break;
}
Statement-x;

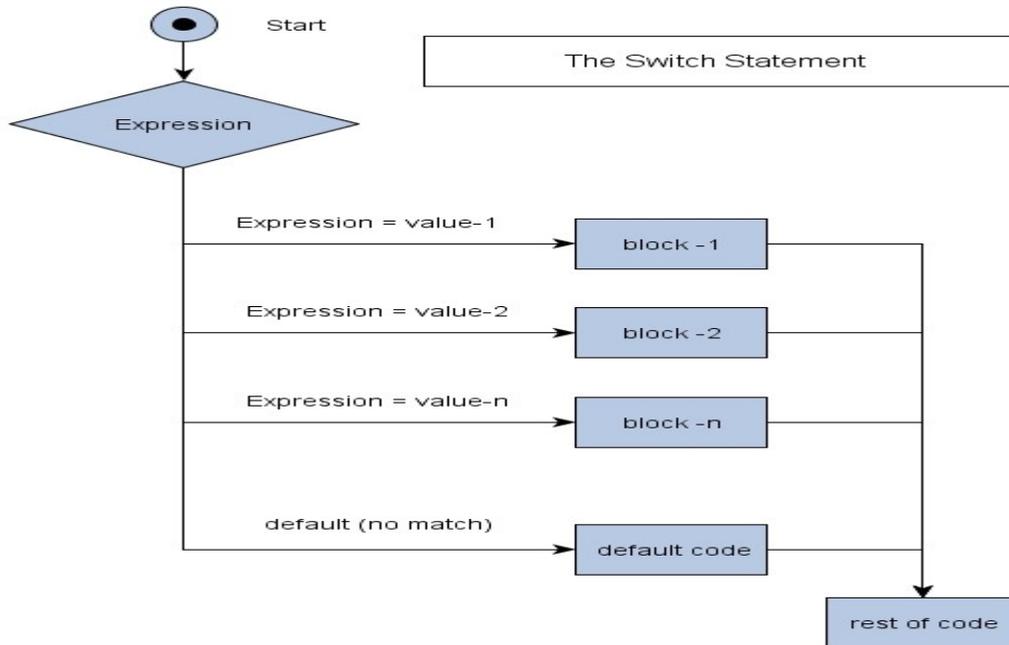
```

The expression is an integer expression or characters. Value-1, value-2 are constants or constant expressions and are known as case labels. Each of these values should be unique within a switch statement. block 1, block 2 etc are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a colon(:).

When a switch is executed, the value of the expression is successfully compared against the values value1, value 2 etc. If a case is found whose value matches with the value of the expression, then the block of statements that follow the case are executed.

The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x.



The ?: operator

C has an operator for two way selection. This operator is a combination of ? and : and takes 3 operands. The operator is popularly known as conditional operator. The general syntax of conditional operator is

Conditional expression ? statements 1: statements 2;

The conditional expression is evaluated first, if the result is true, then it executes statements 1. If the result of expression is false, then it executes statement 2. For ex

```
int y = ( x>0) ? 1 : 0;
```

If value of x is greater than 0, then 1 will be assigned to y. If value of x is not greater than 0, then 0 will get assigned to y.

The conditional operator may be nested for evaluating more complex assignment decisions. For ex

```
Y=((d==0)?1 : ((d>0)? 2:3))
```

If value of d is 0, then 1 is assigned to y. if value of d is greater than 0, then 2 will be assigned to y otherwise 3 will be assigned to y.

The goto statement

C supports the goto statement to branch from one point to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general syntax of goto and label statements are

goto label;	label:
.....	statement;
.....
label:
Statement;	goto label;
<u>Forwarded jump</u>	<u>Backward jump</u>

The label: can be anywhere in the program either before or after the goto label; statement. During program execution, when a statement like

goto begin;

is met, the flow of control will jump to the statement immediately following the label begin:. This happens unconditionally.

Note that a goto breaks the normal sequential execution of the program. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a backward jump. If the label; is placed after the goto label; some statements will be skipped and the jump is known as forward jump.

Goto statement can also be used to transfer the control out of a loop where certain peculiar conditions are encountered.

Dangling else problem

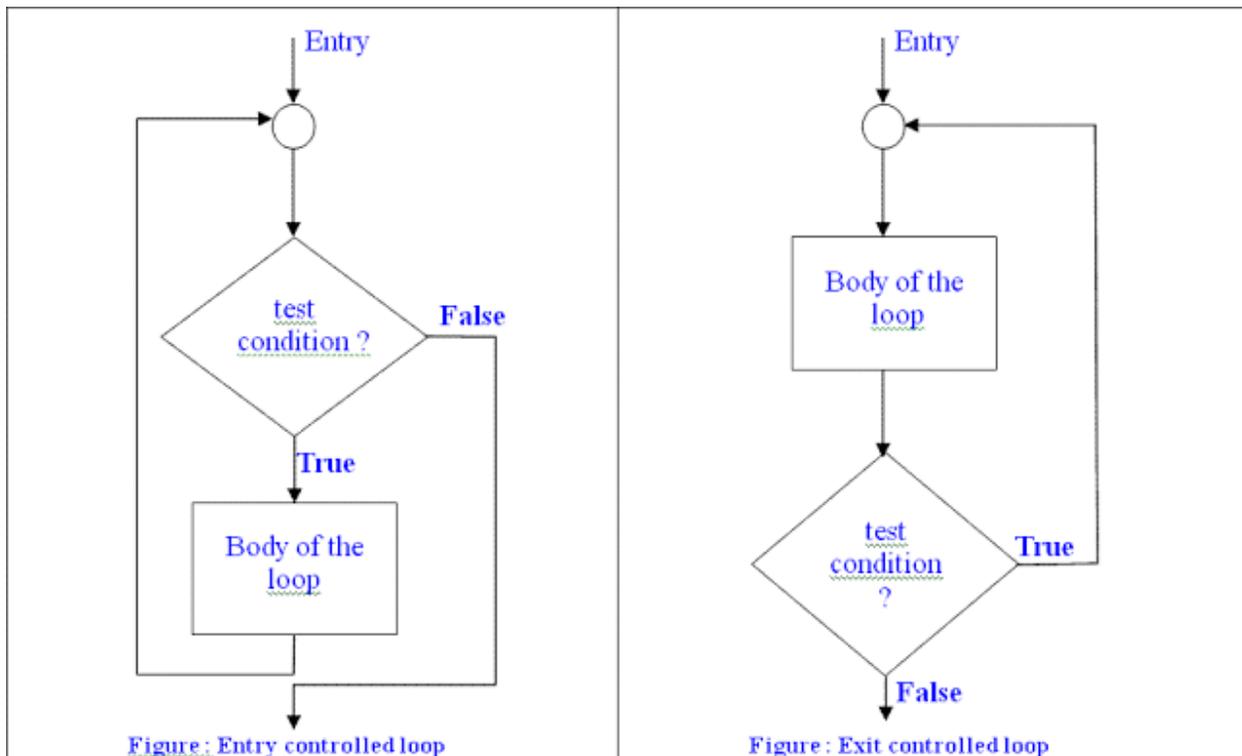
One of the classic problems encountered when we start using nested if else statements is the dangling else. This occurs when a matching else is not available for an if. The solution to this problem is, always match an else to the most recent unmatched if in the current block. In some cases, it is possible that the false condition is not required. In such situations else statement may be committed.

Decision making and looping

If the set of instructions are executed repeatedly then we call it as looping. In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the body of the loop and the

other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, the loop can be classified as entry controlled loop and exit controlled loop. In the entry controlled loop, the condition is tested, before executing loop statements. If the condition evaluates to false, then loop statements will not be executed. In exit controlled loop, the condition is tested at the end of the loop and the loop statements are executed unconditionally for the first time. The entry controlled and exit controlled loops are known as pre test and post test loops respectively.



The test conditions should be carefully stated in order to perform the desired number of loop executions. When the condition evaluates to false, the control comes out of the loop. In case, it does not do so, the control sets up an infinite loop and the body is executed over and over again.

A looping process, in general would include the following four steps.

1. Setting and initialization of a condition variable.
2. Test for a specified value of the condition variable for execution of the loop.
3. Execution of the statements in the loop
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides three constructs for performing loop operations. They are

1. The while statement
2. The do statement
3. The for statement

Sentinel loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops can be classified into 2 categories.

1. Counter controlled loops
2. Sentinel controlled loops

When we know in advance exactly how many times the loop will be executed, we can counter controlled loop. We use a control variable known as counter. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter controlled loop is sometimes called definite repetition loop.

In a sentinel controlled loop, a special value called a sentinel value is used to change the loop control expression from true to false. For example, when reading data we may indicate the “end of data” by a value , like -1 to 999. The control variable is called sentinel variable. A sentinel controlled loop is often called indefinite repetition loop because the number of repetitions is not known before the loop begins executing.

The while statement

The general syntax of while statement is

```
While(condition)
{
Body of the loop
}
```

The while is an entry controlled loop statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body the test condition is once again evaluated and if it is true, the body is executed once again. The process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

The do statement

It is an exit controlled loop. It might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. The general syntax is

```
do
{
body of the loop
}
While( condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

The for statement

Simple for loops

The for loop is another entry controlled loop that provides a more concise loop control structure. The general syntax is

```
for( initialization; condition; increment)
{
body of the loop
}
```

The execution of for statement is as follows.

1. Initialization of the control variables is done first using assignment statements.
2. The value of the control variable is tested using the test condition. The test condition determines when the loop will exit. If the condition is true, the body of the loop is executed, otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now the control variable is incremented and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of control variable fails to satisfy the test condition.

Additional features of for loop

The for loop in C has several capabilities that are not found in other loop constructs. For example more than one variable can be initialized at a time in the for statement. That is

```
for (i=0; p=1; i<10; i++)
```

The initialization section has two parts i=0, p=1 separated by comma.

Like the initialization section, the increment section may also have more than 1 part. For example

```
for (i=0; j=0; i<10; i++;j++)
```

The multiple arguments in increment section are separated by comma.

The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable. For example

```
for( i=1; i<10 && sum<20; i++)
```

The loop uses compound test condition with the counter variable i and sentinel variable sum.

It is also permissible to expressions in the assignments statements of initialization and increment sections. For example

```
For ( x=(m+n)/2; x>0; x=x/2)
```

Another unique aspect of for loop is that one or more sections can be omitted if necessary. For example

```
m =5;
for(; m!=100)
{
m=m+2;
}
```

Both the initialization and increment sections are omitted in the above for statement. The initialization has been done before the for statement and the control variable is incremented inside the for loop. In such cases, the sections are left blank. However, the semicolons separating the sections must present. If a test condition is not present, the for statement sets up an infinite loop. Such loops can be broken using break or goto statements in the loop.

We can set up time delay loops using null statements as follows.

```
for(j=1000; j>0; j=j-1)
;
```

This loop is executed 1000 times without producing any output, it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a null statement. This can be written as

```
for(j=1000; j>0; j=j-1);
```

Nesting of for loops

Nesting of loops, that is one for statement within another for statement is allowed in C. for example

```
for(i=0;i<10;i++) outer loop
{
```

```
for (k=0;k<10;k++)  inner loop
{
}
}
```

The nesting may continue up to any desired level.

Jumps in loops

Loops performs a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing say 100 names. A program loop written for reading and testing the names 100 times must be terminated as soon as the desired name is found. C permits a jump from one statement to another within a loop as well as jump out of a loop.

Jumping out of a loop

An early exit from a loop can be accomplished by using break statement or goto statement. When a break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is the break will exit only a single loop.

Since a goto statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of goto is to exit from deeply nested loops when an error occurs. A simple break statement would not work here.

1. while(...)
 {
 if (condition)
 break;
 }
2. do
 {

 if(condition)
 break;

 while(...);

3. for(...)
 {

 if (error)
 break;

 }
4. for (...)
 {

 for (...)
 {

 if(condition)
 break;

 }
5. while(...)
 {
 if (error)
 goto stop;

 if(condition)
 goto abc;

 abc:

 }
 Stop:

6. for (...)
 {

 for (...)
 {

 if (error)
 goto error;

```

....
}
.....
}
error;
....
....

```

Skipping a part of a loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for same job, we may like to exclude the processing of data of application belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the break statement, c supports another similar statement called the continue statement. However unlike the break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler, “skip the following statements and continue with the next iteration” . the syntax of continue is continue, continue;

The use of continue statement is given below :

1. while(condition)


```

      {
      if(...)
      continue;
      .....
      ....
      }
      
```
2. do


```

      {
      .....
      if(.....)
      continue;
      .....
      ....
      }
      while(condition);
      
```

```
3. for(initialization; condition; increment)
    {
    .....
    if(...)
    continue;
    ....
    ....
    }
```

Avoiding goto

It is a good practice to avoid using goto. There are many reasons for this. When goto is used, many compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable.

Jumping out of the program

We can jump out of a program by using the library function exit (). In case, due to same reason, we may wish to break out of a program and return to the operating system, we can use exit () function. For example

```
if (condition)
    exit(0);
```

The exit() function take an integer value as its argument. Normally Zero is used to indicate normal termination and a non zero value to indicate termination due to same error or abnormal condition. The use of exit() function requires the inclusion of the header file <stdlib.h>

Unit 3

ARRAYS

In a program, if we want to handle large amount of data, for example, we may need to store marks of 60 students, then we can use derived data type known as array.

An array is a fixed size, sequenced collection of elements of same data type. Arrays will avoid declaration of multiple variables to store large amount of data. So, array makes the program smaller and efficient.

One dimensional arrays:

A list of items can be given one variable name by specifying a size and such a variable is called single subscripted variable or a one dimensional array.

Declaration of one dimensional array:

Like any other variable, arrays must be declared before they are used so that compiler can allocate memory for them. The general syntax of array declaration is,

type variable_name[size];

The type specifies the type of data that can be stored in an array such as int, float, char etc. The size indicates the maximum number of data that can be stored in an array. For example,

float height[50];

declares height to be an array can contain maximum of 50 real values.

Every value in an array can be called as **element**. Every element in an array is referred by a number called **index**. Always index of array starts with **0**.

That is always the first element of the array have the index value 0 and last element of the array have the index value size-1.

Similarly,

Int marks[10];

declares marks as an array to contain a maximum of 10 integer values.

If we try to store more number of elements than size of array would not necessarily cause an error. Rather it might result in unpredictable program results. The size should always be a integer value.

char name[10];

declares name as character array that can hold maximum of 9 characters. When the compiler sees character array, it terminates it with a null character (10). So while declaring character arrays. We must allow one extra element space for null character.

Initialization of one dimensional arrays:

After an array is declared, it must be initialized, otherwise it will contain garbage values. An array can be initialized at either of the following stages.

- a) at compile time
- b) at run time

Compile time initialization:

We can initialize the elements of the array in the same way as ordinary variables when they are declared. The general syntax of initialization of array is,

type array_name[size]={list of values};

The values in the list are separated by commas. For example,

int arr[3]={0, 0, 0}

arr		
0	0	0
0	1	2

Will declare the variable arr as array of size 3 and will assign 0 to each memory location. If the number of values in the list are less than the size, they only that many values will be initialized. The remaining memory locations will be initialized to zero automatically. For example,

int a[5]={3, 4}

a				
3	4	0	0	0
0	1	2	3	4

will initialize first two memory locations to 3 and 4, remaining 3 memory locations will be initialized to 0. If we initialize array at compile time, then it is not mandatory to mention size of array. For example,

int x[]={2, 3, 9, 10}

will declare x as array with size 4. The compiler will find size of array by looking at number of values. Character arrays may be initialized in a similar manner. For example,

char s[4]="4i";

s			
4	I	\0	\0
0	1	2	3

will declare s as array with size 4. First two memory locations will be initialized to 4 and I, remaining memory locations will be initialized to null character.

The character array can also be initialized as,

char s[4]={'4', 'I'}

Similarly,

char s1[]="Hello";

will declare s1 as array with size 6, where last memory location will be initialized to null character.

If we initialize more values than the specified size, the compiler will produce an error.

For example,

```
int a[3]={2, 5, 9, 10}
```

will produce error.

Run time initialization:

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example,

```
float sum [100];
for (i=0; i100; i++)
{
    If(i50)
        sum[i]=0.0;
    else
        sum[i]=1.0;
```

The first 50 elements of array sum will be initialized to zero and remaining 50 elements will be initialized to 1.0 at run time (execution time).

We can also use scanf function to initialize an array. For example,

```
int a [3];
scanf("%d%d%d", &a[0], &a[1], &a[2]);
(or)
int a[3];
for(i=0; i23; i++)
{
    Scanf("%d", & a[i]);
}
```

*During run time initialization, empty memory locations will be initialized to garbage values.

Searching and sorting:

Searching and sorting are the two most frequent operations performed on arrays.

Sorting is the process of arranging elements according to their values, either in ascending or descending order. Many sorting techniques are available. The simple and most important among them are,

1. bubble sort

2. selection sort
3. insertion sort

searching is the process of finding the index of specified element in array. The specified element is often called the search key. If we are able to find search key in an array, then the search said to be successful, otherwise it is unsuccessful. The two most commonly used search techniques are,

1. Linear search (sequential search)
2. Binary search

Two dimensional arrays:

There could be situations where a table of values will have to be stored. For example,

	Test 1	Test 2	Quiz 1	Quiz 2
Rahul	48	45	13	12
Ramya	49	48	14	13
Radika	40	42	11	14

The above table shows the marks scored by students in tests and quizzes. We can think of this table as a matrix consisting of rows and columns. Each row represents marks scored by a particular student. C allows us to define such tables of values by using two dimensional array.

The general syntax to declare two dimensional array is,

Type array_name[rows][columns];

For example,

int a[10] [5];

declares a as array with maximum of 10 rows and 5 columns.

Initialization of two dimensional arrays:

Like the one dimensional array, two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

int x[2][3] = {0, 0, 0, 1, 1, 1};

initializes the elements of first row to zero and second row to one. The initialization is done row by row.

a		
00	01	02
0	0	0
10	11	12
1	1	1

The index of two dimensional array will be having two digits. First digit of index specifies row number and second digit of index specifies column number.

The above initialization statement can also be written as,

```
int x[2][3]={{0, 0, 0}, {1, 1, 1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two dimensional array in the form of a matrix as shown below.

```
int x[2][3]={
                {0, 0, 0},
                {1, 1, 1}
            };
```

Permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
int x[2][3]={
                {1, 1},
                {2}
            };
```

Will initialize the first two elements of the first row to one, the first element of second row to two and all other elements to zero.

When all elements are to be initialized to zero, the following short cut method may be used.

```
int m[3][5]={{0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero. While other elements are automatically initialized to zero. The following statement will also achieve the same result.

```
int m[3][5]={0, 0};
```

Memory layout:

The elements of array are stored contiguously in increasing memory locations, essentially in a single list. If we consider memory as row of bytes, with the lowest address on the left and highest address on the right, a simple array will be stored in memory with the first element at left end and the last element at the right end. Similarly, a two dimensional array is stored “row-wise starting from the first row and ending with the last row, treating each row like a simple array. For example,

```
int arr[3][3]={{10, 20, 30}, {40, 50, 60}, {70, 80, 90}};
```

	arr		
	00	01	02
0	10	20	30
1	40	50	60
2	70	80	90

Will be stored in memory as,

row 0			row 1			row 2		
10	20	30	40	50	60	70	80	90
00	01	02	10	11	12	20	21	22

Character arrays and strings:

A string is a sequence of characters that is treated as a single data item. A group of characters defined between double quotes is a string constant. For example,

“Hello”

The string constant can be printed as output using printf() function. For example,

```
Printf(“Hello world”);
```

Will display

Hello world

If we want to print string with double quotes, then we need to use backslash (\). That is,

```
Printf(“\” Hello world \””);
```

will print

“Hello world”

Declaring and initializing string variables:

C does not support string as a data type. However, it allows us to represent strings as character arrays. In C, therefore a string variable is any valid C variable name used is always declared as an array of characters. The general syntax of declaration of string variable is,

```
Char string_name[size];
```

The size determines the maximum number of characters can be stored in array. Some examples are,

```
Char city[10];
```

```
Char name[25];
```

When compiler sees character array, it automatically assigns null character at the end. Therefore, the size of string (character array) should be equal to maximum number of characters in string plus one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms.

```
Char city[9]= "New York";
Char city[9]={ 'N', 'e', 'w', ' ', 'Y', 'o', 'r', 'k' };
```

The reason that the size of city has to be 9 is that the sting New York contains 8 characters and one last memory location is for null character.

C also allows us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example,

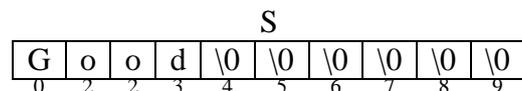
```
Char s[ ]={ 'G', 'O', 'O', 'D' };
```

defines the array s with size 5.

We can also declare the size much larger than the number of characters. That is,

```
Char s[10]= "Good";
```

is allowed. In this case, the computer creates a character and of size 10, places the value "Good", in it, terminates it with the null character, and initializes all other elements to NULL. That is,



However, the following declaration is illegal.

```
Char s[2]= "Good";
```

This will result in compile time error. Also we cannot separate initialization form declaration. That is,

```
Char s[5];
S = "Good";
```

is not allowed.

Why do we need null character at end of string:

As we know, string is not a data type in C. the character array size is not always the size of the string and most often it is much larger than the string stored in

it. Therefore, the last element of the array need not represent the end of string. So, we need some way to determine the end of the string data and the null character serves as the “end of string” marker.

Reading strings from terminal:

Using scanf function:

The familiar input function scanf can be used with %S format specifier to read a string. For example,

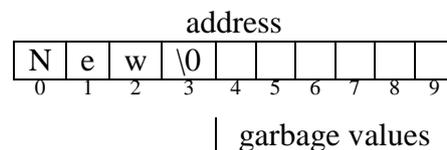
```
Char address [10];  
Scanf(“%S”, address);
```

While reading string, we should not use compressed (&) symbol before character array name, because array name itself points to address of fist element of array.

The problem with the scanf function is that it terminates the reading when it finds space. A space includes blank space, new line, tabs, carriage return etc. therefore, if we give input from keyboard as,

New York

Then only the word New will be read into the array address, since scanf stops reading when it finds space in input. The scanf function automatically assigns null character at the end and therefore the character array should be large enough to held the input string plus the null character. That is,



The remaining memory locations will be filled with garbage values.

If we want to read entire input New York, then we need two character arrays of appropriate sizes. That is,

```
Char adr1[5], adr2[5];  
Scanf(“%s%s”, adr1, adr2);
```

Will assign string New to adr1 and York to adr2.

We can also specify the field width in the scanf statement for reading a specified number of characters from the input string. That is,

```
Scanf(“%ws”, array_name);
```

If the width w is equal to or greater than the number of characters in input, then entire string will be read and stored to array variable.

If the width is less than the number of characters in the string, the excess characters will not be read.

Consider the following statements,

```
Char name [10];
```

```
Scanf ("%s", name)
```

If we give input as RAM, then it is stored as,

R	A	M	\0						
0	1	2	3	4	5	6	7	8	9

garbage values

If we give input as college, then it will read only 5 characters.

c	o	l	l	e	\0				
0	1	2	3	4	5	6	7	8	9

garbage values

Reading a line of text:

The scanf() function with %s or %ws can read only strings without white spaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as edit set conversion code %[..] that can be used to read a line containing a variety of characters including whitespaces. For example,

```
Char line [80];
```

```
Scanf ("%[\n]", line);
```

```
Printf ("%s", line);
```

Will read a line of input from keyboard and display the same on the output.

Using getchar and gets functions:

Getchar() function can be used to read single character from keyboard. The syntax to use getchar function is,

```
Char ch;
```

```
Ch=getchar ( );
```

Note that getchar() function has no parameters. The getchar and gets function are defined in header file.

```
<Stdio.h>
```

We can use getchar function repeatedly to read successive single character from input and store it into a character array. Thus, the entire line of text can be read and stored in an array. The reading is terminated when the new line character(\n) is entered.


```
Printf(“%*.s”, w, d, array_name);
```

Prints the first d characters of array in the width w. For example,

```
Printf(“%*.s”, 10, 4, “Hello world”);
```

Will print

						H	e	l	l
--	--	--	--	--	--	---	---	---	---

Using putchar and puts functions:

Putchar function is used to print single character on output screen. The syntax to use putchar function is,

```
Char ch='a';
```

```
Putchar(ch);
```

It prints the value of ch on output. The above statement is equivalent to,

```
Printf(“%c”, ch);
```

We can use putchar() function repeatedly to print a string of characters stored in an array using a loop. For example,

```
Char name [6] = “Hello”;
```

```
for (i=0; i<5; i++)
```

```
Putchar(name[i]);
```

The putchar and puts functions are present in header file <Stdio.h>

Another more convenient way of printing values is to use the function puts. The syntax to use puts function is,

```
Char name[10];
```

```
Gets(name);
```

```
Puts(name);
```

reads a line of text from keyboard and prints on screen.

Arithmetic operations on characters:

C allows us to manipulate characters the same way we do with numbers. Whenever a character is used in an expression, it will automatically be converted into an integer value (ASCII value) by the system.

```
Char x= 'a';
```

```
Printf(“%d”, x);
```

will display the output as 97.

It is also possible to perform arithmetic operations on characters. For example,

```
int x= 'z' - 1;
```

The ASCII value of z is 122 and therefore it will assign 121 to x.

We can also use character constants in relational expression. For example,

```
if(ch>= 'A' && ch<= 'z')
```

Would test whether the character contained in variable ch is an upper case alphabet or not.

We can convert a character digit to its equivalent integer value using the following statement.

```
int x = character - '0';
```

Where character contains character digit. For example, let us assume character contains the digit '7'. Then,

X=ASCII value of '7' – ASCII value of '0'

= 55 – 48

= 7

The C library supports a function that converts a string of digits into their integer values. The syntax is,

```
int x=atoi(string);
```

Where string is a character array containing a string of digits. For example,

```
char number [5] = "1988";
```

```
int year = atoi(number);
```

The function atoi converts the string "1988" to its numeric equivalent 1988 and assigns it to the integer variable year. The atoi() function is present in the header file stdlib.h

String handling functions:

C supports a large number of string handling functions that can be used to carry out many of the string handling functions are,

Function	Action
Strcat()	Concatenates two strings
Strcmp()	Compares two strings
Strcpy()	Copies one string to another
Strlen()	Finds the length of string
Strrev()	Reverse the string

Strcat() function:

The strcat() function joins two strings together. The syntax will be,

Strcat (string1, string2);

String1 and string2 are character arrays. When the function strcat is executed, string2 is appended to string1. It does so removing null character at the end of string1 and placing string2 from there. The string2 remains unchanged. For example,

Char s1[12]= "Hello";	h	e	l	l	o	\0	\0	\0	\0	\0	\0	\0	
Char s2[6]= "world";	w	o	r	l	d	\0							
printf("%10s", "united states");	u	n	i	t	e	d		s	t	a	t	e	s

Strcat function can also append string constant to a string variable. For example,

Strcat(s1, "Good");

C permits nesting of strcat functions. For example

Strcat(strcat(string1, string2), string3); is allowed and concatenates all three strings together. The resultant string is stored in string1.

Strcmp() function:

The strcmp() function compares two strings and returns a value 0 if they are equal, returns positive value if first string is greater than second one or returns a negative value if string1 is less than string2. The syntax is,

Strcmp(string1, string2);

String1 and string2 are string variables or string constants. For example,

Strcmp(name1, name2);

Strcmp(name1, "anu");

Strcmp("anu", "asha");

For example,

Strcmp("Hi", "Hi")

Will return value 0, because both strings are equal.

Strcmp("Ram", "Ram");

Will return -14. That is it will return numeric difference first non-matching characters in the strings ASCII value of 'a' – ASCII value of '0'.

97 - 111

- 14

Strcmp("has", "as");

Will return 7 that is,

ASCII value of 'h' – ASCII value of 'a'

104 – 97

7

Strcpy() function:

The syntax will be,

```
Strcpy(string1, string2);
```

Will assigns the contents of string2 to sting1. String2 may be a character array variable or a sting constant. For example,

```
Strcpy (city, "Delhi");
```

Will assign the sting "Delhi" to the string variable city similarly,

```
Strcpy(city1, city2);
```

Will assign the contents of the sting variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2.

Strlen() function:

This function counts and returns the number of characters in a string. The sytax is,

```
Int n=strlen(string);
```

N receives the value of length of string. The counting ends at first null character. For example,

```
Char s[10]= "Hello";
```

```
Int n = strlen(s);
```

Here, n will receive the value 5.

Strrev() function:

The strrev() function reverses the contents of string variable. The syntax is,

```
Strrev (string1);
```

For example,

```
Char s[6]= "Hello";
```

```
Strrev(s);
```

After reversing, s contains "olleH".

POINTERS

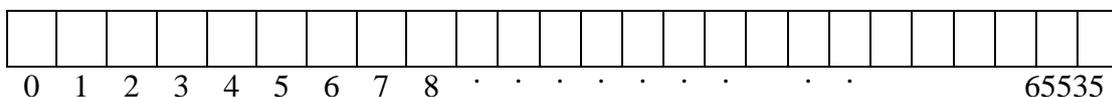
A pointer is a derived data type in C. pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They are,

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers reduce length and complexity of programs.
4. They increase the execution speed and thus reduce the program execution time.

Understanding Pointers:

The computer's memory is a sequential collection of storage cells. Each cell, commonly known as a byte, has a number called address associated with it. Typically the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64k memory will have its last address as 65,535.

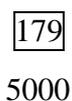


That is $64 \times 1024 = 65536$

Whenever we declare a variable, the system allocates somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number consider the following statement.

```
int quantity=179;
```

This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for quantity. That is

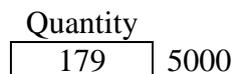


During execution of the program, the system always associates the name quantity with the address 5000. We may have access to the value 179 by using either

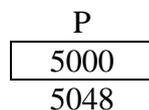
the name quantity or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables. A pointer variable, is therefore nothing but a variable that contains an address, which is a location of another variable in memory.

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of quantity to a variable P.

The link between the variables P and quantity can be visualized as below.



Since the value of the variable P is the address of the variable quantity, we may access the value of quantity by using the value of P and therefore we say that the variable P points to the variable quantity.



Pointers are built on three concepts.

1. Pointer constants
2. Pointer values
3. Pointer variables

Memory addresses within a computer are referred to as pointer constants. We cannot change them, we can only use them to store data values.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The values thus obtained is known as pointer value.

Once we have pointer value, it can be stored into another variable. The variable that contains a pointer value is called a pointer variable.

Accessing the address of a variable:

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. The address of a variable can be determined with the help of address operator &. For example,

$$P = \& \text{quantity}$$

Would assign the address of variable quantity to the variable P.

Declaring Pointer Variables:

The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things.

1. The * tells the variable Pt_name is a pointer variable.
2. Pt_name needs a memory location.
3. Pt_name points to a variable of type data_type.

For example,

```
Int *P;
```

declares the variable P as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed to by P and not the type of the value of the pointer. Similarly the statement.

```
Float *x;
```

declares x as a pointer to a floating point variable. The declaration cause the compiler to allocate memory location for the pointer variable. If the pointer variables have not been initialized, they contain garbage values.

Pointer variables are declared similarly as normal variables except for * operator. This symbol can appear anywhere between the type name and pointer variable name. That is,

1. int* p;
2. int *p;
3. int * p;

the syntax 2 is popular because this style is convenient to have multiple declarations in the same statement. For ex.,

```
int *p, x, *q;
```

Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as initialization. Once a pointer variable has been declared we can use the assignment operator to initialize the variable. For exp.,

```
int quantity;  
int *p;  
P=&quantity;
```

We can also combine the initialization with the declaration. That is,

```
Int quantity;
```

```
Int *P = &quantity;
```

We must ensure that the pointer variables always point to the corresponding type of data. For ex.,

```
float a, b;  
Int x, *p;  
p=&a; /*error */  
b=*P;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of a variable, the declaration of pointer and the initialization of pointer. For ex.

```
int x, *p=&x;
```

is valid. It declares x as an integer variable and P as a pointer and then initializes P to the address of x. and also remember x is declared first. The statement,

```
int *P=&x, x;
```

is not valid.

We could also define a pointer variable with an initial value of NULL or Zero. That is,

```
int *p=NULL;  
int *p=0;
```

Pointers are flexible. We can make the same pointer to point to different data variables in different statements. For ex.,

```
int x, y, z, *p;  
P =&x;  
P =&y;  
P =&z;
```

We can also use different pointers to point to the same variable. For ex

```
int x;  
int *P1=&x;  
int *P2=&x;  
int *P3=&x;
```

Accessing a variable through its pointer:

Once a pointer has been assigned the address of a variable, the value of a variable can be accessed using the indirection operator (*). Another name for indirection operator is dereferencing operator. For example,

```
Int quantity, *P, n;  
Quantity =179;  
P=&quantity;  
N=*P;
```

The first line declares quantity and n as integer variables and P as a pointer pointing to an integer. The second line assigns the value 179 to quantity and the third line assigns the address of quantity to the pointer P. in fourth line, *P gives the value of variable quantity and assigns to n. thus the value of n would be 179. The two statements,

```
P = &quantity;  
n = *P;
```

Are equivalent to

```
n= *&quantity;
```

Which in turn equivalent to

```
n= quantity;
```

Pointer Expressions:

Like other variables, pointers can be used in expressions. For ex,

```
Int a=10, b=20, *P1, sum, z; y;  
P1=&a;  
P2=&b;  
y=*P1 * *P2;  
sum= sum+ *P1;  
*P2= *P2+10;  
Z=5* - *P2/ *P1;
```

In the above line, there is a space between / and * otherwise compiler will give error because /* is considered as beginning of comment.

C allows us to add integers or subtract one pointer from another. P1+4, P2-2 and P1-P2 are allowed. If P1 and P2 are both pointers to same array, the P2-P1 gives the number of elements between P1 and P2.

We can also use shortant operators with pointers,

```
P1++;
```

-P2;

Sum+=*P2;

In addition to arithmetic operations, pointers can also be compared using relational operators. The expressions such as P1<P2, P1==P2 and P1!=P2 are allowed. We cannot use pointers in division or multiplication. For example,

P1/P2

P1*P2

P1/3

Are not allowed. Similarly two pointers cannot be added. That is P1+P2 is not allowed.

Pointer increments and scale factor:

The pointers can be incremented like

P1=P2+2;

P1=P1+1;

However, an expression like,

P1++;

Will make the pointer P1 to point to next value of its type. For ex. If P1 is an integer pointer with an initial value 2800, then after the operation P1=P1+1, the value of P1 will be 2804, and not 2801. That is when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the scale factor.

Pointers and arrays:

When an array is declared, the compiler allocates sufficient amount of memory to contain all elements of array in contiguous memory locations. For ex.

int x[5]={1, 2, 3, 4, 5};

Suppose the initial address (base address) of x is 1000 and assuming that integer needs 5 bytes, the elements are stored as,

x				
0	1	2	3	4
1	2	3	4	5
1000	1004	1008	1012	1016

The name of array x always contain starting address 1000.

That is,

X=&x[0]=1000;

If we declare P as an integer pointer, then we can make pointer P to point to the array x. That is,

```
P=x;
```

This is same as

```
P=&x[0];
```

The address of element of an array can be calculated using index and its starting address. That is,

```
address of x[3] =initial address+(3xscale factor of int)
                =1000+(3x4)
                =1012
```

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that *(P+3) gives the value of x[3].

STRUCTURES

C supports a user defined data type known as structure, a mechanism for packing data of different types. A structure can be defined as collection of variables of same type or of different type.

Arrays Vs Structures:

1. An array is a collection of related data elements of same type. Structure can have elements of different type.
2. An array is derived data type whereas a structure is a user defined type.

Defining a structure:

The general syntax of structure definition is

```
Struct tag_name
{
    datatype var1;
    datatype var2;
    .....
};
```

Consider a book consisting of book_name, author, number of pages and price, book structure can be defined as

```
Struct book
{
    Char book_name[20];
    Char author [30];
```

```

    Int number_of_pages;
    Float price;
};

```

The keyword struct declares a structure to hold the details of four data fields, namely book name, author, number of pages and price. These fields are called structure elements or members. Each element may be of different type. The name of structure is called tag_name.

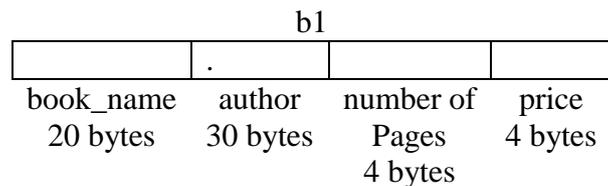
Declaring Structure Variables:

After defining a structure, we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. For ex,

```
Struct book b1, b2, b3;
```

Declares b1, b2, b3 as variables of struct book.

When we define a structure, memory will not be allocated. But when we declare a variable of structure, memory will get allocated. For example, memory allocation of b1 is



The use of tag name is optional. That is,

The use of tag name is optional,

```

Struct
{
    Char book_name[20];
    Char title[30];
    Int number_of_pages;
    Float price;
} b1, b2, b3;

```

We can use the keyword typedef to define a structure.

```

Typedef struct
{
    Char book_name[20];
    Char title[30];
    Int number_of_pages;
}

```

```
Float price;  
};
```

Accession Structure Members:

We can access and assign values to the members of a structure. The members themselves are not independent variables. They should be linked to the structure variable in order to make them meaningful members. For example, the word author has no meaning, whereas author of book1 has meaning. The link between member and structure variable is established using the member operator. Which is also known as dot operator or period operator. For example,

```
b1-author
```

is the variable representing author of b1.

Structure Initialization:

A structure variable can be initialized both at compile time and run time.

Compile Time Initialization:

```
Struct complex  
{  
    int real;  
    int imag;  
}c1={3,4};
```

Will assigns 3 to real and 4 to imag. There is a one to one correspondence between members and their values.

In the same way structure can also be initialized as,

```
Struct complex  
{  
    int real;  
    int imag;  
};  
Struct complex c1={2, 5};  
Struct complex c2={1, 8};
```

We can also initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the right side. The uninitialized members will

be assigned with default values. That is zero for integer and float variables, 1 0 for characters and strings. For example,

```
Struct complex
{
    Int real;
    Int complex;
}c1={2};
```

Will assign 2 to real and default value 0 to imag.

Run time initialization:

The structure variable can be initialized at run time assign scanf() function.

For example,

```
Scanf(“%d%d”,&c1.real, &c1.imag);
```

Copying and comparing structure variables:

Two variables of the same structure can be copied the same way as ordinary variables. If c1 and c2 are variables of structure complex, then the following statement is valid.

```
c1=c2;
```

However,

```
c1!=c2;
```

are not valid. C does not permit any logical operations on structure variables.

Word boundaries and slack bytes:

Computers stores structures using the concept of word boundary. The size of a word boundary is machine dependent. In a computer with two bytes word boundary, the members of a structure are stored left aligned on the word boundary. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as slack byte.

1	2	3	4	
Char	Slack Byte	int		

When we declare structure variables, they may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal. Therefore C does not permit comparison of structure variables directly.

Operations on individual members:

The individual members are identified using the member operator, the dot. A member with the dot operator along with the structure variable can be treated like any other variable and therefore can be manipulated using expressions and operators. For example,

```
C3.real=c1.real+c2.real;
```

File management in C

We can use the functions such as scanf() and printf() to read and write data. These are a console oriented I/o functions, which always use the terminal (Keyboard and monitor) as the target place. This works fine as long as the data is small. However, many real life problems involve large volumes of data and in such situations console oriented I/o operations pose two major problems.

1. It becomes time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, c supports a number of functions that have the ability to perform basic file operations, which include

1. Naming a file
2. Opening a file
3. Reading data from a file
4. Writing data to a file
5. Closing a file

There are two distinct ways to perform file operations in C. the first one known as the low level I/o and uses unix system calls. The second method is referred to as the high level I/o operation and uses functions in c standard I/o library.

Defining and opening a file:

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system.

They include,

1. File name
2. Data structure
3. Purpose

Filename is a string of characters that make up a valid file name for the operating system. It may contain two parts, a primary name and an optional period with the extension. For example,

Input.data

Sotre

Prog.c

Text.out

All files should be declared as FILE before they are used. FILE is a defined data type. When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

The general syntax for dedclaring and opening a file is,

```
FILE *fp;
```

```
Fp=fopen("filename", "mode");
```

The first statement declares the variable fp as a pointer of type FILE. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode will do this job. Mode can be one of the following;

r - Open the file for reading only

w- Open the file for writing only

a- Open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen;

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exist.
2. When the mode is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the mode is 'reading' and if it exists, then the file is opened with the current contents safe, otherwise an error occurs. Consider the following statements.

```
FILE *p1, *p2;
```

```
P1=fopen("data", "r");
```

```
P2=fopen("results", "w");
```

The file data is opened for reading and results is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If data file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They are,

rt- The existing file is opened to the beginning for both reading and writing.

wt- same as w except both for reading and writing.

at- same as a except both for reading and writing.

Closing a file:

A file must be closed as soon as all operations on it have been completed. This ensures that all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files might help one the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/o library supports a function to do this for us. The general syntax is,

```
fclose(file_pointer);
```

This would close the file associated with file_pointer. For ex,

```
FILE *p1, *p2;  
P1=fopen("input", "w");  
P2=fopen("output", "r");  
.....  
.....  
fclose(p1);  
fclose(p2);
```

The above code segment opens two files and closes them after all operations one them are completed. Once a file is closed, its file pointer can be raised for another file.

All files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

Input/output operations on files:

getc and putc functions:

The simplest file I/o functions are getc and putc. Assume that a file is opened with mode w and file pointer fp1. Then, the statement,

```
putc(c, fp1);
```

Writes the character contained in the character variable `c` to the file associated with pointer `fp1`. Similarly, `getc` is used to read a character from a file that has been opened in read mode. For example,

```
C=getc(fp1);
```

Would read a character from the file whose file pointer is `fp1`. The file pointer moves by one character position for every operation of `getc` or `putc`. The `getc` will return an end of file marker `EoF`, when end of the file has been reached. Therefore, the reading should be terminated when `EoF` is encountered.

getw and putw functions:

The `getw` and `putw` are integer oriented functions. They are similar to the `getc` and `putc` functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general syntax of `getw` and `putw` functions are,

```
putw(a, fp);
```

```
b=getw(fp);
```

`putw` will write data contained in `a` to a file pointed to by `fp`. `getw` will read integer value from file pointed to by `fp` and store in `b`.

fprintf and fscanf functions:

`fprintf` and `fscanf` can handle a group of mixed data simultaneously. The functions `fprintf` and `fscanf` perform I/o operations that are identical to the familiar `printf` and `scanf` functions, except they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general syntax of `fprintf` is,

```
fprintf(fp, "control string", variables);
```

where `fp` is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for the items in the list. The variables is list of variables where data is present. For example,

```
fprintf(fp, "%s%d%f", name, age, 7.5);
```

here, `name` is an array variable of type `char` and `age` is an `int` variable.

The general syntax of `fscanf` is

```
fscanf(fp, "control string", variables);
```

This statement would cause the reading of the items in the variables from the specified by fp, according to the specifications contained in the control string. For example,

```
Fscanf (fp, "%s%d", item, &quantity);
```

Like scanf, fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

UNIT 4

USER DEFINED FUNCTIONS

One of the strengths of C language is functions. Function is a set of instructions which will do a particular task. C functions are classified into two categories, namely Library Functions and User Defined Functions. The Library functions are not required to be written by us whereas a user defined function has to be developed by the user at the time of writing a program. The examples of library functions are printf, scanf, cos, sqrt etc.

Need for user defined functions:

Main is specially recognized function in C. every program must have a main function to indicate where the program has to begin its execution. While it is possible to write any program with only single function main, but it leads to number of problems. The program may become too large and complex and as a result the task of debugging, testing and maintaining becomes difficult. If a program is divided into parts, then for each part we can write program independently and later combined into a single unit. These independently written programs are called subprograms that are much easier to understand, debug and test. In C, such subprograms are referred to as functions.

There are time when certain type of operations or calculations are repeated at many points throughout a program. For example, we might the factorial of a number at several places in the program. In such situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both the time and space.

This division of problems into subproblems has many advantages. They are

1. It provides top down approach. In this programming style, the high-level logic of the overall problem is solved first while the details of each lower level function are addressed later,
2. The length of program can be reduced.
3. It is easy to identify and debug errors.
4. A function may be used by many other programs. This means that a C Programmer can use the functions which are written by others to write his program.

Main Program

Function A

Function B

Function C

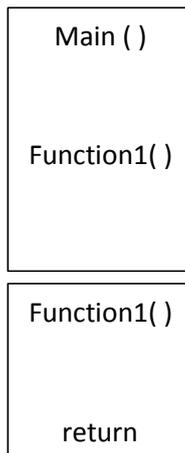
Top down approach

A Multi-Function Program:

Once a function has been written, it can be treated as a black box that takes some data from the main program and returns a value. The inner details of operation are invisible to rest of program. All that the program knows about a function is what goes in and what comes out. Every C program can be designed using a collection of these block boxes known as functions.

Any function can call any other function. In fact, it can call itself. A called function can also call another function. A function can be called more than once.

Except the starting point, there are no other rules of precedence or hierarchies among the functions of a program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is usual practice to put all called functions at the end.



Flow of control in multi-function program

Some characteristics of modular programming are,

1. Each module/function should do a single task.
2. Communication between functions can be established using functions calls and returning values.
3. No communication take place between functions if there is no function call.
4. All functions will have single entry and exit point.

Elements of user defined functions:

In order to make use of a user defined function. We need to write three things that are related to functions.

1. Function definition
2. Function call
3. Function declaration

The function definition is an independent program module that is written to implement the requirements of a function. In order to use this function, we need to invoke it at a required place in the program. This is known as the function call. The function which calls another function is called calling function. The function which is invoked by other function is known as called function. The program should

declare any function that need to be call later. This is known as function declaration or function prototype.

Definition of functions:

A function definition also known as function implementation, should include the following elements.

1. Function name
2. Function type
3. List of parameters
4. Local variable declarations
5. Function statements
6. A return statement

All six elements are grouped into two parts. They are,

1. Function header [combination of function name, function type and list of parameters].
2. Function body [combination of local variable declarations, function statements and a return statement]

A general syntax & function definition is,

```
Function type    function name [Parameter List]
{
    Local variable declaration;
    Statement 1;
    Statement 2;
    ..... Statement n;
    Return statement;
}
```

Function Header:

The function header consists of three parts, the function type also known as return type, the function name and the formal parameter list. Semicolon should not be used at the end of function header.

The function type specifies the type of value that the function is expected to return to calling function. If the return type is not explicitly specified, C will assume it as integer type. If the function is not returning anything, then we need to specify the return type as void. The value returned is the output produced by the function.

The function name is any valid C identifier and therefore must follow the same rules of formation as other variables in C. the names should be appropriate to the task performed by the function. Library function names should not be used as user defined function names.

The parameters list declares the variables that will receive the data sent by the calling function. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are called as **Formal Parameters**. The parameters are also known as **Arguments**.

The parameter list contains declaration of variables separated by commas and surrounded by parenthesis. For example,

```
float quadratic (int a, int b, int c)
```

```
double power (double x, int n)
```

```
float mul (float x, float y)
```

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

Note that the declaration of parameter variables cannot be combined, that is,

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

is not valid.

A function need not always receive data from calling function. This can be indicated by empty parenthesis. That is,

```
int sum ( )
```

the function neither receives any data not returns any value. This can be indicated as

```
void sum ( )
```

Function Body:

The function body contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts in the order given below.

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A return statement that returns the value evaluated by the function.

If a function does not return any value, then it is not required to write return statement and return type should be specified as void.

Return values and their types:

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. While it is possible to pass any number of values from calling function to called function, but called function is able to return only one value at a time.

The general syntax of return statement is

```
return ;
```

or

```
return expression
```

The first return does not return any value, it meant to return control of execution. An example of second return statement is,

```
return p;
```

returns the value of p.

A function may have more than one return statements. This situation arises when the value returned is based on certain conditions. For example,

```
    if (x<0)
        return 0;
    else
        return 1;
```

Function Calls:

A function can be called by simply using the function name followed by a list of variable names. The variables names or values in function call are called actual arguments (or) actual parameters.

When a compiler encounters a function call, the control of execution will get transfer to called function. That function is then executed line by line as defined and a value will be returned when it encounters a return statement.

They are many different ways to call a function. They are,

```
mul (10, 5);
mul (n, 5);
mul (10, n);
mul (m, n);
mul (m+5, n);
```

However, a function cannot be used on the left side of assignment statement. That is,

```
mul (a, b)=15
```

is invalid.

Function declaration:

Like variables, all functions in a C program must be declared, before they are called. A function declaration (also known as function prototype) consists of four parts.

1. Function type (return type)
2. Function name
3. Parameter list
4. Terminating semicolon

The general syntax of function declaration is,

```
function_type function_name (parameter list);
```

This is very similar to function header line except the terminating semicolon.

Rules to be followed while using functions:

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition in number and order with the parameters in function call and function declaration.
4. Use of parameter names in function declaration is optional.
5. The return type is optional, if function returns int type data.
6. The return type must be void, when the function does not return a value.
7. When the data types of variables in function declaration and function definition does not match, they compiler will give an error.

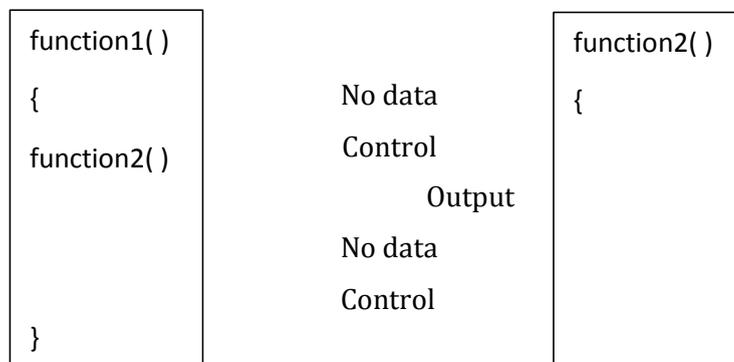
Category of functions:

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

1. Functions without arguments and without return value.
2. Functions with arguments and without return value.
3. Functions without arguments and with return value.
4. Functions with arguments and with return value.
5. Functions that return multiple values.

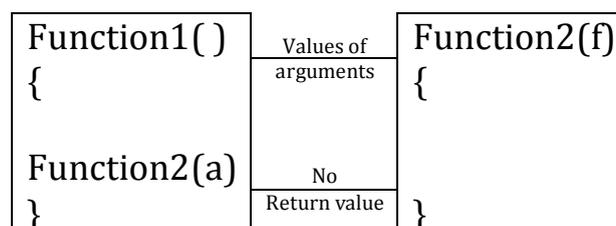
Functions without arguments and without return value:

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function. Then, there is no data transfer between calling function and called function. This is illustrated in the diagram below.



Functions with arguments and without return value:

The nature of data communication between the calling function and the called function is illustrated below.



The actual and formal arguments should match in number, type and order. The values of actual arguments are assigned to the formal arguments on a one to one basis starting with the first argument.

We should ensure that the function call has matching arguments. In case, the actual arguments are more than the formal arguments, the extra actual arguments are discarded. On the other hand, if the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. No error messages will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions or constants. The variables used in actual arguments must be assigned values before the function call is made.

When a function call is made, only a copy of the values of actual arguments is passed into the called function. What occurs inside the function will have no effect on the variables used in the actual arguments list.

Variable number of arguments:

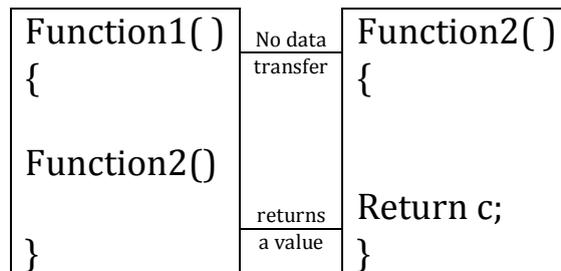
Some functions have a variable number of arguments and data types which cannot be known at compile time. The printf and scanf functions are typical examples. The ANSI standard proposes new symbol called the ellipsis to handle such functions. The ellipsis consists of three periods (...) and used as shown below.

Double area (float d, ...)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and type.

Functions without arguments and with return value:

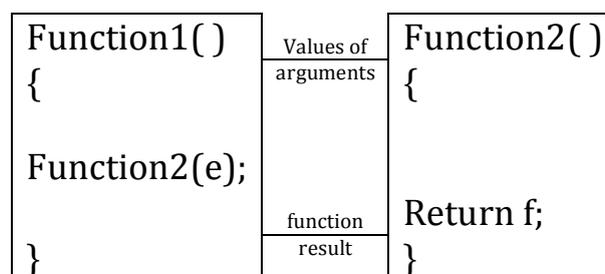
There could be a situation where we may need to design functions that may not take any arguments but returns a value to the calling function. The communication between calling function and called function is given below.



Functions with arguments and with return value:

To assure a high degree of portability between programs, a function should generally be written without involving any I/O operations. For example, different programs may require different output formats for display of results these shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like block box that receives a predefined form of input and outputs a desired value. Such functions will have two way data communication as shown below.



Nesting of functions:

Nesting of functions is nothing but one function class another function. C permits nesting of functions. Main () function can call function1, which can call function2, function2 can call function3 and so on. There is no limit as to how deeply functions can be nested.

The nesting does not mean defining one function within another.

Recursion:

When a called function in turn calls another function a process of chaining occurs. Recursion is a special case of this process, where a function calls itself.

Very simple example of recursion is,

```
Main ( )  
{  
    printf ("recursion");  
    main ( );  
}
```

Here, main () function is calling itself, this is called recursion. But the above example runs for infinite number of times.

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets and the problem. When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

Passing arrays to functions:

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one dimensional array to a

called function, it is sufficient to mention the name of the array, without any subscripts and the size of the array as arguments.

For example, the call

Largest (a, n)

Will pass the whole array `a` to the called function. The called function expecting this call must be appropriately defined. The largest function header must look like

Float largest (float a[], int size)

The function `largest` is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of number of elements in the array. The declaration of the formal argument array is made as follows.

Float a []

The pair of brackets informs the compiler that the argument `a` is an array of float values. It is not necessary to specify the size of the array here.

In C, the name of the array represents the address of its first element. By passing the array name, we are in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as pass by address (or pass by reference). We cannot pass a whole array as value as we do in case of ordinary variables.

When dealing with array arguments, there will be one major distinction. If a function changes the values of the elements of an array, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the

contents of the array are not copied into the formal parameter array, instead information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function however, this does not apply when an individual element is passed on as argument.

Two dimensional arrays as parameters to functions:

Like one dimensional arrays, we can also pass two dimensional arrays to functions. The approach is similar to one dimensional arrays.

The rules to pass two dimensional array as parameter to function are,

1. The function must be called by passing only array name without subscripts.
2. In the function definition, we must indicate that the array has two dimensions by including two brackets.
3. The size of second dimension must be specified.
4. The prototype declaration should be similar to function header.
