764 - Sri Ranganathar Institute of Polytechnic College
1/104 A, Athipalayam, Thudiyalur to Kovilpalayam Road,
Coimbatore – 641110
Tamil Nadu
Phone No: (0422)2904008,2904009
E-mail: sripoly@yahoo.co.in

NAME OF THE FACULTY	:	**SASTHI KUMAR C**
COURSE NAME	:	**DIPLOMA IN COMPUTER ENGINEERING**
SUBJECT CODE	:	**452330**
SEMESTER	:	**III**
SUBJECT TITLE	:	**C Programming and Data Structures**

VIDEO'S URL

UNIT - I PROGRAM DEVELOPMENT & INTRODUCTION TO C

ALL IN ONE -https://youtu.be/4I5Husjrj8A?t=574

PROGRAM DEVELOPMENT CYCLE - https://youtu.be/YV6ykfG1nQY

EXECUTION OF C - https://youtu.be/VDslRumKvRA?t=40

OPERATORS -

https://youtu.be/50Pb27JoUrw?list=PLBlnK6fEyqRhqQV_MzlT8xsPQnsGcMdIo&t=270

UNIT – I

PROGRAM DEVELOPMENT & INTRODUCTION TO C

1.1 PROGRAM , ALGORITHM & FLOW CHART

1.1.1 PROGRAM - DEFINITION

A computer program is a sequence of instructionswritten to perform a specified task with a computer.

Programs are written in a programming language. These programs are then translated into machinecodeby a compilerand linkerso that the computer can execute it directly or run it line by line (interpreted) by an interpreterprogram.

1.1.2 PROGRAM DEVELOPMENT LIFE CYCLE

The process of developing software, according to the desired needs of a user, by following a basic set of interrelated procedures is known as Program Development Life Cycle (PDLC)

PDLC includes various set of procedures and activities that are isolated and sequenced for learning purposes but in real life they overlap and are highly interrelated.

Tasks of Program Development

The basic set of procedures that are followed by various organizations in their program development methods are as follows:

1. Programspecification.
2. ProgramDesign.
3. Programcoding.
4. Programtesting.
5. Program documentation.
6. ProgramMaintenance.

Fig No 1 le

1. PROGRAMSPECIFICATION

This stage is the formal definition of the task. It includes the specification of inputs and outputs, processing requirements, system constraints, and error handling methods.

Thisstepisverycriticalforthecompletionofasatisfactoryprogram.Itisimpossibletosolveaproblem by using a computer, without a clear understanding and identification of the problem. Inadequate identification of problem leads to poor performance of the system. The programmer should invest a significant portion of his time in problemidentification.

2. PROGRAMDESIGN

In this phase the design of the system is designed. The Design is developed by the analysts and designers. The system analyst design the logical design for the designers and then designer get the basic idea of designing the software design of Front end and back end both.

The system analyst and Designer work together in designing the software design and designer design the best software design under the guidance of system analyst.

3.PROGRAMCODING

Thissteptransformstheprogramlogicdesigndocumentsintoacomputerlanguageformat.Thisstage translates program design into computer instructions. These instructions are the actual program. It is the crucial job of programmer to develop a code by following the flowchart and that code is written in any computerlanguage.

4. PROGRAMTESTING

Program testing is the process of checking program, to verify that it satisfies its requirements and to detect errors. These errors can be of any type - Syntax errors, Run-time errors and Logical errors . Testingincludenecessarystepstodetectallpossibleerrorsintheprogram.Thiscanbedoneeitherat a module level known as unit testing or at program level known as integrationtesting.

Syntax errors also known as compilation errors are caused by violation of the grammar rules of the language. The compiler detects, isolate these errors and terminate the source program after listing the errors. Common syntax errors include

- missing or misplaced; or},
- missing return type for aprocedure,
- Missing or duplicate variabledeclaration.
- Type errors thatinclude
  - type mismatch onassignment,
  - type mismatch between actual and formalparameters.

Logical errors: These are the errors related with the logic of the program execution. These errors are not detected by the compiler and are primarily due to a poor understanding of the problem or a lack of clarity of hierarchy of operators. Such errors cause incorrect result.

Errors such as mismatch of data types or array out of bound error are known as execution errors or **runtime errors**. These errors are generally undetected by the compiler, so programs with run-time error will run but produce erroneous results.

Debuggingisamethodicalprocessoffindingandreducingthenumberofbugsinacomputerprogram making it behave asexpected.

5. PROGRAMDOCUMENTATION

This task is performed by the programmer to make the code user friendly i.e. if new person got the code then he/she can easily understand that which statement performed what task.

Proper documentation is useful in the testing and debugging stages. It is also essential in the maintenance and redesign stages. A properly documented program can be easily reused when needed; an undocumented program usually requires so much extra work that the programmer might as well start from scratch. The techniques commonly used in documentation are flowcharts, comments, memory maps, parameter and definition lists, and program library forms.

Proper documentation combines all or most of the methods mentioned. Documentation is a time-consuming task. The programmer performs this task simultaneously with the design, coding, debuggingandtestingstagesofsoftwaredevelopment.Gooddocumentationsimplifiesmaintenance and redesign, and makes subsequent taskssimpler.

6. PROGRAMMAINTENANCE

During this phase, the program is actively used by the users. If the user encounters any problem or wants any enhancement, then repeat all the phases from the starting, so that the encountered problem

is solved or enhancement is added.

Programming language is a set of grammatical rules for instructing a computerto perform specific tasks. The term *programming language* usually refers to high-level languages, such as BASIC, C, C++,COBOL, FORTRAN, Ada, and Pascal. There are two major types of programming languages. These are LowLevelLanguagesandHighLevelLanguages.LowLevellanguagesarefurtherdividedinto*Machine language* and *Assemblylanguage*.

LOW LEVEL LANGUAGES

Low level languages are machine oriented and require knowledge of computer hardware and its configuration.

MachineLanguage

Machine Language is the only language that is directly understood by the computer. It does not need any translator program. It is written as strings of 1's (one) and 0's (zero). For example, a program instruction may look like this:

1011000111101

Itisnotaneasylanguagetolearnbecauseofitsdifficulttounderstand.Itisefficientforthecomputer butveryinefficientforprogrammers.Itisconsideredtothefirstgenerationlanguage.Itisalsodifficult to debug the program written in thislanguage.

Advantage: Machinelanguageprogramsarerunveryfastbecausenotranslationprogramisrequired for the CPU.

Disadvantages

1. It is very difficult to program in machine language. The programmer should know details of hardware to writeprogram.
2. Theprogrammershouldrememberalotofcodestowriteaprogramwhichresultsinprogram errors.
3. It is difficult to debug theprogram.

(b) AssemblyLanguage

Thecomputercanhandlenumbersandletter.Therefore, somecombinationofletterscanbe usedto substitutefornumberofmachinecodes.ThesetofsymbolsandlettersformstheAssemblyLanguage and a translator program is required to translate the assembly Language to machine language. This translator program is called`*Assembler*'.

Advantages:

1. Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
2. It is easier to correct errors and modify programinstructions.
3. Assembly Language has the same efficiency of execution as the machine level language. Because this is one-to-one translator between assembly language program and its corresponding machine language program.

Disadvantages:

1. The assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration.

HIGH LEVEL LANGUAGES

The assembly language and machine level language require deep knowledge of computer hardware. But High-level languages are machine independent. Programs are written in English-like statements. As high – level languages are not directly executable, translators (compilers or interpreters) are used to convert them into machine language.

Advantages of High Level Languages

1. These are easier to learn. Less time is required to writeprograms.
2. They are easier to maintain. Programs written in high-level languages are easier todebug.
3. Programs written in high-level languages are machine –dependent. Therefore programs developed on one computer can be run on another with little or nomodifications.

1.1.4 FEATURES OF A GOOD PROGRAMMING LANGUAGES

Simplicity: A good programming language must be simple and easy to learn and use. For example, BASIC is liked by many programmers only because of its simplicity. Thus, a good programming language should provide a programmer with a clear, simple and unified set of concepts which can be easily grasped.

Naturalness: A good language should be natural for the application area it has been designed. That is,itshouldprovideappropriateoperators,datastructures,controlstructures,andanaturalsyntaxin order to facilitate the users to code their problem easily andefficiently.

Abstraction: Abstractionmeanstheabilitytodefineandthenusecomplicatedstructures.Thedegree of abstraction allowed by a programming language directly affects itswritability.

Efficiency: The program written in good programming language are efficiently translated into machine code, are efficiently executed, and acquires as little space in the memory as possible.

Compactness: In a good programming language, programmers should be able to express intended operation concisely.

Locality: A good programming language should be such that while writing a program, a programmer need not jump around visually as the text of the program is prepared. This allows the programmer to concentrate almost solely on the part of the program around the statements currently being worked with.

Extensibility: A good programming language should allow extension through simple, natural, and elegant mechanisms. Almost all languages provide subprogram definition mechanisms for this purpose.

1.1.5 ALGORITHM - DEFINITION

1.1.5.1. ALGORITHM - DEFINITION

Algorithm is a step-by-step method of solving a problem or making decisions.

1.1.6 PROPERTIES OF ALGORITHM

1. **Finiteness:** An algorithm must always terminate after a finite number of steps. It means after every step one reach closer to solution of the problem and after a finite number of steps algorithm reaches to an endpoint.
2. **Definiteness:** Each step of an algorithm must be precisely defined. It is done by well thought actions to be performed at each step of the algorithm. Also the actions are defined unambiguously for each activity in the algorithm.
3. **Input**: Any operation to be performed needs some beginning value/quantities associated with different activities in the operation. So the value/quantities are given to the algorithm before it begins.
4. **Output:** One always expects output/result (expected value/quantities) in terms of output from an algorithm. The result may be obtained at different stages of the algorithm. If some result is from the intermediate stage of the operation then it is known as intermediate result and result obtained at the end of algorithm is known as end result. The output expected value/quantities always have a specified relation to the inputs.
5. **Effectiveness:** Algorithms to be developed/written using basic operations.

Any algorithm should have all these five properties otherwise it will not fulfill the basic objective of solving a problem in finite time.

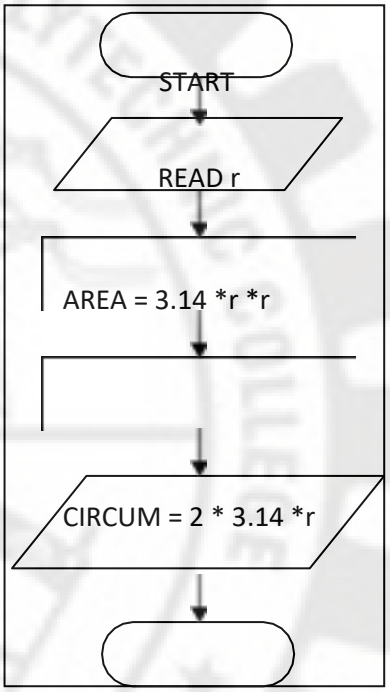
1.1.7 CLASSIFICATION OF ALGORITHMS

An algorithm may be implemented according to different basic principles.

- **Recursive or iterative**: A recursive algorithm is one that calls itself repeatedly until a certain condition matches. It is a method common to functional programming. Iterative algorithms use repetitive constructs like loops.
- **Logical or procedural:** An algorithm may be viewed as controlled logical deduction. A logic component expresses the axioms which may be used in the computation and a control component determines the way in which deduction is applied to the axioms.
- **Serial or parallel:** Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. This is a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures to process several instructions at once. They divide the problem into sub-problems and pass them to several processors.
- **Deterministic or non-deterministic:** Deterministic algorithms solve the problem with a predefined process whereas non-deterministic algorithm must perform guesses of best solution at each step through the use of heuristics.

1.1.8 ALGORITHMS LOGIC

Algorithmic logic is classified into three types. They are: (i) Sequential Logic (ii) Selection or Conditional Logic and (iii) Repetition logic. All the computing is done using only these types.



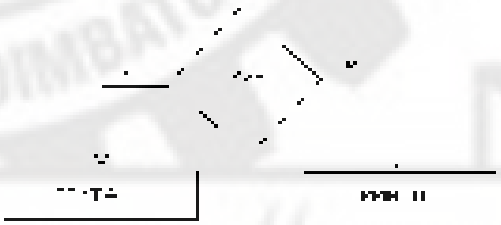
SEQUENTIAL LOGIC

As the name suggests, Sequential Logic consist of one action followed by another in a logical progression. In other words, perform operation A and then perform operation B and so on. This structure is represented by writing one operation after another.

Fig No 1.2 Example for Sequential Logic

SELECTION OR CONDITIONAL LOGIC

Selection Logic allows the program to make a choice between two alternate paths, whether it is true (1) or false (0). First statement is a conditional statement.



If the condition is true, Operation 1 is performed; otherwise Operation 2 will be performed. For example, if A>B, then print A, else print B

Fig No 1.3 Example for Selection or Conditional Logic

Thisiscalledthe"if…Then…Else"structure.Iftheansweris"Yes",thenthecontrolwillbetransferred to some other path. Otherwise, if the answer is "No", then the execution goes to the next statement without doinganything.

ITERATION OR REPETITION LOGIC

Repetition Control Structure is also termed as Iteration Control Structure or Program Loop. Repetition causes an interruption in the sequence of processing. In the Repetitive Control Structure, an operation or a set of operations is repeated as long as some condition is satisfied. The performed operation will be the same, but the data will change every time.

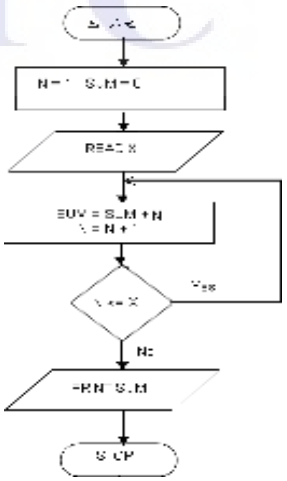


Fig No 1.4 Example for Repetition Logic

When a sequence of statements is repeated against a condition, it is said to be in a loop. Using looping, the programmer avoids writing the same

set of instructions again and again.

Some examples on developing algorithms using step-form:

- Each algorithm will be logically enclosed by two statements START andSTOP.
- Input or READ statements are used to accept data fromuser.
- PRINT statement is used to display any user message. The message will be displayed to be enclosed withinquotes.
- The arithmetic operators =, +, *, -, / will be used in theexpression.
- The commonly used relational operatos will include : >, <. >=, <=. +,!=
- The most commonly used logical operators will be AND, OR,NOT

Example 1 : To make a coffee

Step1: Take proper quantity of water in a cooking pan

Step2: Place the pan on a gas stow and light it

Step3: Add Coffee powder when it boils

Step4: Put out the light and add sufficient quantity of sugar and milk

Step5: Pour into cup and have it.

Example 2 : To find the area and circumference of a circle

1. Start
2. Read the radius of the circle r
3. Find the area and circumference of the circle using formula
   Area =3.14*r*r
   Circum =2*3.14*r
4. Print the area and circumference of thecircle
5. Stop

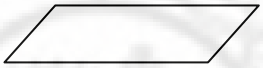
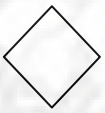
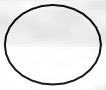
1.1.9 FLOW CHART

Aflowchartisadiagrammaticrepresentationthatshowsthesequenceofoperationstobeperformed to get the solution of aproblem.

IMPORTANCE OF FLOW CHART

Flowcharts facilitate communication between programmers and business people. Flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicatedandlengthyproblems.Oncetheflowchartisdrawn,itbecomeseasytowritetheprogram in any high level language. Flowcharts are helpful in explaining the program to others. Hence, a flowchart is a must for the better documentation of a complexprogram.

FLOWCHART SYMBOLS

S.No	Name of the Symbol	Symbol	Meaning
1	Start /Stop (Oval)		Represents the start and end of the program
2	Processing (Rectangle)		Represents arithmetic and data movementinstructions.
3	Input / Output (Parallelogram)		It is used to represent the input or output function.
4.	Decisions (Diamond)		Represents decision making and branching. It has one entry and two or more exit paths
5.	Connector (Circle)		Used to join two parts of a program
6	Flow lines (Arrow lines)		Used to link various boxes together to form the flow chart and indicate the direction of the flow.

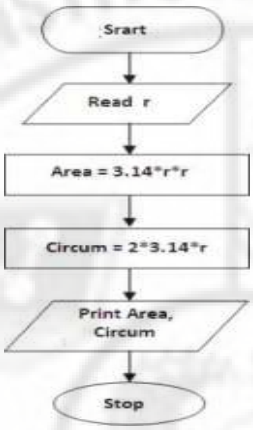
ADVANTAGES OF USING FLOWCHARTS

1. ***Communication:*** Flowcharts are better way of communicating the logic of a system to all concerned.
2. ***Effective analysis****:* With the help of flowchart, problem can be analysed in more effectiveway.
3. **Proper documentation:** Program flowcharts serve as a good program documentation, which is needed for variouspurposes.
4. ***Efficient Coding:*** The flowcharts act as a guide or blueprint during the systems analysis and program developmentphase.
5. ***Proper Debugging:*** The flowchart helps in debuggingprocess.
6. ***Efficient Program Maintenance:*** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on thatpart

LIMITATIONS OF USING FLOWCHARTS

1. *Complex logic:* Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex andclumsy.
2. *Alterations and Modifications:* If alterations are required the flowchart may require re-drawing completely.

FLOW CHART FOR FINDING THE AREA AND CIRCUMFERENCE OF CIRCLE



Algorithm and flow chart for finding the product of first n natural numbers

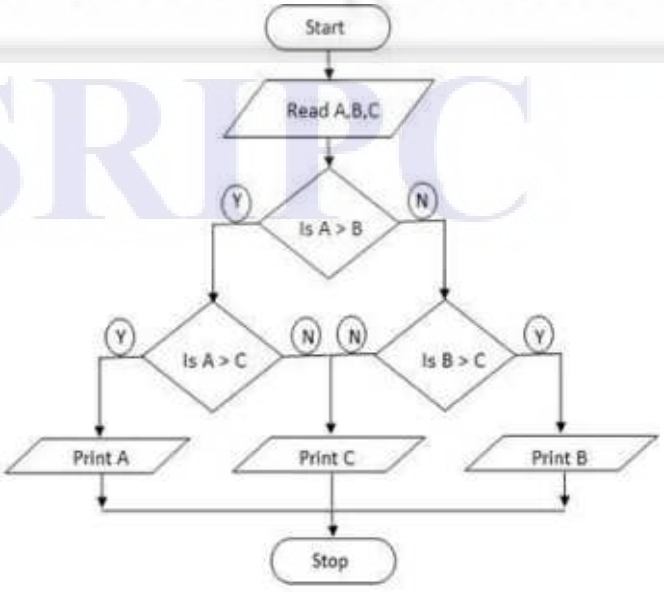
ALGORITHM

1. Start
2. Read the numbern.
3. [illegible]
4. While
   (count<=n)
   do
   [illegible]
   End the while loop.
5. Write “product of num is”,P.
6. Stop

FLOW CHART

Algorithm and flow chart for finding the largest of 3 numbers

1. Start.
2. Read three numbers s A, B andC.
3. Find A is greater than B and C, if so print the variable A.
4. If B is greater than A, find B is greater than C, if so print the variableB.
   Else print the variable C.
5. Stop



Algorithm and flow chart for finding whether the given number is odd or even.

ALGORITHM

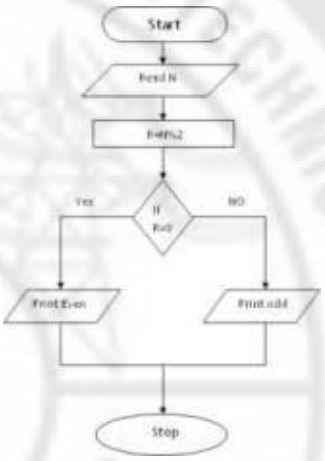
1. Start
2. Read the numberN.
3. Find the remainder(R) of N divided by2 using the modulus operator(N%2)
4. If the remainder iszero.

   Print "The number is Even Number".

   Else

   Print "The number is odd number".
5. Stop.

FLOW CHART



1.2.1 HISTORY OF C

C is a general purpose computer programming language and was developed at AT&Ts bell laboratory of USA in 1972. It was originally created by Dennis Ritchie. By 1960, different computer languages were used for different purposes. So, an International Committee was set up to develop a language that is suitable for all purposes. This language is called ALGOL 60.

To overcome the limitation of ALGOL 60, a new language called *Combined Programming Language (CPL)* was developed at Cambridge University. CPL has so many features. But it was difficult to learn and implement. Martin Richards of Cambridge University developed a language called *"Basic Combined Programming Language" (BCPL).* BCPL solves the problem of CPL. But it is less powerful. At the same time, Ken Thomson at AT&T's Bell laboratory developed a language called 'B'. Like BCPL, B is also very specific. Ritchie eliminated the limitations of B and BCPL and developed'C'.

1960	ALGOL	International Group	1972	Traditional C	Dennis Ritchie
1964	CPL	Cambridge University	1978	K&R C	Kernighan & Ritchie
1967	BCPL	Martin Richards	1989	ANSI C	ANSI Committee
1970	B	Ken Thompson	1990	ANSI/ISO C	ISO Committee

1.2.2 FEATURES OF C LANGUAGE

- Chasalltheadvantagesofassemblylanguageandallthesignificantfeaturesofmodernhigh-level language. So it is called a "*Middle Level Language*".
- C language is a very powerful and flexiblelanguage.

- C language supports a number of data types and consists of rich set ofoperators.
- C language provides dynamic storageallocation.
- C Compiler produces very fast objectcode.
- C language is a portable language. A code written in C on a particular machine can be compiled and run on anothermachine.

1.2.3 STRUCTURE OF A C PROGRAM

Any C Program consists of one or more function. A function is a collection of statements, used togethertoperformaparticulartask.Anoverviewofthe structureofaCprogram isgivenbelow:AC program may contain one or more sections. They are illustratedbelow.

Documentation Section: The documentation section consists of a set of comment lines giving the name of the program, the author and other details. It consist of a set of comment lines . These lines are not executable. Comments are very helpful in identifying the program features

Fig No 1.5 Structure of CProgram

Preprocessor Section: It is used to link system library files, for defining the macros and for defining the conditional inclusion.

Definition section : The definition section defines all symbolic constants.

Global Declaration Section: There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

main()functionsection: EveryCprogrammusthaveone main function section. This section contains two parts; declaration part and executablepart.

a) Declaration part : The declaration part declares all the variables used in the executablepart.

b) Executable part : There is at least one statement in the executablepart.

These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace.

Subprogram section : The subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

SAMPLE C PROGRAM

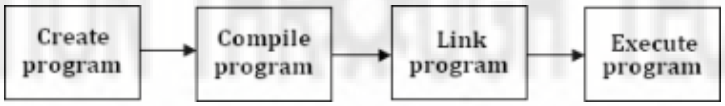
```
#include <stdio.h>

int main () { int number, remainder;

printf ("Enter your number to be tested: ");
scanf ("%d", &number);

remainder = number % 7;
if ( remainder == 0 )

printf ("The number is divided by 7.\n");
else

printf ("The number is not divided by 7.\n"); }
```

General rule for C Programming:

1. Every executable statement must end with semicolon symbol (;).
2. EveryCProgram,mustcontainexactlyonemainmethod(startingpointoftheprogramexecution)
3. All the system defined words (keywords) must be used in lowercaseletters.
4. Keywords cannot be used as user definednames.
5. For every open brace ({), there must be respective closingbrace(}).

1.2.4 EXECUTING A "C" PROGRAM

C program executes in following four steps.: 1. Creating the program 2. Compiling the Program 3. Linking the Program with system library 4. Executing theprogram



1. **Creatingaprogram:**Typetheprogramandedititinstandard'C'editorandsavetheprogram with**.c**asanextension.Thisisthesourceprogram.Thefileshouldbesavedas'*.c'extension only.

2. **Compiling (Alt + F9) theProgram:**
   - This is the process of converting the high level language program to Machine level Language (Equivalent machine instruction) .
   - Errors will be reported if there is any, after thecompilation
   - Otherwise the program will be converted into an object file (.obj file) as a result of the compilation
   - After error correction the program has to be compiledagain
3. **Linking a program to library:** The object code of a program is linked with libraries that are needed for execution of a program. The linker is used to link the program with libraries. It creates a file with '*.exe'extension.

4. **Execution of program:** This is the process of running (Ctrl + F9) and testing the program with sample data. If there are any run time errors, then they will bereported.

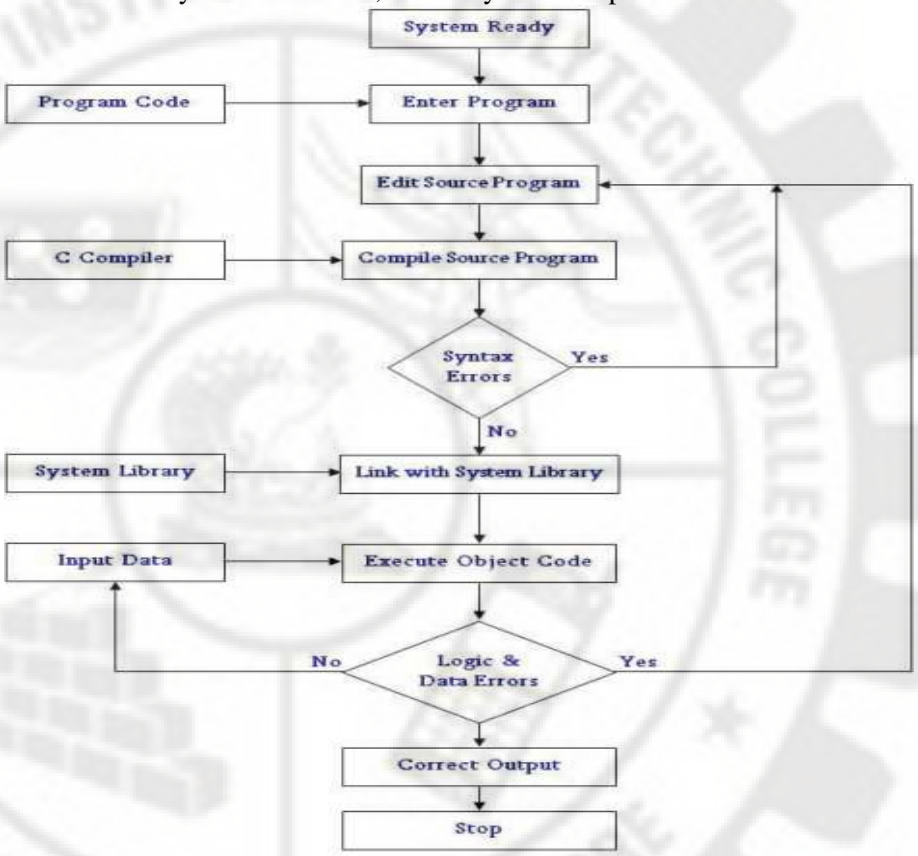


Figure : Process of compiling and running a C program.

1.3 VARIABLES , CONSTANTS AND DATA TYPES

1.3.1. C CHARCTER SET

EveryCprogramcontainsstatements.Thesestatementsareconstructedusingwordsandthesewords areconstructedusingcharactersfromCcharacterset.Clanguagecharactersetcontainsthefollowing set ofcharacters...

1. Alphabets 2. Digits 3.Special Symbols

Alphabets: Clanguagesupportsallthealphabetsfromenglishlanguage.Loweranduppercaseletters together supports 52alphabets.

lower case letters - **atoz** UPPER CASE LETTERS - **A toZ**

Digits :C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols :C language supports rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, back spaces and other special symbols.

,	Comma	&	Ampersand

.	Period	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus Sign
?	Question Mark	+	Plus Sign
'	Aphostrophe	<	Opening Angle (Less than sign)
"	Quotation Marks	>	Closing Angle (Greater than sign)
!	Exclamation Mark	(	Left Parenthesis
|	Vertical Bar	)	Right Parenthesis
/	Slash	[	Left Bracket
\	Backslash	]	Right Bracket
~	Tilde	{	Left Brace
-	Underscore	}	Right Bracket
$	Dollar Sign	#	Number Sign
%	Percentage Sign		

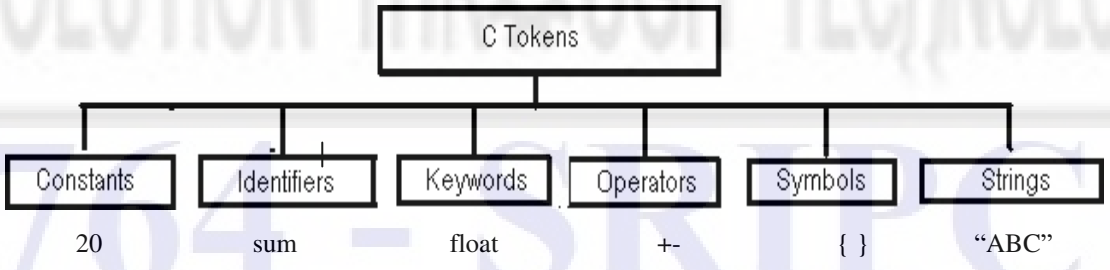
White Space Characters:

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words in strings. scanf() uses whitespace to separate consecutive input items from each other.

1.Blank Space 2.HorizontalTab 3.CarriageReturn 4.NewLine 5. FormFeed

1.3.2. C TOKENS

C tokens are the basic buildings blocks in C. Smallest individual units in a C program are the C tokens. C tokens are of six types. The figure 1.7 shows the C tokens.



KEYWORDS :

The meaning of these words has already been explained to the C compiler. All the keywords have fixed meanings. These meanings cannot be changed. So these words cannot be used for other purposes. *All keywords are in lower case.* The keywords are known *as Reserved words*. Keywords or Reserved words are *Pre-defined identifiers*. 32 keywords are available in C.

Properties of Keywords

1. All the keywords in C programming language are defined in lowercase lettersonly.
2. Every keyword has a specific meaning; users can not change thatmeaning.
3. Keywords cannot be used as user defined names like variable, functions, arrays, pointersetc...
4. Every keyword in C programming language represents some kind of action to be performed by the compiler.

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

1.3.4. IDENTIFIERS

Thenamesofvariables,functions,labelsandvariousotheruser-definedobjectsarecalled *Identifiers.* Identifiers are used for defining variable names, function names etc. The general rules to be followed when constructing an identifierare:

1) Identifiers are a sequence of characters. The only characters allowed are alphabetic characters, digits and the underscore character. Special characters are not allowed in identifiername.
2) The first character of an identifier is a letter or an underscorecharacter.
3) Identifiers are case sensitive. For example, the identifiers TOTAL and Total aredifferent.
4) Keywords are not allowed as identifiername.

VALID IDENTIFIERS:

Name area interest circum amount rate_of_interest sum

INVALID IDENTIFIERS

Identifers	Reason
4^{th}	The first letter is a numeric digit.
int	The Keyword should not be a identifier
First name	Blank space is not allowed.

1.3.5. CONSTANTS

The data values are usually called as Constant. Constant is a quantity that does not change during program execution. This quantity can be stored at a location in the memory of the computer. *C has four types of constants*: *Integer, Floating, String and Character.*

INTEGER CONSTANT

*AnIntegerConstantisanintegernumber.Aninteger constantisasequenceofdigits.**Therearе3types of integers namely decimal integer, octal integer and hexadecimalinteger.***

Decimal Integer consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non digit characters are not permitted between digits.

Example for valid decimal integer constantsare : 123 -31 0 562321 +78

Octal Integer constant consists of any combination of digits from 0 through 7 with a O at the beginning.

Examples of octal integers are:O26 O O347 O676

Hexadecimal integer constant is preceded by OX or Ox. They may contain alphabets from A to F or a to f. or a decimal digit (0 to 9). The alphabets A to F refers to 10 to 15 in decimal digits.

Example of valid hexadecimal integers are:*OX2 OX8C OXbcd Ox123*

REAL CONSTANT

Real constants are numbers with fractional part. Real constants are often called as *Floating Point constants.*

RULES

1. A real constant must have atleast onedigit.
2. It must have a decimal point.
3. It could be either positive ornegative.
4. Default sign is positive (If nosign)
5. Special characters are notallowed.
6. Omittingofdigitbeforethedecimalpoint,ordigitsafterthedecimalpointisallowed.(Ex .655,12.)

Examples:+325.34 426.0 -32.76 -48.5792

Real Constants are represented in two forms:

i) Fractionalform
ii) Exponentialform

EXPONENTIAL FORM or SCIENTIFIC FORM

The Exponential form is used to represent very large and very small numbers. *The exponential form representation hastwoparts: 1) mantissa and 2) exponent*. The part appears before the letter 'e'iscalled*mantissa*andthepartfollowingtheletter'e'iscalled*exponent*,whichrepresentsapower of ten.

mantissa e exponent

The general form of exponential representationis.

where "mantissa" is a decimal or integer quantity and the "exponent" is an integer quantity with an optional plus or minus sign.

The general rules regarding exponential forms are

1. The two parts should be separated by a letter “e” or“E”.
2. The mantissa and exponent part may have a positive or a negativesign.
3. Default sign of mantissa and exponent part ispositive.
4. The exponent must have at least onedigit.

Examples

Thevalue123.4 may be written as 1.234E+2 or

12.34E+1. The value 0.01234 may be written as 1.234E-2

or12.34E-3.

Differences between floating point numbers and integer numbers.

Integerincludesonlywholenumbers,butfloatingpointnumberscanbeeitherwholeorfractional.
Integers are always exact, whereas floating point numbers sometimes can lead to loss of mathematicalprecision.
Floating point operations are slower in execution and often occupy more memory than integer operations.

CHARACTER CONSTANTS

A character constant is either a single alphabet, a single digit or a single special character enclosed within a pair of single inverted commas.

The maximum length of a character constant can be 1 character

Examples: ‘A’ ‘1’ ‘$’ ‘‘ ‘;’

Each character constant has an integer value. This value is known as ASCII value. For example, the statementprintf (“”%d””,’A’) would print the value of 65. Because the ASCII value of A is 65.

Similarly the statement printf (“%c”,65) would display the letter ‘A’.

STRING CONSTANTS

A combination of characters enclosed within a pair of double inverted commas is called as “String Constant”.

Examples: “PRESTO” “75.567” “$825”“RANGANATHAR”

Each string constant ends with a special character ‘\0’. This character is not visible when the string is displayed. Thus **“PRESTO”** contains actually 6 characters. The ‘\0’ character acts as a string terminator. This character is used to find the end of a string.

Rememberthatacharacterconstant(e.g.,'A')andthecorrespondingsingle-characterstringconstant ("A") are not equivalent.

A character constant has an equivalent integer value, whereas a single-character string constant does not have an equivalent integer value and, in fact, consists of two characters - the specified character followed by the null character (*\ 0).*

1.3.6. VARIABLES – DEFINITION AND RULES

A quantity, which may vary during program execution, is called variable. Variables may be used to store a data value. Variables are actually memory locations , used to store constants. The variables are identified by names. The program may modify the values stored in a variable.

Rules for constructing variable names

1. A variable name is the combination of characters. The length of a variable depends upon the complier.
2. The first character must be an alphabet orunderscore.
3. Specialcharacterslikecommaorblankarenotallowedexceptanunderscorecharacter.Theonly characters allowed are letters, digits andunderscore.
4. The variable name should not be akeyword.

Examples :area interest circumference fact date_of_birth

1.3.7. DECLARING VARIABLES

InC,allthevariablesusedinaprogramaretobedeclaredbeforetheycanbeused.Allvariablesmust be declared in the beginning of the function. Type declaration statement is used to declare the type of various variables used in the program. The syntax for declarationis,

```
data_typevariable_name(s)
```

where data-type is a valid data type plus any other modifiers. variable-name(s) may contain one or more variable names separated by commas.

Examples :int a,b,c; long int interest; unsigned charch;

Declaration statement is used to allocate memory space for the variable. Declaration statement also provides a name for the location. Declaration statement declares that the program will use that variable name to identify the value stored at the location.

For example, the declaration

char sub-name

allocatesamemorylocationofsizeonebyte,ofcharactertype.Thismemorylocationisgivenaname ofsub-name.

1.3.8. INITIALIZATION OF A VARIABLE (ASSIGNING VAUSES TO VARIABLES

The process of assigning initial values to variables is called initialization of variables. In C, an uninitialised variable can contain any garbage value. Therefore, the programmer must make sure all

the variables are initialised before using them in any of the expressions. The value for a particular variableisinitializedthroughdeclarativestatementorassignmentstatement.Forexampletoinitialize the value 10 to an integer variable i, the following two methods areused.

i) inti =10;

ii) inti;
i = 10;

The above two methods declares that i is an integer variable with an initial value of 10.

Examples :(i) float pi= 3.14; (ii) char alpha= ''h''; (iii) int account =10;

More than one variable can be initialized in a single statement. For example, the statement

int x = 1, y = 2, z;

declares x to be an int with value 1, y to be an int with value 2 and z to be an int of unpredictable value. Same value can be initialized to several variables with the single assignment statement.

Examples

```
inti, j, k, l, m, n;
float a, b, c, d, e, f;
i = j = k = l = m = n = 20;
a = b = c= d = e = f = 13.56;
```

1.3.9. DECLARING VARIABLES AS CONSTANTS

A constant is a quantity whose value does not change during program execution. The qualifier **const**is used to declare the variable as constant at the time of initialization. The general form to declare variables as constant is

```
const data type variable = value;
```

whereconst - keyword datatype- valid type such as int, floatetc.

Example: const float PI =3.14;

constis a new data type qualifier. This tells the computer that the value of the **float** variable **PI** must not be modified by the program. But , it can be used on the right-hand side of an assignment like any other variable.

1.3.10. DECLARING VARIABLES AS VOLATILE

The qualifier volatile is used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources not by the program(from outside the program). General form to declare a variable as volatile is

```
volatile data type variable;
```

where volatile-keyword data type - valid types such as int, floatetc.

Example: volatile intx;

Thevalueofxmaybealteredbysomeexternalfactorsevenifitdoesnotappearontheleft-handside ofanassignmentstatement.Whendeclaringavariableasvolatile,thecompilerwillexaminethevalue of the variable each time it is encountered to see whether any external alteration has changed the value.

1.3.11. DATA TYPES

C supports several data types. Each data type may be represented differently inside the computer's memory. There are four data types in C language. They are,

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

Basic Data Types

Basic or Fundamental data types include the data types at the lowest level. i.e. those which are used for actual data representation in the memory. All other data types are based on the fundamental data types.

Examples: char, int, float, double.

inttypeisusedtostorepositiveornegativeintegers.floatdatatypeisusedtostoreasingleprecision floating-point (real) number. Floating-point numbers are stored in 32 bits with 6 digits of precision. double data type is used to hold real numbers with higher storage range and accuracy than the type float. The data type char is used to store onecharacter.

The size (number of bytes) and range of numbers to be stored in each data type is shown below.

Data Type	Size (Bytes)	Range
char	1	-128 to 127
int	2	-32,768 to 32, 767
float	4	3.4 E-38 to 3.4 E+38
double	8	1.7 E-308 to 3.4 E+308

Derived Data Types

These are based on fundamental data types. i.e. a derived data type is represented in the memory as a fundamental data type.

Examples: pointers, structures, arrays.

Void data type :

1. void is an empty data type that has novalue.
2. This can be used in functions andpointers.

1.3.12. DATA TYPES MODIFIERS(QUALIFIERS)

The basic data type may be modified by adding special keywords. These special keywords are called *data type modifiers (or) qualifiers.* Data type modifiers are used to produce new data types.

Themodifiers are: *signed unsigned long short*

The above modifiers can be applied to integer and char types. Long can also be applied to double type.

Integer Type

Asignedintegerconstantisintherangeof-32768to+32767.Integerconstantisalwaysstoredintwo bytes. Intwobytes,itisimpossibletostoreanumberbiggerthan+32767andsmallerthan -32768. Out of the two bytes used to store an integer, the leftmost bit is used to store the sign of the integer. So the remaining 15 bits are used to store a number. If the leftmost bit is 1, then the number is negative. If the leftmost bit is 0, then the number ispositive.

C has three classes of integer storage namely short int, int and long int. All of these data types have signed and unsigned forms. A short int requires half the space than normal size. Unsigned numbers are always positive and consume all the bits for storing the magnitude of the number. *The long and unsigned integers are used to declare a longer range of values.*

Floating Point Type

Floating point number represents a real number with 6 digits precision. When the accuracy of the floatingpointnumberisinsufficient,usethedoubletodefinethenumber.Thedoubleissameasfloat but with longer precision. To extend the precision further , use long double, which consumes 80 bits of memoryspace.

Character Type

Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitlyappliedtochar.Whileunsignedcharactershavevaluesbetween0and255,signedcharacters have values from –-128 to127.

Size and Range of Data Types on 16 bit machine.

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127
Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

1.3.13. OVERFLOW AND UNDERFLOW OF DATA

Assigningavaluewhichmorethantheupperlimitofthedatatypeiscalledoverflowandless than its lower limit is calledunderflow.

In case of integer types, overflow results wrapping towards negative side and underflow results wrapping towards positiveside.

In case of floating point types, overflow results +INF and underflow results –INF.

Example 1 :

```
#include <stdio.h>
void main()
{
int a = 32770;
printf( “%d””,a);
}
```

output :- 32766

The range of integer is – -32768 to + 32767, assigning 32770 results overflow and wrap towards –negative side.

Example 2:

```
#include <stdio.h>
void main()
{
int a = 33000;
float b = 3.4e50;
printf( “%d%f””,a,b);
}
```

Output :- 32536+INF

The range of integer is – -32768 to + 32767, assigning 33000 results overflow and wrap towards –negative side and +INF is a result of float overflow.

1.3.14. COMMENTS

Comments are used to make a program more readable. Comments are not instructions. Comments are remarks written in a program. These remarks are used to give more information about the program. The compiler will ignore the comment statements.

In C, there are two types of comments.

1. **Single Line Comments:** Single line comment begins with // symbol. Any number of single line comments can bewritten.
2. **Multiple Lines Comments:** Multiple lines comment begins with /* symbol and ends with */. Any number of multiple lines comments can be included in aprogram.

In a C program, the comment lines are optional. All the comment lines in a C program just provide the guidelines to understand the program and its code.

Examples:

(e.g.) /* Program to find the Factorial */

Any number of comments can be placed at any place in a program. Comments cannot be nested. For example

/* Author Arul /* date 01/09/93*/*/ is notpossible.

A comment can be split over more than one line. For example, the following is valid.

/* This statement is used to find the sum of two
numbers */

1.3.15. ESCAPE SEQUENCES

Charactercombinationsconsistingofa backslash(\)followedbyaletterorby acombinationofdigits are called "escape sequences." To represent a newline character, single quotation mark, or certain othercharactersinacharacterconstant,escapesequencesareused.Anescapesequenceisregarded as a single character and is therefore valid as a characterconstant.

- The escape sequence characters are also called as backslash characterconstants.
- These are used for formatting theoutput

For example, a line feed (LF), which is referred to as a *newline* in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters. The commonly used escape sequences are listed below.

ESCAPE CHARACTER	MEANING	ESCAPE CHARACTER	MEANING
bell (alert)	\a	carriage return	\r
backspace	\b	quotation mark (")	*
horizontal tab	\t	apostrophe (')	\’
vertical tab	\v	question mark (?)	\?
newline (line feed)	\n	backslash	\\
form feed	\f	null	\0

The following program outputs a new line and a tab and then prints the string *This is a test.*

```
#include
<stdio.h>intmain()
{
        printf("\n\tThis is a test.");
        return 0;
```

}

Characteristics

- Although it consists of two characters, it represents singlecharacter.
- Every combination starts with backslash(\)
- They are non-printingcharacters.
- It can also be expressed in terms of octal digits or hexadecimalsequence.
- Each escape sequence has unique ASCIIvalue.

1.3.16. SYMBOLIC CONSTANTS

- Names given to values that cannot be changed. Implemented with the #define preprocessor directive.

```
#define N 3000
#define FALSE 0
```

Preprocessor statements begin with a # symbol, and are NOT terminated by a semicolon.
Traditionally, preprocessor statements are listed at the beginning of the sourcefile.

preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like N) which occur in the C program are replaced by their value (like 3000). Once this substitution has taken place by the preprocessor, the program is thencompiled.

In general, preprocessor constants are written in UPPERCASE. This acts as a form of internal documentation to enhance program readability andreuse.

In the program itself, values cannot be assigned to symbolicconstants.

Use of Symbolic Constants

• Consider the following program which defines a constant calledTAXRATE.

```
#include <stdio.h>
#define TAXRATE
0.10 main () {

float balance;
float tax;
balance = 72.10;

tax = balance * TAXRATE;

printf("The tax on %.2f is %.2f\n",balance, tax);}
```

1.4. COPERATORS

1.4.1. INTRODUCTIONS TO OPERATORS

C has a very rich set of operators. So C language is some times called “the language of operators”. Operatorsareusedtomanipulatedataandvariables.Anoperatorisasymbol,whichrepresentssome particular operation that can be performed on some data. The data itself (which can be either a variable or a

constant) is called the'operand'.

Operators operate on constants or variables, which are called *operands*. Operators can be generally classified as either unary, binary or ternary operators. *Unary operators* act on one operand, *binary operators* act on two operands and *ternary operators* operate on three operands.

Depending on the function performed, the operators can be classified as

Arithmetic	Relational	Logical	Conditional	Assignment
Increment & Decrement	Modulo division	Bitwise	Special operators	

1.4.2. ARITHMETIC OPERATORS

Arithmetic operators are used to perform arithmetic operation in C. Arithmetic operators are divided into two classes: (i) Unary arithmetic operators and (ii) Binary arithmetic operators

Binary operators

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

ArithmeticOperators+,-,*and/canbeappliedtoalmostanybuilt-indatatypes.Supposethataand b are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resultingvalues.

Operation	Value
a+ b	13
a–b	7
a* b	30
a/ b	3
a%b	1

Modulo operator: The modulo operator (%) gives the remainder of the division between the two integervalues.ForModulodivision,thesignofresultisalwaysthatofthefirstoperandordividend.

Forexample, 13%-5 =3; -13 % - 5= -3; -13% 5 =-3;
13% 5 = 3;

Example :

```
main()
{
int a, b,c,d;
a =10;
b = 4;
```

```
c= a/b;
d = a % b;
printf(“”%d %d””, c, d);
}
```

Modulo division operator cannot be used with floating point type. C does not have no option for exponentiation.

Unary minus operator

In unary minus operation, minus sign precedes a numerical constant, variable or an expression. A negative number is actually an expression, consisting of the unary minus operator, followed by a positive numeric constant. *Unary minus operation is entirely different from the subtraction operator (-). The subtraction operator requires twooperands.*

1.4.3. INCREMENT AND DECREMENT OPERATORS

C contains two special operators ++ and --. ++ is called Increment operator. -- is called Decrement operator; The above two operators are called unary operators since they operate on only one operand. The operand has to be a variable and not a constant. Thus, the expression 'a++' is valid whereas '6++' is invalid.

Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

Syntax:

Incrementoperator: ++var_name; (or) var_name++;

Decrement operator: – --var_name; (or) var_name–--;

Example:

Increment operator : ++ i ; i ++ ;

Decrement operator : --i ; i-- ;

If the operator is used before the operand, then it is called *prefix operator.* (Example: ++a, --a). If the operator is used after the operand, then it is called *postfix operator.* (Example: a++, a--).

Difference between pre/post increment & decrement operators in C:

Below table will explain the difference between pre/post increment and decrement operators .

Operator	Operator/Description
Pre increment operator (++i)	value of i is incremented before assigning it to the variable i
Post increment operator (i++)	value of i is incremented after assigning it to the variable i
Pre decrement operator (–i)	value of i is decremented before assigning it to the variable i
Post decrement operator (i–)	value of i is decremented after assigning it to variable i

Prefix and Postfix operators have the same effect if they are used in an isolated C statement. For example, the two statements

x++; and ++x; have the same effect

But prefix and postfix operators have different effects when they are assigned to some other variable. For example the statements

z = ++x; and z = x++; have different effects.

Assume the value of x to be 10. The execution of the statement z= ++x; will first increment the value of x to 11 and assign new value to z. The above statement is equal to the following two statements.

x = x + 1
; z = x ;

The execution of the statement z = x++; will first assign the value of z to 10 and then increase the value of x to 11. The above statement is equal to

z = x ;
x = x + 1 ;

The decrement operators are also in a similar way, except the values of x and z which are decreased by 1.

Other Examples

a = 10,b=6	a=10, b =6
c = a* b++	c = a *++b

Output:

c=60;	c =70

The expression n++ requires a single machine instruction such as INR to carry out the increment operation. But n+1 operation requires more instructions to carry out this operation. So the execution of n++ is faster than n+ 1 operation.

1.4.4. RELATIONAL OPERATORS

Relational operators are used to test the relationship between two operands. The operands can be variables, constants or expressions. C has six relational operators. They are

Operator	Meaning	Operator	Meaning
<	is less than	>=	is greater than or equal to
<=	is less than or equal to	==	is equal to
>	is greater than	!=	is not equal to

An expression containing a relational operator is called as a relational expression. The value of the relationalexpressioniseithertrueorfalse.Ifitisfalse,thevalueoftheexpressionis0andifitistrue, the value is1.

Examples :

Supposethati,jandkareintegervariableswhosevaluesare1,2and3,respectively.Severalrelational expressions involving these variables are shownbelow.

Expression	*Interpretation*	Value
i< j	true	1
(1 + j) >= k	true	1

(j + k) > (i + 5)	false	0
k != 3	false	0
j ==2	true	1

1.4.5. LOGICAL OPERATORS

Logical operators are used to combine or negate expression containing relational expressions. C provides three logical operators.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logicalexpressionisthecombinationoftwoormorerelationalexpressions.Logicaloperatorsareused to combine the result of evaluation of relational expressions. Like the simple relational expression, a logical expression also gives value of one orzero.

LOGICAL AND (&&)

Thisoperatorisusedtoevaluatetwoconditionsorexpressionssimultaneously.Ifboththeexpressions totheleftandtotherightofthelogicaloperatoristruethenthewholecompoundexpressionistrue.

Example : a > b && x = =10

The expression to the left is a > b and that on the right is x == 10.The whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

LOGICAL OR (||)

The logical OR is used to combine two expressions or the condition. If any one of the expression is true, then the whole compound expression is true.

Example : a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

LOGICAL NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words, it just reverses the value of the expression.

Examples

Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is *5.5,* and c is a character variable that represents the character 'w '. Several complex logical expressions that make use of these variables are shown below.

Expression	*Interpretation*	*Value*
(i>= 6) && (c == ' w')	true	1
(i>= 6) || (c == 119)	true	1
(f < 11) && (i> 100)	false	0
(c != ' p ')|| ((i + f) <= 10)	true	1

The truth table for the logical operators is shown here using 1's and 0's.

p	q	p && q	p || q	!p

0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

1.4.6. CONDITIONAL OPERATORS

Simple conditional operations can be carried out with the conditional operator (?:). *The conditional operator is used to build a sionditional expression.* The Conditional operator has two parts: *?and :*

An expression that uses the conditional operator is called conditional expression. The conditional operator is a ternary operator because it operates on three operands. The general form is

```
expression1 ?expression 2 : expression 3 ;
```

The expression1 is evaluated first. If it is true (non – zero), then the expression 2 is evaluated and its valueisreturned.Ifexpression1isfalse(zero),thenexpression3isevaluatedanditsvalueisreturned. Only one expression either expression 2 or expression 3 will be evaluated at atime.

Conditional expression frequently appear on the righthand side of a simple assignment statement. The resulting value of the conditional expression is assigned to the identifier on the left.

Example : max = c >d ? c : d;

The purpose of the above statement is to assign the value of c or d to max, whichever is larger. First the condition (c > d) is tested. If it is true, max = c; if it is false max = d;

1.4.7. ASSIGNMENT AND SHORT-HAND ASSIGNMENT OPERATORS

The Assignment Operator (=) evaluates an expression on the right hand side of the expression and substitutes this value to the variable on the left hand side of the expression.

Example : x = a + b;

Here the value of a + b is evaluated and substituted to the variable x.

Inaddition,Chasasetofshorthandassignmentoperators.Shorthandassignmentoperatorsareused to simplify the coding of a certain type of assignment statement. The general form of this statement is

```
varoper = exp;
```

Here *var*is a variable, *exp*is an expression or constant or variable and *oper*is a C binary arithmetic operator. The operator *oper* = is known as shorthand assignment operator

The above general form translatesto: var=*var opexp;*

The compound assignment statement is useful when the variable name is longer. For example

amount-of-interest = amount-of-interest * 10 ;

can be written as

amount-of-interest * = 10 ;

For example a = a+1; can bewrittenas a + =1

The short hand works for all binary operators in C. Examples are

x - = y; is equal to x = x - y;
x * = y; is equal to x = x * y;
x / = y; is equal to x = x / y;
x %= y; is equal to x = x % y;

1.4.8. BITWISE OPERATORS

The combination of 8 bits is called as one byte. A bit stores either a 0 or 1. Data are stored in the memory and in the registers as a sequence of bits with 0 or 1.

Bitwiseoperatorsareusedformanipulationofdataatbitlevel.*Theoperatorsthatareusedtoperform bit manipulations are called bit operators.* C supports the following six bitoperators.

Operator	Description	Operator	Description
&	Bitwise AND	~	One's Complement
|	Bitwise OR	<<	Shift left
^	Bitwise X-OR	>>	Shift right

These operators can operate only on integers and not on float or double data types. All operators except ~ operator are binary operators, requiring two operands. When using the bit operators, each operand is treated as a binary number consisting of a series of individual 1s and 0s. The respective bits in each operand are then compared on a bit by bit basis and result is determined based on the selected operation.

Bitwise AND operator

Bitwise AND operator(&) is used to mask off certain bits. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Assume the two variables a and b, whose values are 12 and 24. The result of the statement c = a & b is

a	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c=a&b	0	0	0	0	1	0	0	0

Thevalueofcis8.**Tomasktheparticularbit(s)inavalue,ANDtheabovevaluewithanothervalue by placing 0 in the corresponding bit in the secondvalue.**

Bitwise OR operator *Bitwise OR operator(|) is used to turn ON certain bits.* The result of ORing operation is 1 if any one of the e bits have a value of 1; otherwise it is 0. Assume the two variables a and b, whose values are 12 and 24. The result of the statement c = a | b is

a	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c=a|b	0	0	0	1	1	1	0	0

The value of c is 28. **To turn ON a particular bit(s) in a pattern of bits, OR the above value with another value by placing 1 in the corresponding bit in the second value.**

Bitwise Exclusive OR operator

The result of bitwise Exclusive OR operator (^) is 1 if only one of the bits is 1; otherwise it is 0. The result of a ^ b is

A	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c= a^b	0	0	0	1	0	1	0	0

The value of c is 20.

Bitwise Complement operator

The complement operator ~ complements all the bits in an operand. That is, 0 changes to 1 and 1 changes to 0. If the value of a is 00001100, then ~a is 11110011.

Shiftoperators

The shift operators are used to move bit pattern either to the left or right. The general forms of shift operatorsare

Leftshift : operand <<n Rightshift : operand >>n

where*operand* is an integer and *n* is the number of bit positions to be shifted. The value for n must be an integer quantity.

Assume the value of a is 12. Then the result of a << 2 will be follows:

a	0	0	0	0	1	1	0	0
a<<2	0	0	1	1	0	0	0	0

The value of c is 48. The above operation shifts the bits to the left by two places and the vacated places on the right side will be filled with zeros. **Every shift to the left by one position corresponds tomultiplicationby2.**Shiftingtwopositionsisequaltomultiplicationby2*2i.e,by4.Similarly,every shift to the right by one position corresponds to division by2.

1.4.9. SPECIAL OPERATORS

C supports some special operators like comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. And ->).

Comma Operator:

Comma operator is used in the assignment statement to assign many values to many variables.

Example: int a=10, b= 20, c- 30;

Comma operator is also used in for loop in all the three fields.

Example: for (i=0, j=2; i<10; i=i+1;j=j+2);

Sizeof Operator:

Another unary operator is the sizeofoperator . This operator returns the size of its operand, in bytes. This operator always precedes its operand. The operand may be a variable, a constant or a data type qualifier. Consider the following program.

```
main(){
int sum;
printf("%d",
sizeof(float));
printf("%d", sizeof
(sum));
printf("%d", sizeof (234L));
printf("%d", sizeof ('A')); }
```

Here the first printf( ) would print out 4 since a float is always 4 bytes long. With this reasoning, the next three printf( ) statements would output 2, 4 and 2. Consider an array school[]= "National" . Then, sizeof (school) statement will give the result as 8.

1.4.10. HIERARCHY OF OPERATIONS (PRECEDENCE AND ACCOCIATIVITY)

The order in which the operations are performed in an expression is called hierarchy of operations. The priority or precedence of operators is given below.

The arithmetic operators have the highest priority. Both relational and logical operators are lower in precedence than the arithmetic operators (except !). The shorthand assignment operators have the lowest priority.

Order	Category	Operator	Operation	Associativity
1	Highest precedence	( ) [ ] → : : .	Function call	L → R Left to Right
2	Unary	! ~ + - ++ - - & * Size of	Logical negation (NOT) Bitwise 1's omplement Unary plus Unary minus Pre or post increment Pre or post decrement Address Indirection Size of operant in bytes	R → L Right -> Left

3	Multiplication	* / %	Multiply Divide Modulus	L → R
4	Additive	+ -	Binary Plus Binary Minus	L → R

5	Shift	<< >>	Left shift Right shift	L → R
6	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	L → R
7	Equality	== !=	Equal to Not Equal to	L → R
8	Bitwise AND	&	Bitwise AND	L → R
9	Bitwise XOR	^	Bitwise XOR	L → R
10	Bitwise OR	|	Bitwise OR	L → R
	Logical AND	&&	Logical AND	L → R
12	Conditional	? :	Ternary Operator	R → L
13	Assignment	= *= %= /= += -= &= ^= |= <<= >>=	Assignment Assign product Assign reminder Assign quotient Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	R → L
14	Comma	,	Evaluate	L → R

Anexpressionwithintheparenthesesisalwaysevaluatedfirst.*Onesetofparenthesescanbeenclosed within another. This is called nesting of parentheses.* In such cases, innermost parentheses is evaluatedfirst.

Associatively means how an operator associates with its operands. For example, the unary minus associated with the quantity to its right, and in division the left operand is divided by right. The assignment operator '=' associates from right to left. Associativity also refers to the order in which C evaluatesoperatorsinanexpressionhavingsameprecedence.Forexample,thestatementa=b=20 / 2 ; assigns the value of 10 to b, which is then assigned to 'a', since associativity said to be fromright to left.

1.4.11. EVALUATION OF EXPRESSIONS

Example 1: Determine the hierarchy of operations and evaluate the following expression:
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

Stepwise evaluation of this expression is shown below:

i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8 operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8 operation: /
i = 1 + 1+ 8 - 2 + 5 / 8 operation: /
i = 1 + 1 + 8 - 2 + 0 operation: /

i = 2 + 8 - 2 + 0 operation: +
i = 10 - 2 + 0 operation:
+ i = 8 + 0 operation : -
i = 8 operation: +

Note that 6 / 4 gives 1 and not 1.5. This so happens because 6 and 4 both are integers and therefore would evaluate to only an integer constant. Similarly 5 / 8 evaluates to zero, since 5 and 8 are integer constants and hence must return an integer value.

Example 2: Determine the hierarchy of operations and evaluate the following expression:
k = 3 / 2 * 4 + 3 / 8 + 3

k = 3 / 2 * 4 + 3 / 8 + 3
k = 1 * 4 + 3 / 8 + 3 operation: /
k = 4 + 3 / 8 + 3 operation: *
k = 4 + 0 + 3 operation:
/ k = 4 + 3 operation: +
k = 7 operation +

1.4.12. ASSIGNMENT STATEMENTS

Assignment statements are used to assign the values to variables. Assignment statements are constructed using the assignment operator (=). General form of assigning value to variable is

```
variable = expression;
```

where expression is a constant or a variable name or the expression (Combinations of constants, variables and operators).

Examples : (i) a = 5; (ii) a = b; (iii) a = a+b; (iv) a = a >b;

Rules to be followed when constructing assignment statements

1. Only one variable is allowed on the left hand side of '=' expression a = b * c is valid, whereas a + b = 5 isinvalid.
2. Arithmeticoperatorscanbeperformedonchar,int,floatanddoubledatatypes.Forexample the following program is valid, since the addition is performed on the ASCII value of the characters x andy.

```
chara,b;
int z;
a =
'x'; b
= 'y'
z = a +b;
```

3. All the operators must be written explicitly. For example D = x. y .z is invalid. It must be written as D = x * y *z.

Multiple Assignment Statement

The same value can be assigned to more than one variable in a single assignment. This is possible by using multiple assignments. For example to assign the value 30 to the variable a, b, c, d is

```
a = b = c = d = 30;
```

However, this cannot be done at the time of declaration of variables.

For example

int a = b = c = d = 30 is invalid.

1.4.1. EXPRESSIONS

An expression is the combination of Variables and Constants connected by any one of the arithmetic operator. Examples for expressions are:

(i) a + b (ii)b+5 (iii) a+b*5-6/c (iv) 8.2+a/b+6.7

Integer expression: If all the variables and constants in an expression are of integer type, then this type of expression is called as integer expression. An integer expression will always give a result in integer value.

Real Expression: If all variables and constants in an expression are of real type, then this type of expression is called as real expression, A real expression will always give a result in real value.

MixedmodeExpression: Iftheelementsinanexpressionareofbothrealandintegertypes,thenthis type of expression is called as mixed mode expression. A mixed mode expression will always give a result in realvalue.

Examples:

Integer Expression	Real Expression	Mixed Mode Expresiion
inta,b,c; c = a+b/5 + c/a;	float a,b,c; c = a+b/2.0 + c+ 5.6	inta,b; float c,d; d = 5 + a/c + c/b + 6.1

1.4.14. TYPECONVERSION

Implicit type conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic type conversion is known as *implicit type conversion*

When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called *type promotion*. The order of various data type is

char<int< long < float < double

For example if an expression contains an operator between int and float, the int would be automatically promoted to a float before carrying out the operation. The following table shows the final data type when two operands are of different types.

Opearnd1	Opearnd2	Result
char	int	Int
char	float	float
int	float	float
int	char	Int
long int	float	fFloat
float	double	double
double	char	double

double	long int	double

explicit Conversion (Type cast operator)

Consider for example the ratio of number of female and male students in a class is

Ratio =female_students /male_students

Sinceiffemale_studentsandmale_studentsaredeclaredasintegers,thedecimalpartwillberounded offanditsratiowillrepresentawrongfigure.Thisproblemcanbe solvedbyconvertinglocallyoneof the variables to the floating point as shownbelow.

Ratio = (float) female_students / male_students

Theoperatorfloatconvertsthefemale_studentstofloatingpointforthepurposeofevaluationofthe expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of theresult.

The process of such a local conversion is known as explicit conversion or casting a value. The general

(type_name) expression

form is

I/OFUNCTIONS

C has no input and output statement to perform the input and output operations. C language has a collection of library functions for input and output (I/O) operations. The input and output functions are used to transfer the information between the computer and the standard input / output devices.

Console I/O functions - functions to receive input from keyboard and write output to VDU.

1.5.1. FORMATTED AND UNFORMATTED FUNCTIONS

The basic difference between formatted and unformatted I/O functions is that the formatted functionsallowtheinputreadfromthekeyboardortheoutputdisplayedontheVDUtobeformatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how manyspaceswouldbepresentbetweenthetwovalues,thenumberofplacesafterthedecimalpoints etc., can be controlled using formattedfunctions.

Console Input/Output functions

Formatted functions				Unformatted functions		
Type	*Input*	*Output*		*Type*	*Input*	*Output*
char	scanf()	printf()		char	getch()	putch()
					getchar()	putchar()
					getche( )	
int	scanf()	printf()		int	-	-
float	scanf()	printf()		float	-	-
string	scanf()	printf()		string	gets()	puts()

1.5.2. PRINTF () FUNCTION

printf ( ) function is used to display data on the monitor. The printf ( ) function moves data from the computer's memory to the standard output device. The general format of printf ( ) function is

```
printf ( "control string", list of arguments );
```

Comma is used to separate the control string from variable list.

The control string cancontain: -the characters that are to be displayed on thescreen.
-format specifiers that begin with a % sign.
-escape sequences that begin with a backward slash (\) sign.

Examples

printf (""%f, %f, %d, %d"", i, j, k + i, 5);
printf ("Sum of %d and %d is %d"", a, b, a + b
); printf (""Two numbers are %d and % d"",a,b
);

printf ( ) never supplies a newline automatically. Therefore multiple printf ( ) statements may be used to display one line of output. A new line can be introduced by the new line character \n.

Conversion Characters or Format Specifiers

The character specified after % is called a conversion character. Conversion character is used to convert one data type to another type.

Conversion Character	Meaning
%c	Data item is displayed as a single character
%d	Data item is displayed as a signed decimal integer
% e	Data item is displayed as a floating-point value with an exponent
% f	Data item is displayed as a floating-point value without an exponent
%i	Data item is displayed as a signed decimal integer
%o	Data item is displayed as an octal integer, without a leading zero
%s	Data item is displayed as a string
%u	Data item is displayed as an unsigned decimal integer
%x	Data item is displayed as a hexadecimal integer, without the leading Ox

List of Variable Arguments

The arguments may be constants, single variable or array name or more complex expressions.

1.5.3. FORMATTED PRINTF FUNCTIONS

Formatted printf function is used for the following purposes:

1. To print values at the particular position of thescreen.
2. To insert the spaces between the two values.
3. To give the number of places after the decimalpoint.

The above things can be achieved by adding modifier to the format specifiers. Modifiers are placed between the percent sign (%) and the format specifier. A maximum field width, the number of

decimal places, left justification can be specified by using the modifiers.

The number placed between percent sign (%) and the format specifier is called a field width specifier.

For outputting Integer Numbers

The general format of the field width specifier is

%Wd

where W is the minimum field width. If the value to be printed is greater thanthe specified field width,thewholevalueisdisplayed.Iftheoutputisshorterthanthelengthspecified,remainingspaces will be filled by blank spaces and the value is right justified. The following example illustrates the output of the number 1234 under different formats.

Format	Output					
printf(“”%d””,1234)	1	2	3	4		
printf(“”%6d””,1234)			1	2	3	4
printf(“”%2d””,1234)	1	2	3	4		
printf(“”%-6d””,1234)	1	2	3	4		
printf(“”%4d””,-1234)	-	1	2	3	4	
printf(“”%06d””,1234)	0	0	1	2	3	4

Commonly used Flags:

-	Data item is left justified within the field
+	A sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
0	Causes leading zeros to appear instead of leading blanks.

Outputting of Real Numbers

The general format for field width specification for outputting real number is

% W.d f

Where W represents the minimum number of width used for displaying the output value and “d” indicatesthenumberofdigitstobedisplayedafterthedecimalpoint.*Thisnumberiscalledprecision.* The precision is an unsigned number. The given value is rounded to d decimal places and printed right-justified in the field of W columns. The default precision is 6 decimal places. Real numbers are also displayed in exponentiation form by using the specificatione.

The following example illustrates the output of the number y = 12.3456 under different formats.

printf(“”%8.4f””,y)		1	2	.	3	4	5	6
printf(“”%8.2f””,y)				1	2	.	3	5
printf(“”%-8.2f””,y)	1	2	.	3	5			
printf(“%f””,y)	1	2	.	3	4	5	6	0	0
printf(“%8.2e”,y)	1	.	2	3	e	+	0	1	

Outputting of string data

The general format for field width specification for outputting string data is

```
% W. d s
```

where W represents the minimum number of width used for displaying the output value and d indicates the maximum number of characters that can be displayed. If the precision specification is less than the total number of characters in the string, the excess rightmost characters will not be displayed.

Whenusingstring,fortheminimumfieldwidthspecification,leadingblankswillbeaddedifthestring is shorter than the specified field width, Additional space will be allocated if the string is larger than the specified fieldwidth.

Example

```
main(){
char item[9] = “”computer””;
printf(%5s\n%11s\n%11.5s\n%.5s”,item,item,item,item);}
```

Output:

```
  c   o   m   P   U   t   e   r
              C   O   m   p   u   t   e   r
                          c   o   m   p   u
  c   o   m   p   U
```

1.5.4. SCANF() FUNCTION

scanf( ) function is used to read the value from the input device. The general format of scanf ( ) function is

```
scanf (“Control string”, list of address of arguments );
```

The list of address of arguments represents the data items to be read. Each variable name must be preceded by an ampersand (&). The arguments are actually pointer which indicate where the data items are stored in the memory. Example of a scanf function is

```
scanf ( “%d %f %c””, &a, &b, &ch);
```

The control string consists of format specifier, white space character and non-white space character. The format specifier in scanf are very similar to those used with printf ( ).

Important points while using scanf are

1. Every basic data type variable must be preceded by (&) sign**. In the case of string and array data type, the data name is not preceded by the character&.**
2. Text to be printed is not allowed in scanf statement. For example *scanf (“Enter thenumber %d”, &a);* is not valid.
3. The data items must correspond to thearguments.
4. If two or more data items are entered, they must separated by white space characters. (blank spaces, tabs or new line characters). Data items may continue into two or more lines. For example, for the statement

   ```
   scanf (“”%s %d %f””, name, &regno, &avg );
   ```

   the data items can be entered in the following methods.

(i) Arul 12345 85.56

(ii) Arul 12345 85.56

(iii) Arul 12345 85.56

(iv) Arul 12345 85.56

Modifier with scanf functions

A modifier is used to specify the maximum field length. The modifier is placed between % sign and the format specifier.

Example :scanf ("%6s"", str1);

is used to read a string of maximum 6 characters. If the input is more than 6 characters, first 6 characters will be assigned.

Length of data items may be lesser than the specified field width. But the number of characters in the actual data item cannot exceed the specified field width. Any excess character will not be read. Such uncovered characters

1.5.5. UNFORMATTED FUNCTIONS

may be incorrectly interpreted. This excess character acts as the input for the next data items.

GETCHAR () FUNCTION

getchar() function is used to read one character at a time from the standard input device. When the getchar()functioniscalled,itwaitsuntilakeyispressedandassignsthischaracterasavaluetogetchar function. The value is also echoed on thescreen.

The getchar() function does not require any argument. But a pair of empty parentheses must follow the word getchar. In general, the getchar function is written as

```
variable_name = getchar();
```

where variable_name is a valid C identifier that has been declared as char type.

For example,

```
char letter ;
letter = getchar ();
```

willassignthecharacter'A'tothevariableletterwhenpressingAonthekeyboard. getchar () accepts all the characters upto the pressing of enter key, but reads the first character only.

PUTCHAR( ) FUNCTION

Single character can be displayed using the function putchar ( ). The function putchar ( ) stands for "put character" and uses a argument. The general form of the putchar( ) function is

```
putchar (argument);
```

The argument may be a character variable or an integer value or the character itself contained within a single quote.

Examples

```
void main()
```

```
{ char x =
''A'';
putchar (x);
putchar (''B'');
}+
```

gets() function

gets() accepts any line of string including spaces from the standard Input device (keyboard). gets() stops reading character from keyboard only when the enter key is pressed.

Syntax for gets()

```
gets(variable_name);
```

puts()function

putsdisplays a single / paragraph of text to the standard output device.

Syntax for puts in C :

```
puts(variable_name);
```

```
#include<stdio.h>
#include<conio.h>
void main()

{

chara[20];
gets(a);

puts(a);
```

Sample program :

1. **gets**is used to receive user input to the array. **gets**stops receiving user input only when the **Newline character (Enter Key)** isinterrupted.
2. **puts**is used to display them back in themonitor.

KEY POINTS TO REMEMBER

- program in any programming language. Generally, program development life cycle contains the following 6 phases …. Problem Definition , Problem Design , Coding , ,Testing , Documentation ,Maintenance.
- A computer program is a sequence ofinstructionswritten to perform a specified task with a computer.
- Programming language is a set of grammatical rules for instructing a computer to perform specifictasks.
- Low Level languages are divided in to Machine language and Assemblylanguage.
- Features of the High Level Languages are(i) Simplicity (ii) Naturalness: (iii) Abstraction: (iv) Efficiency: (v) Efficiency: (vi) Compactness (vii) Locality (viii)Extensibility.
- Algorithm is a step-by-step method of solving a problem or makingdecisions.
- Aflowchartisadiagrammaticrepresentationthatillustratesthesequenceofoperationstobe performed to get the solution of aproblem.
- Chasalltheadvantagesofassemblylanguageandallthesignificantfeaturesofmodernhigh-
  - level language. So it is called a “Middle Level Language”.
- TheCprogramminglanguageisastructureorientedprogramminglanguage,developedatBell Laboratories in 1972 by DennisRitchie
- The Features of C programming language are Reliability, Portability, Flexibility, Interactivity, Modularity, Efficiency andEffectiveness.
- AllCprogramsarehavingthefollowingsections/parts:Documentationsection,LinkSection, Definition Section, Global declaration section, Function prototype declaration section, Main function, User defined function definitionsection
- Astatementcausesthecompilertocarryoutsomeaction.Thereare3differenttypesof
  - statements – expression statements compound statements and control statements. Every statement ends with a semicolon.
- Statement may be single or compound (a set of statements). Most of the statements in a C program are expressionstatements.
- C language character set contains the following set of characters... Alphabets ,Digits , Special

Symbols

- Every C program is a collection of instructions and every instruction is a collection of some individual units. Every smallest individual unit of a c program is calledtoken.
- Keywordsarepre-definedwordsinaCcompiler.Eachkeywordismeanttoperformaspecific function in a C program. C language supports 32keywords.
- The value of the constants can not be modified by the program once they are defined. Constants refer to fixed values. The different types of constants are :(i)Integer (ii)Real
    - (iii) Character (iv) String
- A constant is an entity that doesn't change whereas a variable is an entity that maychange.
- Identifierisacollectionofcharacterswhichacts asnameofvariable,function,array, pointer, structure, lableetc...
- Asinglecharacterenclosedwithinapairofsinglequotesiscalledsinglecharacterconstant.
    - Example : 'A' . Sequence of characters enclosed in double quotes. Example:"SALEM"
- Data used in c program is classified into different types based on its properties. In c programminglanguage,datatypecanbedefinedasasetofvalueswithsimilarcharacteristics.

- All the values in a data type have the same properties. The memory size and type of value of a variable are determined by variable data type.

- Primary data types are also called as Built-In datatypes. The following are the primary datatypes in c programming language Integer Data type , Floating Point Datatype ,Double Datatype, Character Datatype
- The void data type means nothing or no value. Generally, void is used to specify a function which does not return anyvalue
- Derived data types are user-defined data types. The derived datatypes are also called as user defineddatatypesorsecondarydatatypes.Incprogramminglanguage,thederiveddatatypes are created using the following concepts...Arrays , Structures ,Unions andEnumeration
- An operator is a symbol, which represents some particular operation that can be performed on some data. Operators operate on constants or variables, which are calledoperands.
- These C operators join individual constants and variables to formexpressions.
- A Modulus operator gives the remainder value. This operator is applied only to integral operands and cannot be applied to float ordouble.
- Operators, functions, constants and variables are combined together to formexpressions.
- Consider the expression A + B * 5. where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is anexpression.
- If all the variables and constants in an expression are of integer type, then this type of expression is called as integerexpression.
- Bit operators are used to perform bit operations. Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (rightshift).
- These operators are used to perform logical operations on the given expressions. There are 3 logicaloperatorsinClanguage.Theyare,logicalAND(&&),logicalOR(||)andlogicalNOT(!).
- Conditional operators return one value if condition is true and returns another value is condition is false. This operator is also called as ternaryoperator.
- Operators that act upon a single operand to produce a new value are called unaryoperator.
- C supports special operators like comma operator, size of operator, pointer operators (&and
  - *) and member selection operators (. and ->).
- Thepriorityorprecedenceinwhichtheoperationsinanarithmeticstatementareperformed is called the hierarchy ofoperations.
- When there are more than one operator with same precedence [priority] then we consider associatively , which indicated the order in‟ which the expression has to be evaluated. It may be either from Left to Right or Right toLeft.
- The combination of backslash(\) and some character which together represent onecharacter
  - are called escape sequences. Ex.: '\n' represents newline character.
- Thecommentlinesaresimplyignoredbythecompiler.Theyarenotexecuted.InC,thereare two types of comments. (i) Single line comments (ii) Multi linecomments

5.

REVIEW QUESTIONS AND PROGRAMS

PART – A (2Marks)

1. Define the term Program.
2. What are the different phases available in program development life cycle.
3. What are the different types of errors ?
4. Define Programming Language.
5. Give the different types of programming languages.
6. Write down the advantages and disadvantages of machine language.
7. Write down the advantages and disadvantages of assembly language.
8. Define Algorithm and Flow Chart.
9. List down the steps in executing a C Program.
10. Why C Language is called as Middle Level Language.
11. Define the term constant. List down the various types of constants.
12. What are the rules to be followed when constructing a floating point constant?
13. What are the data qualifiers? List them.
14. What is the output of the following program?

```
main() {
int x = 10;
printf ("%d %d", ++x, ++x);}
```

15. What is the numerical value of the expression x >= y || y >x ?
16. Without using a conditional statement how to find whether a given number is odd or even?
17. Using a conditional statement , write a statement to find the biggest number between given three numbers a , b and c.
18. What is the difference between a statement and a block?
19. Why do we need different data types?
20. Define the term " Token". Give some examples.
21. s++ or s = s+1, which can be recommended to increment the value by 1 and why?
22. Define the term operator? How the operators are classified?
23. What is initialization? Why it is important?
24. What is a variable and what is the "value" of the variable?
25. When dealing with very small and very large numbers, what steps would you take to improve the accuracy of the calculations?

PART – B (3Marks)

1. What are the features of C Language? Explain
2. What are the characteristics of an algorithm? Explain
3. What are symbols used in flow chart? Explain the purpose of each symbol.

4. What are the limitations of using flow chart? Define them.
5. Write down the history of C Language.
6. Mention any six features of C Language.
7. What is a keyword? What are the features of Keyword? How many keywords are available in C Language?
8. What is typecasting? Give an example.
9. Write a c program to add two numbers without using addition operator
10. Name and describe the basic data types.
11. State the use of const and volatile qualifiers.
12. What do you mean by initialization of a variable? Give examples.
13. What are unary operators? State the purpose of each.
14. What do you mean by hierarchy of operations? Give an example.
15. Why do we use comments in programs? What are the two types of comments?
16. What do the getchar() and putchar() functions do ?

PART – C (5 Marks/ 10 Marks)

1. What is program development life cycle? With a neat diagram, explain different phases of program development life cycle.
2. Explain the general structure of C Program.
3. With a grammatical representation, explain the various steps involved in executing a C Program.
4. Briefly explain about the various types of constants used in C Language with examples.
5. Explain about the different types of data types available in C language?
6. Explain type casting with an example program
7. Briefly explain about the precedence and associativity of operators.
8. State the use of conditional operator with an example.
9. What are the special operators available in C language? Explain them
10. Explain different types of operators available in C language? List them with their uses.
11. State the difference between post increment and pre increment operators? Explain your answer by giving examples.
12. Describe the characteristics and purpose of escape sequence characters.
13. Write a program that will obtain the length and width of a rectangle from the user and calculate its area and perimeter.
14. Write a program to print the size of various data types in C.
15. Given the values of the variables a,b and c, write a program to rotate their values such that a has the value of b, b has the value of c, and c has the value of a.

16. Write a program to print out the largest value of given three numbers without using if statement.
17. Explain about the formatted I/O Statement.
18. Explain about unformatted I/O statements.

764 - Sri Ranganathar Institute of Polytechnic College
1/104 A, Athipalayam, Thudiyalur to Kovilpalayam Road,
Coimbatore – 641110
Tamil Nadu
Phone No: (0422)2904008,2904009
E-mail: sripoly@yahoo.co.in

NAME OF THE FACULTY	**:**	**SASTHI KUMAR C**
COURSE NAME	**:**	**DIPLOMA IN COMPUTER ENGINEERING**
SUBJECT CODE	**:**	**35233**
SEMESTER	**:**	**III**
SUBJECT TITLE	**:**	**C PROGRAMMING**

Unit No.	Topics	No. of Hours
I	PROGRAM DEVELOPMENT AND INTRODUCTION TO C	15
II	DECISION MAKING, ARRAYS AND STRINGS	16
III	FUNCTIONS, STRUCTURES AND UNIONS	16
IV	POINTERS	17
V	FILE MANAGEMENT & PREPROCESSORS	16
	TEST AND REVISION	10
	TOTAL	**90**

DETAILED SYLLABUS

UNIT I PROGRAM DEVELOPMENT & INTRODUCTION TO C		**…. 12 HOURS**
1.1	**Program Algorithm & flow chart:** Program development cycle- Programming language levels & features. Algorithm – Properties & classification of Algorithm, flow chart – symbols, importance & advantage of flow chart.	2 Hrs
1.2	**Introduction C: - History of C** – features of C structure of C program –Compiling, link & run a program. Diagrammatic representation of program execution process.	2 Hrs

1.3.	**Variables, Constants & Data types:** C character set-Tokens- Constants- Key words – identifiers and Variables – Data types and storage – Data type Qualifiers – Declaration of Variables – Assigning values to variable - Declaring variables as constants-Declaration – Variables as volatile- Overflow & under flow of data	3 Hrs
1.4	**C Operators:** Arithmetic, Logical, Assignment Relational, Increment and Decrement, Conditional, Bitwise, Special Operator precedence and Associativity. C expressions – Arithmetic expressions – Evaluation of expressions- Type cast operator	3 Hrs
1.5	**I/O statements:** Formatted input, formatted output, Unformatted I/O statements	2 Hrs
UNIT II DECISION MAKING,ARRAYS and STRINGS HOURS	**…… 13**	
2.1.	**Branching:** Introduction – Simple if statement – if –else – else-if ladder , nested if-else-Switch statement – go statement – Simple programs.	4 Hrs
2.2.	**Looping statements:** While, do-while statements, for loop, break & continue statement – Simple programs	3 Hrs
2.3.	**Arrays:** Declaration and initialization of One dimensional, Two dimensional and character arrays – Accessing array elements – Programs using arrays	3 Hrs
2.4	**Strings :** Declaration and initialization of string variables, Reading String, Writing Strings – String handling functions (strlen(),strcat(),strcmp()) – String manipulation programs	3 Hrs
UNIT III FUNCTIONS, STRUCTURES AND UNIONS HOURS	**…… 13**	
3.1.	**Built –in functions:** Math functions – Console I/O functions – Standard I/O functions – Character Oriented functions – Simple programs	3 Hrs
3.2	**User defined functions:** Defining functions & Needs-, Scope and Life time of Variables, , Function call, return values, Storage classes, Category of function – Recursion – Simple programs	6 Hrs
3.3	**Structures and Unions:** Structure – Definition, initialization, arrays of structures, Arrays with in structures, structures within structures, Structures and functions – Unions – Structure of Union – Difference between Union and structure – Simple programs.	4 Hrs
UNIT IV POINTERS HOURS	**……. 14**	
4.1.	**Pointers:** Definition – advantages of pointers – accessing the address of a variable through pointers - declaring and initializing pointers- pointers expressions, increment and scale factor- array of pointers- pointers and array - pointer and character strings – function arguments – pointers to functions – pointers and structures – programs using pointer.	10 Hrs
4.2.	**Dynamic Memory Management:** introduction – dynamic memory allocation – allocating a block memory (MALLOC) – allocating multiple blocks of memory (CALLOC) – releasing the used space: free – altering the size of a block (REALLOC) – simple programs	4 Hrs

UNIT V FILE MANAGEMENT AND PREPROCESSORS ..... 13 HOURS		
5.1	**File Management:** Introduction-Defining and opening a file-closing a file-Input/ Output operations on files—Error handling during I/O operations –Random Access to files—Programs using files	8 Hrs
5.2	**Command line arguments:** Introduction – argv and argc arguments – Programs using command Line Arguments –Programs	2 Hrs
5.3	**The preprocessor:** Introduction – Macro Substitution, File inclusion, Compiler control directives.	3 Hrs

PART - C

1. Difference between exit control loop and entry control loop with example.
2. Explain for loop with example.
3. Explain nested if statement with examples.
4. Explain switch ... case with example.
5. What is subscripted variable and write its rules.
6. Explain one-D array with example.
7. Explain 1-D and 2-D array processing with example.
8. Write a program to arrange N given numbers in ascending order.
9. Explain string handling functions.

UNIT - III

FUNCTIONS, STRUCTURES AND UNIONS

3.1. Built-in function

Built-in functions are functions not to be written by the programmer. But these functions are available in separate files called **header files**. For using these functions the programmer has to include the header files in their programs. The commonly used built in functions are

(i) Math functions
(ii) Console input/output functions
(iii) Standard input/output functions
(iv) Character oriented functions
(v) Graphical functions

Math functions

Mathematical functions are stored in the header file **math.h**. If we want to use mathematical functions in our program, **math.h** header file must be included in the program. Most of the math functions use floating point numbers. The important functions are

sin()

This function is used to find the sine value of the argument. The argument must be in radians. The syntax is

```
double sin(double x);
```

cos()

This function is used to find the cosine value of the argument. The argument must be in radians. The syntax is

```
double cos(double x);
```

tan()

This function is used to find the tangent value of the argument. The argument must be in radians. The syntax is

```
double tan(double x);
```

exp()

This function is used to find the e to the power of x (e^x). The syntax is

```
double exp(double x);
```

ceil()

This function is used to round up a real number. The syntax is

```
double ceil(double x);
```

Example

x = ceil(93.58)

x = 94

floor()

This function is used to round down a real number. The syntax is

```
double floor(double x);
```


Example

x = floor(93.58)

x = 93

abs()

This function is used to find the absolute value of an integer. The absulate value of x is the value of x without sign. The syntax is

```
int abs(int x);
```

Example

X = abs (-7) Y = abs (7)

X = 7 Y = 7

fabs()

This function is used to find the absolute value of a real number. The absolute value of x is the value of x without sign. The syntax is

```
double fabs(float x);
```

Example

X = fabs (-7.5) Y = fabs (7.5)

X = 7.5 Y = 7.5

pow()

This function is used to find the power of any value. The syntax is

```
double pow(double x, double y);
```

where

x and y be any value

Example

X = pow (2, 3) Y = pow (4, 05)
X = 8 Y = 2.0

sqrt()

This function is used to find the square root value of its argument. The general syntax is

```
double sqrt(int x);
```

Example

X = sqrt(4);
X = 2

Console input/output functions

Console input/output functions are stored in the header file **conio.h**. If we want to use console input/output functions in our program, **conio.h** header file must be included in the program. The important functions are

getch()

This function is used to read a single character from keyboard. But the character typed will not be displayed on the screen. The syntax is

```
int getch()
```

Example

```
char x;
x = getch ();
```

getche()

This function is used to read a single character from keyboard. The character typed will be displayed on the screen. The syntax is

```
int getche();
```


Example

```
char x;
x = getche ();
```

putch()

This function is used to display a character on the computer screen. The general form is

```
int putch (character variable);
```

Example

```
char x;
x = getch();     → A is the input
putch (x);       → A will be displayed on the screen.
```

clrscr()

This function is used to clear the screen and to move the cursor to upper left hand corner of the screen. The general form is

```
clrscr();
```

gotoxy()

This function is used to place the cursor at the desired location on the screen. The general form is

```
void gotoxy(int x, int y);
```

where

x, y - position to move the cursor.

Example

goloxy(10, 20);

The cursor moves to the point (10, 20).

deline()

This function is used to delete the line containing the cursor and moves all lines below it one line up. The general form is

```
deline();
```

wherex()

This function is used to find out the horizontal cursor position. The general form is

```
int wherex();
```

wherey()

This function is used to find out the vertical cursor position. The general form is

```
int wherey();
```

Standard input/output functions

Standard input/output functions are stored in the header file **stdio.h.** If we want to use input/output functions in our program, **stdio.h** header file must be included in the program. The important functions are

scanf () function

Refer Page No.1.57

printf () function

Refer Page No.1.60

gets()

Refer Page No.1.66

puts()

Refer Page No.1.68

getchar()

Refer Page No.1.66

putchar()

Refer Page No.1.67

putc()

Refer Page No.5.04

getc()

Refer Page No.5.03

getw()

Refer Page No.5.05

putw()

Refer Page No.5.06

fprintf()

Refer Page No.5.07

fscanf()

Refer Page No.5.06

Character oriented functions

Character oriented functions are stored in the header file **ctype.h.** It we want to use character oriented functions in our program, **ctype.h** header file must be included in the program. The important functions are

isdigit()

This function is used to check whether a given character is a digit are not. The general form is

```
int isdigit(char c);
```

This function returns a non-zero integer value if its argument is a digit. Else zero.

isalpha()

This function is used to check whether a given character is an alphabet or not. The general form is

```
int isalpha(char c);
```

This function returns a non-zero integer if its argument is an alphabet. Else zero.

isupper()

This function is used to check whether a given alphabet is in upper case or not. The general form is

```
int isupper(char c);
```

This function returns a non-zero integer if its argument is in upper case. Else zero.

islower()

This function is used to check whether a given alphabet is in lower case or not. The general form is

```
int islower(char c);
```

This function returns a non-zero integer if its argument is in lowercase. Else zero.

toupper()

This function is used to convert the lower case letter to upper case. The general form is

```
char toupper(char c);
```

Example

toupper('a')

The output is 'A'.

tolower()

This function is used to convert the upper case letter to lower case. The general form is

```
char tolower(char c);
```

Example

tolower ('Z')

The result will be 'z'

ispunct()

This function is used to check whether a given character is a punctuation character or not. The general form is

```
int ispunct(char c);
```

This function returns a non-zero integer value if its argument is a punctuation character. Else zero.

3.2. User defined functions

A userdefined function or a function is defined as a group of statements written by the programmer to carry out some specific well defined task.

All C-programs consist of one or more functions. Out of this one function should be **main**. This function shows from where the program execution begins. All other functions will be controlled by the function **main**.

Need of user defined functions

The following are the important needs for user defined functions.

(i) Reduced complexity

With the help of functions, large problems can be divided into small problems called sub-problems. Then each sub-problem can be written as a individual function. By interfacing all the individual functions the given problem is solved. This concept reduces the complexity of the programs.

(ii) Reusability

The functions written by a user can be reused by other users without any modification.

(iii) Easy debugging

Since the programs are witten as a collection of functions, locating and correcting errors become easy.

(iv) Extendability

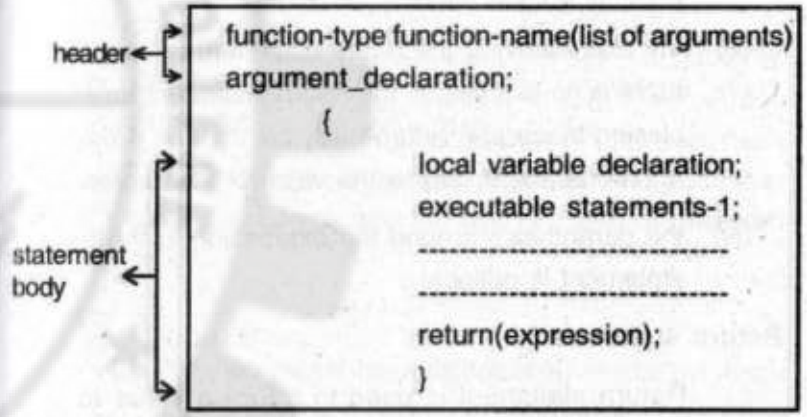
Programs can be easily extended from small to large.

Defining function

A function should be defined before it is used. A function has two parts

(i) function header
(ii) statement body

The general form is



where,

function-type	-	represents the data type of the calculated answer returned by the function
function-name	-	valid C-variable name
list of arguments	-	formal arguments
argument_declaration	-	declaration of formal arguments such as int, char etc.
local_variable_declaration	-	declaration of variables in the statement body.

return - key word to return the calculated answer from the function to other function.

Rules

(i) list of arguments and argument declaration are optional.

(ii) if the function has no list of arguments an empty parenthes's is a must.

(iii) the expression in the return statement is optional. If there is no expression the return statement acts as a closing brace and return back the control. If there is an expression it returns the value of the expression.

(iv) the parenthes's around the expression in the return statement is optional.

Return statement

Return statement is used to return a value to the calling function. The general form is

```
return [OR] return(expression);
```

If there is no expression, the return statement acts as a closing brace and returns back the control to the calling function.

If there is expression, it returns the value of the expression. The parenthes's around the expression is optional.

Example

(i) return; - No return value. But the control is transferred to calling function.

(ii) return(a+b); - The value of the expression a+b is calculated and returned to the calling function.

(iii) return a+b - Valid statement.

Declaring function-type

The **function type** in the function header is optional. If there is no function type in the function header the return statement by default returns an **integer value**.

But with the help of the **option function-type** in the function header we can declare the type of the value returned by the function.

Example

(i) function-name(list of arguments)
this function returns integer value

(ii) **int** function-name(list of arguments)
this function returns integer value

(iii) **float** function-name (list of arguments)
this function returns floating point value.

Function returning nothing

If the function is not to return any value, we can declare the function of type **void**. This tells C not to save any temporary space for a value to be sent by the function. The general form is

```
void function-name (list of arguments)
```

Example of function

```
abc(i, j)
int i,j;
{
        int k;
        k = i+j;
        return(k);
}
```

abc is the name of the function. It has two formal parameters i, j. The statement body has another variable k and we are finding the value of k by adding the values of the formal parameters i and j. The return statement returns the value of k to other function.

Calling a function

A defined function can be called from other functions by specifying its name followed by a list of arguments enclosed within parentheses. The general form is

```
function-name (list of arguments);
```

where

function-name - name of a already defined function.
list of arguments - actual arguments.

Rules

(i) function- name should be the name used in the function definition (called function)

(ii) list of arguments is optional

(iii) if the function call has no arguments, an empty pair of parentheses is a must.

(iv) data type of the arguments should match with the already defined function (called function) arguments.

(v) the called function returns only one value per call.

Function declaration

Function declaration means, declaring the defined function in the main program. The general form is

```
datatype function name ();
```

where,

data type - It should be same as in the function definition.

function name - name of the defined function.

If the defined functions return data type (function type) is void or int, then their is no need to declare the defined function in the main program. But if the data type value is float or char, then it should be declared in the function main.

Example

```
#include<stido.h>
main()
{
        float x, y;
        float abc();            function declaration
        scanf("%f %f", &x, &y);
        printf("%f", abc(x,y));
}

float abc(i,j)          function definition
float i,j;
        {
                float k;
                k = i+j;
                return (k);
        }
```

Note : If the function is defined before main function, then there is no need for function declaration.

Formal and actual arguments

Formal arguments

The arguments present in the **function definition** are called **formal arguments**. These are also called **dummy arguments** because it receives values only from the **calling function**. For example consider the function definition.

```
abc(i, j)
int i, j;
{
        int k;
        k = i+j;
        return(k);
}
```

In this function i and j are called formal or dummy arguments because the values for the arguments i and j are not available. So we cannot execute this function.

Actual arguments

The arguments present in the function calling is called actual arguments. These are called actual arguments because it has values to sent to the already defined function (called function) formal arguments.

Function call

There are two types of function call. They are

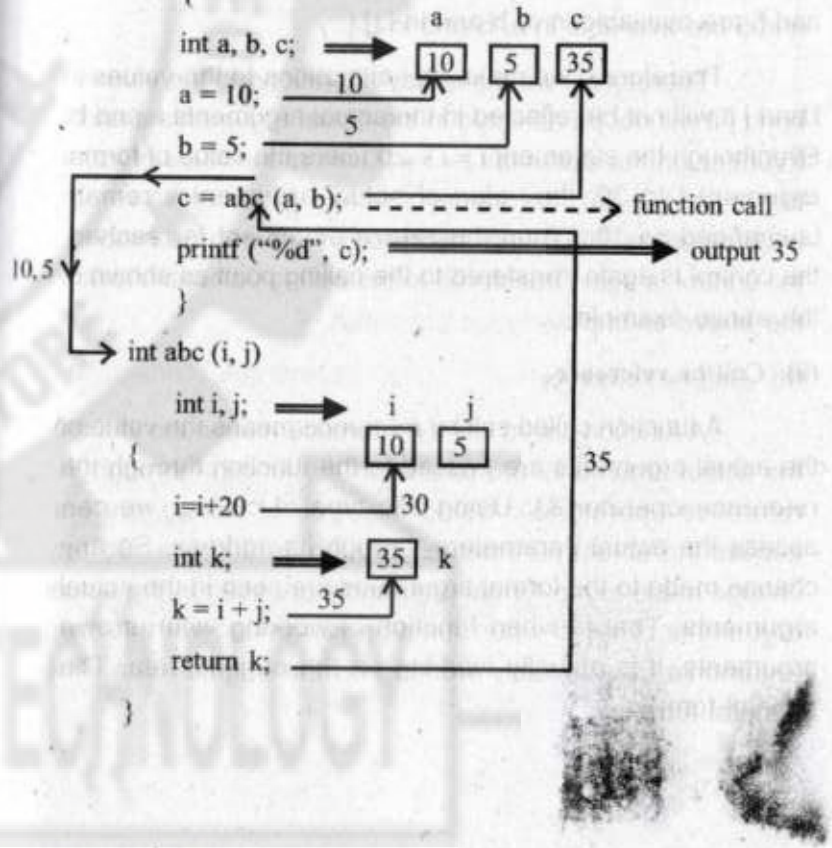
(i) Call by value
(ii) Call by reference

(i) Call be value

A function called call by value means, the values of the actual arguments (data items) are passed as values to the functions formal arguments. That is, the values of the actual arguments are copied to the functions formal arguments. Thus any alternation made to the values sent within the function are not seen in the actual arguments (calling function)

Example

```
#include<stdio.h>
main ()
{
int a, b, c;
a = 10;
b = 5;
c = abc (a, b);
printf ("%d", c);
}
int abc (i, j)
int i, j;
{
i=i+20
int k;
k = i + j;
return k;
}
```



Execution procedure

The execution of the program starts from **main()** function and proceedes normally upto function calling statement.

C = abc (a, b);

When this statement is executed, the program control transfers to the called function **abc (i, j)** along with the values 10,5 (i.e., a = 10, b = 5). That is the function **abc (i, j)** is called with the **actual values 10, 5**. Therefore, the dummy arguments i and j recevies the values 10 and 5. That is, the values 10 and 5 are copied in i, j as i = 10, j = 5.

From the above, it is clear that the actual values 10 and 5 are available in **a, b** and in **i, j.**

Therefore if we make any alterations to the values in i and j it will not be reflected in the actual arguments **a** and **b**. Eventhough the statement i = i + 20 alters the value of formal argument **i** to 30, the value of actual argument **a** remain unchanged as 10. When the **return** statement is reached, the control is again transfered to the calling point as shown in the above example.

(ii) Call be reference

A function called call by reference means the value of the actual arguments are passed to the function through the reference operator(&). Using this type of calling, we can access the actual parameters through its address. So any change made to the formal arguments are seen in the actual arguments. That is when function is working with its own arguments, it is actually working on the original data. The general form is

Function declaration

```
datatype functionname(datatype1&, datatype2&, ...);
```

where

datatype - valid data type such as isnt, float etc.
functionname - name of the defined function reference operator
& - reference operator.

Function definition

```
datatype functionname(datatype1&argname1,
                      datatype2&argname2,.....)
{
    ----------
    ----------
    ----------
}
```

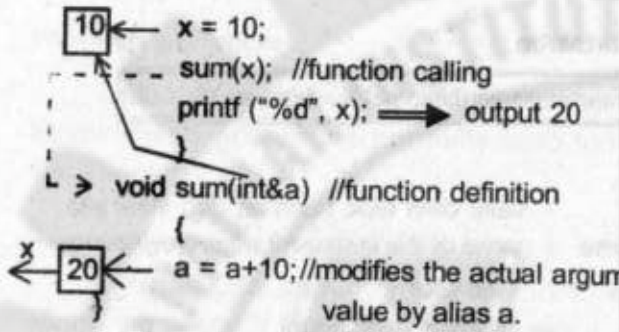
where

datatype - valid data type such as int, float etc
functionname - valid C++ name
& - reference operator

argname1, argname2..... - formal arguments.

Example

```
#include<iostream.h>
void main()
    {
    void sum(int&);        //function declaration
    int x;
```



Execution procedure

The execution of the program starts from main() and proceedes normally up to function calling statement.

sum (x);

when this statement is executed, the program control transfers to the called function sum (int & a). But in the function there is no formal argument to receive the actual argument x value 10, Instead it has an argument to point the address of the actual argument x. So actual argument **X** and formal argument **&a** points to the same location as shown in the above example. Therefore the change made to formal argument **a** will affect the actual argument **X** as shown in the above program.

Category of function

The categories of function are

(i) Functions with no arguments and no return value.
(ii) Functions with no arguments and return value.
(iii) Functions with arguments and return value.

(i) Functions with no arguments and no return values

This is the simplest function. This does not receive any arguments (data) from the calling function and does not return any value to the calling function.

Example

(i)
```
main( )
{
        int a, b;
        - - - - -
        - - - - -
        message( );  /*No actual arguments to send*/
        - - - - -
        - - - - -
}
void message( )    /*No formal arguments to receive*/
{
        - - - - -
        - - - - -
        return;  /*No return value to the called
                        function main*/
}
```

(ii) Write a program to find the sum of two numbers using function with no arguments and no return values.

```
#include<stdio.h>
main()
{
        int sum();
        sum();
}
```

```
void sum()
        {
                int i, j, x;
                scanf("%d %d", &i,&j);
                x=i+j;
                printf("sum=%d",x);
                return;
        }
```

(ii) Functions with no arguments and return values

This function does not receive arguments (data) from the calling function and return the computed value back to the calling function.

Example

(i)
```
main( )
{
        int a, b;
        - - - - -
        - - - - -
        message ();
}
message ()
{
        int x,y value;      /*local variable declaration*/
        - - - - -
        value = x+y
        return(value);   /*this statement returns the
                                value to the calling
}
```


(ii) Write a 'C' program to find a square root of a number using a function.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
    float sqt();
  float y;
    y=sqt();
    printf("\n sqare root value is :%f",y);
    getch();
    return;
}
/*function with no arguments and return value*/
  float sqt()
    {
        float x;
        printf("\n enter the number :");
        scanf("%f",&x);
         return (sqrt (x));
    }
```

Output

enter the number: 9

sqare root value is : 3.000

(iii) Functions with arguments and return values

These functions receive arguments (data) from the calling function and return the computed value back to the calling function.

Example

```
main( )
{
        int a, b;
        - - - - -
        - - - - -
        message (a, b);/*actual arguments to send*/
}
message (x, y)    /*formal arguments to receive*/
int x,y;          /*formal arguments declaration*/
{
        int value;          /*local variable declaration*/
        - - - - -
        value = x+y
        return(value);    /*this statement returns the
                            value to the calling function*/
}
```

Programs

1. **Write a program to add two numbers using function with arguments and return values.**

```
#include <stdio.h>
main()
{
        int sum (int, int) ;
        int i, j, value;
        scanf("%d %d", &i, &j);
        value = sum(i, j);
        printf("sum = %d", value);
}
```



```
sum(x, y)
int x, y;
{
        int z;
        z = x+y;
        return(z);
}
```

1. **Write a program to find the square of n numbers using function with arguments and return values.**

```
#include<stdio.h>
main()
{
    int square (int) ;
    int i, n, value;
    printf("enter the value of n \n");
    scanf("%d", &n);
    for(i=0; i<=n; i++)
    {
        value = square(i);
        printf("%d %d", i, value);
    }
}
square(m)
int m;
{
        int temp;
        temp=m*m;
        return(temp);
}
```

3. **Write a program to find the factorial of a given number using function with arguments and return values.**

```
#include <stdio.h>
main()
{
        int fact (int) ;
        int a, n;
        printf("enter any integer number \n");
        scanf("%d", &n);
        a = fact(n);
        printf("the factorial as n = \n", a);
}
fact(n)
int n;
{
        int value, i;
        value = 1;
        if(n == 0)
                return (value);
        else
                {
                        for(i=1; i<n; i++)
                                value = value * i;
                                return(value);
                }
}
```


4. **Write a program to find whether the given year is leap or not.**

```
#include <stdio.h>
int isleap (int y)
{
        if (y % 4 ==0)
                return 1;
                else
                return 0;
}
 main()
  {
  int year;
  printf("\n enter the year :");
  scanf("%d",&year);
  if (isleap (year))
        printf("\n the given year is leap year");
        else
        printf("\n the given year is not leap year");
  getch();
  }
```

5. **Write a user defined functtion to find the sum of individual digits of a number N and write a main program to call the above function.**

```
#include<stdio.h>
main()
   {
   void digit();
   digit();
   }
   void digit()
```

```
{
int n, sum=0;
scanf, ("%d", &n);
while (n>0)
   {
   sum = sum + n%10;
   n=1/10;
   }
printf("%d", sum);
}
```

Scope and lifetime of variables/Storage calss

Scope of variable means how widely a variable is known among the set of functions in a program. **Life time of variable means** how long a variable retains a given value during the execution of the program.

These two factors depend on the **storage class** of a variable. Therefore **storage class** is defined as a concept that defines the scope and life time of a variable. C has the following storage classes.

(i) automatic variables
(ii) external variables
(ii) static variables
(iv) register variables

(i) Automatic variables

The variables which are declared inside a function are called automatic variables. These are called automatic because, their memory spaces are automatically allocated when the function is called and destroyed automatically when the function is exited. The general form is


```
auto datatype variable1, variable2 ..... variablen;
```

where

auto - keyword to define automatic.

Example

```
void main()
   {
      auto int a, b;
      auto float x, y;
      - - - - - - - - -
      - - - - - - - - -
   }
```

The local variables are given the storage class **auto** by default. So it is not necessary to use the keyword **auto**. Therefore the above example can be written as

```
void main()
   {
      int a, b;
      float x, y;
      - - - - -
      - - - - -
   }
```

Scope of the automatic variable is, it is known only within the function. **Life time** is, it is destroyed when we come out of the function.

(ii) External variables

The variables which are declared outside the functions are called external variables. Since these variables are not declared within a specific function, these are common to all the functions in the program. These variables should be referred by the same name and data type throughout

the program. These variables are alive and active throughout the program.

Definition of external variable

An external variable is defined in the same manner as the ordinary variable. This must appear outside the function. **Variable definition will automatically allocate memory space**. The assignment of initial values can be included within the definition.

Example

```
int a,b;
float x = 6.5;
void main()
  {
    - - - - -
    - - - - -
  }
void abc()
  {
    - - - - -
    - - - - -
  }
```

The variables **a,b** and **x** are declared outside the function and are called **external variables**. These are common to the functions main and abc.

Scope of the external variable is, it is known throughout the program. **Life time** is, it is destroyed when the execution of the program is over.

(iii) Static variables

Static variables are variables which retain the values till the end of the program. The general form is

```
static datatype variable1, variable2 ..... variablen;
```

where

static - key word to define static.

There are two types of static variables. They are

(i) internal static variables
(ii) external static variables

Internal static variables are variables declared inside a function and are available to the function itself.

External static variables are variables declared outside the function and are available to all the functions in the program.

Example

(i) static int a, b;
(ii) static int a = 100;
(iii) static char x1;

(i) **Scope** of the static variable depends whether it is defined internally or externally. **Life time** is, it is destroyed when the execution of the program is over.

(ii) external static variables are available to the functions only within a file.

(iv) Register variables

Registers are special storage available within the computer central processing unit. Since they are part of CPU, operation on registers is faster than using memory. The variables which are stored in the registers are called register variables. The general form is

```
register datatype variable1, variable2.....variablen;
```

where

register - keyword

Example

```
register int a, b;
```

Scope of the register variable is, it is known only within the function. **Life time** is, it is destroyed when we come out of the function.

Recursion

Recursion is a programming technique in which a function calls itself again and again.

A function is said to be recursive, if it calls itselt. A well defined recursive function must satisfy the following two conditions.

(i) There must be a recursive formula and a base value for which the function does not call to itself.

(ii) Each call of the function must execute the recursive formula and the calculated value must move closer to the base value.

Example

Factorial value of a posiitive integer n is denoted by n! and it is calculated as

n! = 1 x 2 x 3 x . . . . n

Base solution for factorial function is 0! = 1

Recursive formula for n! is

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

The difference between the normal function and recursive function is, the normal function will be called by other functions by its name. But the recursive function will be called by itself as long as the condition is satisfied.

Example - Program to find the factorial of a given number.

Solution

n! = 1 if n = 0

n! = (n(n-1)! if n > 0

```
# include <stdio.h>
main ()
{
int fact (int);
int i, a;
scanf ("%d", & i);
a = fact (i);
printf ("%d", a);
}
int fact (int n)
{
int f ;
if (n = = 0)
```

```
return (1) ;
else
f = n * fact (n - 1) ;
return (f) ;
}
```

3.3. Structures and union

Structures

Array is a powerful tools to represent group of data items of same data type with a single name. However if we want to represent a group of data items of different data types with a single name, the array is inadequate to the task. But C language provides a data type named as **structures** to do this task. This **structure is defined as a data type to represent several different types of data with a single name**.

Difference between arrays and structures

(i) All data in an array should be of same data type. But in structures data can be of different data types

(ii) Individual entries in an array are called elements. But in structures individual entries are called members.

Difference in defining ordinary variable and structure variable

An integer variable i in C is defined as

int i;

We don't have to tell C what an int is because C already knows it. But while defininining a structure variable we have to tell C about

(i) how the structure looks like

(ii) the variables

Structure definition

A structure definition contains a keyword **struct** and a user defined tag-field followed by the members of the structure within braces. The general form is

```
struct tag-field
    {
        datatype member1;
        datatype member2;
        ----------
        ----------
        datatype membern;
    };
```

where

struct - keyword to define structure

tag-field - name of the structure - valid C name

datatype - valid datatypes such as int, float etc

Rules

(i) tag-field is the name given to the structure

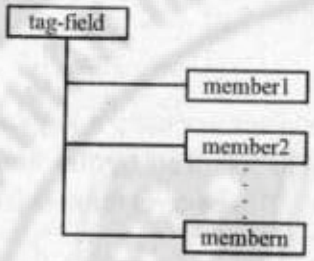
(ii) each member definition should be terminated with semicolon

(iii) since structure definition has compound statements, it should have its own opening and closing braces.

(iv) the semicolon after the closing brace is a must

The structure definition tells C exactly what our structure looks like (format of the structure) as shown below.

struct



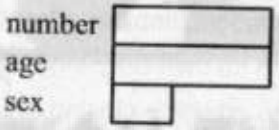
Example

Consider a student information consisting of number, age, sex. The structure definition can be done as followed

```
struct student
    {
        int number;
        int age;
        char sex;
    };
```

This defines a structure with three members namely, number, age and sex of different datatypes. The name of the structure is **student**.

The figure given below shows how this structure format looks like



Note: The members of the structure definition are not variables. So they do not occupy any memory space.

Structure declaration or Variable declaration

Structure declaration means defining variables to the already defined structure. The general form is

```
Struct tag-field variable1, variable2,.....variablen;
```

Where

variable1, variable2,.....variablen are valid C variable names each having n-members. (i.e., member1, member2,...membern)

tag-field - name of the defined structure

Example

Struct student student1, student2, student3;

This declares student1, student2, student3 as structure variables each having three members number, age, sex.

Accessing and giving values to structure members

Dot operator or member operator '.' is used to give data to the sturcture variables individual members. The general form is

```
structure variable . mambername
```

The variable name with a period and the member name is used like any ordinary variable.

Example

(i)
```
struct student
    {
        int number;
        int age;
```

```
        char sex;
    }student1;
```

The above declaration has a variable student1 having three members. The values or data to the members of the variable can be given by any one of the following method

(i) using keyword

```
scanf("%d", student1.number);
scanf("%d",student1.age);
scanf("%c",student1.sex);
```

(ii) using assignment statement

```
student1.number = 1001;
student1.age = 21;
student1.sex = 'm';
```

Note : The importance of member operator is, it gives meaning to the members. For example consider the member name in our example. It has no meaning till it is linked with the variable student1. That is student1.name means name of a student.

(ii) Write a program to find the sum of two complex numbers

```
#include<stdio.h>
main()
{
    struct complex
        {
            float rp;
            float ip;
        };
    struct complex x, y, z ;
```



```
    printf("\n enter the first complex number real part and
imaginary part");
    scanf("%f %f", &x.rp , &x.ip);
    printf("\n enter the second complex number real part
    and imaginary part");
    scanf("%f %f", &y.rp , &y.ip);
    z.rp = x.rp + y.rp;
    z.ip = x.ip + y.ip;
    printf("\n z=%6.2f + 6.2f i", z.rp, z.ip);
}
```

Structure initialization

The members of the structure variable can be assigned initial values. For this the structure should be declared as static. The general form is

static struct tag-field structure variable = {value1, value2, ... valuen};

where

static - storage class
struct - keyword

Example

(i)

```
struct student
    {
        int number;
        int age;
        char sex;
    } static student1 = {100, 21, 'm'};
```

This assigns the value

100 to student1.number
21 to student1.age
m to student1.sex

(ii)

```
struct student
{
    int number;
    int age;
    char sex;
};
static struct student1 = {100, 21, 'M'};
static struct student2 = {101, 21, 'F'};
```

This assigns the value

100 to student1.number
21 to student1.age
M to student1.sex
101 to student2.number
21 to student2.age
F to student2.sex

(iii)

```
struct student
{
    int number;
    int age;

char sex;
} static student1 = {100, 21, 'M'};
- - - - - - - - - - - - - - - - - -
```



```
- - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - -
static struct student student2 = {101, 21, 'F'};
- - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - -
```

This declaration will do the same assignment as in example (ii)

(iv)

```
struct student
{
    int number;
    int age;
    char sex;
};
static struct student1 = {100};
```

Initial value will be assigned to the first member only. C automatically assigns zero to the rest of the members.

Comparsion of structure variables

Comparing two variables of the structure as the way ordinary variables are compared is called comparsion of structure variables. Consider the example given below.

```
struct student
    {
    int number;
    int age;
    char sex;
    };
struct s1 = {100, 21, 'M'};
```

```
struct s2= {101, 20, 'M'};
    - - - - -
    - - - - -
```

s_1 and s_2 are the variables of the structure data type **student**. The variables are given the value.

s1 number = 100	s2.number =101
s1.age = 21	s2.age = 20
s1.sex = M	s2.sex = M

The following operations are allowed between the structure variables s1 and s2.

(i) s1 = s2 ⇒ The member values of s2 variable is assigned to s1 variable.

(ii) s1 = = s2 ⇒ This compares all the member values of s1 and s2. It return 1 if all the values are equal else 0.

(iii) s1! = s2 ⇒ This returns 1 if all the member values are not equal else 0.

Note : This operation will not be supported by all compilers.

Arrays of structures

Array of structure is defined as a process of giving one name to store more than one structure variable. The general form is

```
struct tag-field variable name [size];
```

where

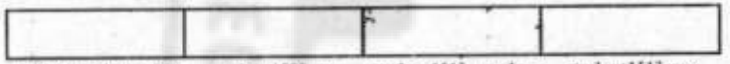
struct - keyword
tag-field - name of the defined structure
size - number of structure variables to store

Example

(i)

```
struct student
  {
    int number;
    int age;
  }
struct student student1[2];
```

This declares **student1** as an array of structures having two elements student1[0], student1[1]. Each element has two members. The figure given below shows the memory occupation.



(ii)

```
struct student
  {
    int number;
    int age;
  };
static struct student student1[2] = {
                                      {100,21},
                                      {101,21}
                                    };
```

This declares student1 as an array of structures having two elements student1[0] and student1[1] and initialize their members as

```
student1[0].number = 100
student1[0].age = 21
student1[1].number = 101
student1[1].age = 21
```

Program

1. **Write a program to assign information about 100 students to a structure and to print out the content of the structure. The members are number, age, sex.**

```
#include <stdio.h>
main()
struct student
  {
     int number;
     int age;
     char sex;
  };
struct student student1[100];
int i;
for(i=0; i<100; i++)
  {
     printf("Enter number, age, sex \n");
     scanf("%d %d %c", &student1[i].number,
    &student1[i].age, &student1[i].sex);
  }
for(i=0; i<100; i++)
  printf("%d %d %c", student[i].number,
     student[i].age, student[i].sex);
}
```


2. **Define a structure data type called 'time' containing three integermembers hour, minute and second. Develop a program that would assign values to the individual members and display the time in the format 16:40:51**

```
#includ<studio.h>
main
struct time
  {
  int hour;
  int miniute;
  int seconds;
  };
  main()
  {
  struct time t;
  scanf ("%d %d %d" &t.hour, &t.miniute; &t.seconds);
  printf ("%d:%d:%d"),t.hour, t.miniute, t.seconds);
  }
```

Arrays within structures

The members of the structure can be defined as array data type. This definition is called arrays within strutures.

Example

```
struct student
{
     char name[20];
     int number;
```

```
int age;
char sex;
int mark[5];
} student1[5];
```

This declares student1 as an array of structures having five elements. In this declaration.

(i) member name is defined as a character array to hold name of maximum 20 characters.

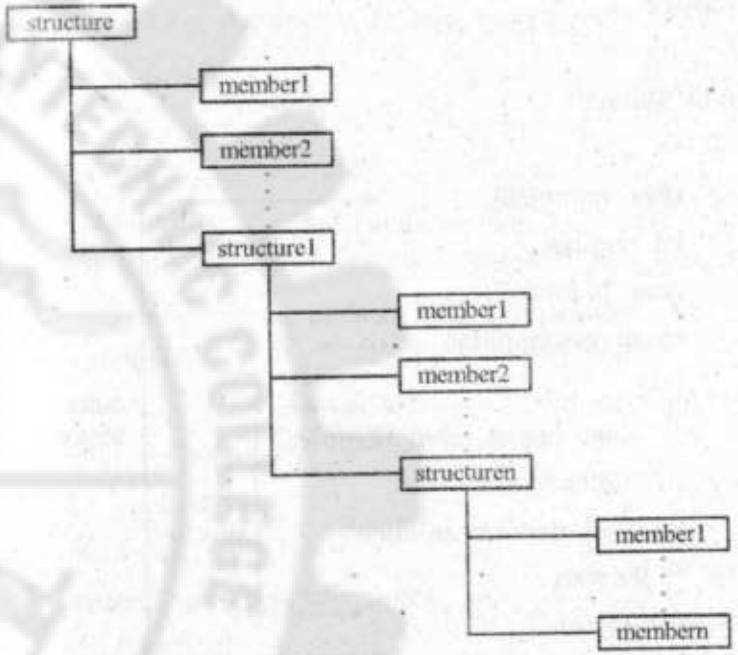
(ii) member mark is defined as a integer array to store five elements (marks) namely mark[0], mark[1].....mark[4].

The members of the structure are accessed by the variables as.

```
student1[0].name
student1[0].number
- - - -
- - - -
student1[0].mark[0]
- - - -
- - - -
student1[0].mark[4]
```

Structures within structures [Nested structure]

When a structure is declared as a member of another structure then it is called structure within structure or nested structure. The figure given below shows the format of the nested structure.



The general forms to access the structure variable is

(i) outerstructure variable.innerstructure1 variable.member name

This is used to access the inner structure1 member

(ii) outerstructure variable.innerstructure1 variable.innerstructure2 variable.membername

This is used to access the inner structure2 member.

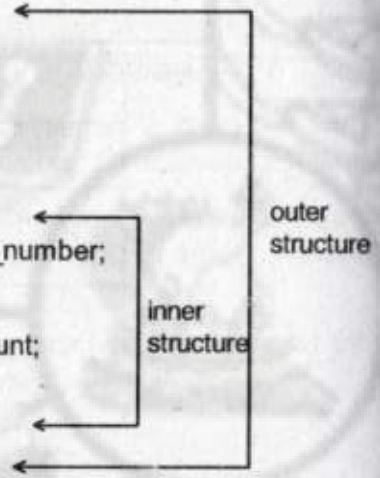
(iii) outerstructure variable.innerstructure1 variable.innerstructure2 variable.- - - - - innerstructurem variable.member name

This is used to access the inner structurem member.

Example

(i)

```
struct student
  {
     char name[20];
     int number;
     char branch[5];
     struct hostel-detail
       {
          int hostel_room_number;
          char food;
          int deposit amount;
       }hostel;
  }student1[10];
```



This declares a structure having four members namely.

name, number, branch and a structure hostel-detail

The outer structure **struct student student1[10]** gives the information about the students college detail and hostel detail. The inner structure struct **hostel-detail** gives the information about the hostel detail of a student.

The members of the structure are accessed as

student1[0].hostel. hostel_room_number - gives the first student hostel room number

student1[7].hostel.food - gives the 8th student type of food in the hostel.

student1[7].number - gives the 8th student number and so on.

(ii) The following declaration is also valid

```
struct hostel-detail
  {
     int hostel-room-number;
     char food;
     int deposit-amount;
  };
struct student
  {
     char name[20];
     int number;
     char branch[5];
     struct hostel-detail hostel; -
  }  student1[10];
```

Program

Write a program using nested structure to access and print the information in the nested structure

```
#include<stdio.h>
main()
  {
     struct hostel_detail
       {
          int hostel_room_number;
          char food;
          int deposit_amount;
       };
```

```
struct student
   {
      char name[20];
      int number;
      char branch[5];
      struct hostel_detail hostel;
   };
static struct student student1 = {"Raju",
                                  101,"CT",10,
                                  'V',2000};
printf("%s %d %s %d %c %d",
         student1.name,
       student1.number,
         student1.branch,
         student1.hostel.hostel_room_number,
           student1.hostel.food,
           student1.hostel.deposit_amount);
}
```

The output will be

Raju 101 CT 10 V 2000

Passing structure as parameter/Structures and function

Passing the entire structure to a function like variables is called passing structure as parameters. The general forms are

(i) Function calling

```
functionname(structurevariablename);
```


where

functionname	-	name of the called function.
structurevariablename	-	variable name of the defined structure (actual argument).

(ii) Function definition

```
datatype functionname(variablename)
struct tag-field variablename;
      {
         ------------------
         ------------------
         return(expression);
      }
```

where

functionname	-	name of the function
struct	-	keyword
tag-field	-	name of the structure to be passed
variablename	-	formal argument which receives the actual argument and it should be of **struct** type.

Example

Program to pass the entire structure to a function and to print it

```
#include <stdio.h>
{
struct student
   {
```

```
        char name [20];
        int number;
        char branch[5];
        };
main ()
{
        static struct student a = {"Raju", 101, "CT"};
        output (a); ⇒ function calling, a structure variable
        }
        void output (b) ⇒ function definition, b formal
                          argument to receive a.
        struct student b;
        {
        printf ("%s, %d, %s", b.name, b.number, b.branch);
        }
}
```

The output will be

Raju 101 CT

Note: The structure variable used as the actual argument and the corresponding formal argument in the called function must be of same type ie **struct type.**

Unions

Union is like a structure data type in which all the members share the same memory area. The general form is

```
union tag-field
    {
        datatype member1;
```



```
        datatype member2;
        - - - - - - - - - -
        - - - - - - - - - -
        datatype membern;
    }variable1, variable2, ..... variablen;
```

where

union - keyword to define union

tag-field - name of the union - valid C name

variable 1.....variablen - valid C variable name

Explanation

Consider the following union declaration

```
union example
    {
        char name[15];
        int number;
    }student;
```

This declares **student** as a variable to the union named **example** having two members. The members are accessed as

student.name

student.number

Because of the property that all members share the same memory space, **the compiler allocates a memory space that is large enough to store the largest variable type in the union.** In our declaration the 15 character

string requires more memory than the integer quantity. So the compiler allocates 15 bytes to the union. This area will be shared by the two variables student.name and student.number.

Therefore at one time we can refer either student.name or student.number depending upon the value stored.

Uses of unions

(i) Unions conserve memory space

(ii) They are useful for applications invloving number of variables, where values need not be assigned to all elements at one time.

Difference between unions and structures

(i) A union can provide storage for one member at a given time. But a structure can provide storage for all members.

(ii) The amount of space allocated for union is based upon the member which requires largest memory space. But in structure each member is given memory space.

(iii) In union in one time we can refer any one variable. But in structure we can refer all the variables.

REVIEW QUESTIONS

PART - A

1. What is built in function?
2. Give any two math functions.
3. Define user defined function.
4. Give any two need for user defined function.
5. What is the use of return statement?
6. What is ment by void function?
7. Give the general form for calling a function.
8. What is formal or dummy arguments?
9. What is actual arguments?
10. Give the two types of function call.
11. What do you mean by call by value?
12. What do you mean by call by reference?
13. Give any two types of function.
14. Define scope of a variable.
15. What is meant by lifetime of a variable.
16. Give the types of storage classes.
17. Define structure.
18. Name the operator used to access the structure members.
19. What is the difference between array and structure?
20. Define structure with in structure.
21. Define union.

764 - Sri Ranganathar Institute of Polytechnic College
1/104 A, Athipalayam, Thudiyalur to Kovilpalayam Road,
Coimbatore – 641110
Tamil Nadu
Phone No: (0422)2904008,2904009
E-mail: sripoly@yahoo.co.in

NAME OF THE FACULTY : **SASTHI KUMAR C**

COURSE NAME : **DIPLOMA IN COMPUTER ENGINEERING**

SUBJECT CODE : **35233**

SEMESTER : **III**

SUBJECT TITLE : **C PROGRAMMING**

Unit No.	Topics	No. of Hours
I	PROGRAM DEVELOPMENT AND INTRODUCTION TO C	15
II	DECISION MAKING, ARRAYS AND STRINGS	16
III	FUNCTIONS, STRUCTURES AND UNIONS	16
IV	POINTERS	17
V	FILE MANAGEMENT & PREPROCESSORS	16
	TEST AND REVISION	10
	TOTAL	**90**

PREPARED BY
C .SASTHI KUMAR MCA., PGDCA. FACULTY/DEPARTMENT OF CSE, SRIPC

DETAILED SYLLABUS

UNIT I PROGRAM DEVELOPMENT & INTRODUCTION TO C …. 12 HOURS		
1.1	**Program Algorithm & flow chart:** Program development cycle- Programming language levels & features. Algorithm – Properties & classification of Algorithm, flow chart – symbols, importance & advantage of flow chart.	2 Hrs
1.2	**Introduction C: - History of C** – features of C structure of C program –Compiling, link & run a program. Diagrammatic representation of program execution process.	2 Hrs
1.3.	**Variables, Constants & Data types:** C character set-Tokens- Constants- Key words – identifiers and Variables – Data types and storage – Data type Qualifiers – Declaration of Variables – Assigning values to variable - Declaring variables as constants-Declaration – Variables as volatile- Overflow & under flow of data	3 Hrs
1.4	**C Operators:** Arithmetic, Logical, Assignment Relational, Increment and Decrement, Conditional, Bitwise, Special Operator precedence and Associativity. C expressions – Arithmetic expressions – Evaluation of expressions- Type cast operator	3 Hrs
1.5	**I/O statements:** Formatted input, formatted output, Unformatted I/O statements	2 Hrs
UNIT II DECISION MAKING,ARRAYS and STRINGS …… 13 HOURS		
2.1.	**Branching:** Introduction – Simple if statement – if –else – else-if ladder , nested if-else-Switch statement – go statement – Simple programs.	4 Hrs
2.2.	**Looping statements:** While, do-while statements, for loop, break & continue statement – Simple programs	3 Hrs
2.3.	**Arrays:** Declaration and initialization of One dimensional, Two dimensional and character arrays – Accessing array elements – Programs using arrays	3 Hrs
2.4	**Strings :** Declaration and initialization of string variables, Reading String, Writing Strings – String handling functions (strlen(),strcat(),strcmp()) – String manipulation programs	3 Hrs
UNIT III FUNCTIONS, STRUCTURES AND UNIONS …… 13 HOURS		
3.1.	**Built –in functions:** Math functions – Console I/O functions – Standard I/O functions – Character Oriented functions – Simple programs	3 Hrs
3.2	**User defined functions:** Defining functions & Needs-, Scope and Life time of Variables, , Function call, return values, Storage classes, Category of function – Recursion – Simple programs	6 Hrs
3.3	**Structures and Unions:** Structure – Definition, initialization, arrays of structures, Arrays with in structures, structures within structures, Structures and functions – Unions – Structure of Union – Difference between Union and structure – Simple programs.	4 Hrs

PREPARED BY

C .SASTHI KUMAR MCA., PGDCA. FACULTY/DEPARTMENT OF CSE, SRIPC

UNIT IV POINTERS ……. 14 HOURS		
4.1.	**Pointers:** Definition – advantages of pointers – accessing the address of a variable through pointers - declaring and initializing pointers- pointers expressions, increment and scale factor- array of pointers- pointers and array - pointer and character strings – function arguments – pointers to functions – pointers and structures – programs using pointer.	10 Hrs
4.2.	**Dynamic Memory Management:** introduction – dynamic memory allocation – allocating a block memory (MALLOC) – allocating multiple blocks of memory (CALLOC) – releasing the used space: free – altering the size of a block (REALLOC) – simple programs	4 Hrs
UNIT V FILE MANAGEMENT AND PREPROCESSORS ..... 13 HOURS		
5.1	**File Management:** Introduction-Defining and opening a file-closing a file-Input/ Output operations on files—Error handling during I/O operations –Random Access to files—Programs using files	8 Hrs
5.2	**Command line arguments:** Introduction – argv and argc arguments – Programs using command Line Arguments –Programs	2 Hrs
5.3	**The preprocessor:** Introduction – Macro Substitution, File inclusion, Compiler control directives.	3 Hrs

UNIT - IV

POINTERS - https://youtu.be/If6DVE616gI?t=1

https://youtu.be/HjRFA9hbrrc

https://youtu.be/R1k3Trk0D8M

PREPARED BY

C .SASTHI KUMAR MCA., PGDCA. FACULTY/DEPARTMENT OF CSE, SRIPC

PART - B

1. Define (a) math function (b) console I/O function.
2. Explain ceil ( ) function.
3. Give the general form of a functions.
4. Give any two need of user defined functions.
5. List the different category of functions.
6. Give the difference between array and structure.
7. Give the general form of structure with in structure.
8. Give the difference between union and structure.

PART - C

1. Explain the functions available in ctype-h.
2. Discuss scope and life time of variables.
3. Explain the different category of functions with example.
4. Explain recursion with example.
5. Explain structure dfinition and initialization.
6. Explain array of structure with example.
7. Explain structure with in structure with example.
8. With example explain how a structure is panel to a function.
9. Explain union with example.

UNIT - IV

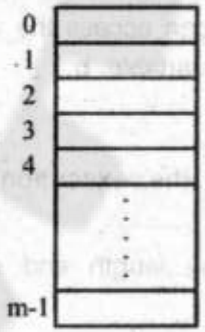
POINTERS

4.1. Pointers

Introduction

Pointers are variables that contain the address of other variables.

As we know, a memory unit is a collection of storage cells which can store information. Each memory cell is given an identification number called address. If a memory can store m-information its address ranges from 0 to m-1 as shown below.

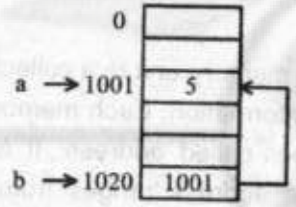


While we are writing program, we used to give our own variable names to the memory cells to store information. But during compilation these user defined variable names are converted into actual memory addresses (number between 0 to m-1). Since memory address are simply

numbers, they can be assigned to some variable name and can be stored in memory like any other variables. The **variables that hold memory addresses are called pointers.**

```
int a = 5;
```

Let us assume that 1001 is the address of the variable **a** and this numeric address is stored in another variable named as **b** having address 1020. The link between the variable a and b can be seen as below.



The name **pointer** because the variable b points to the variable a, so that we can access the value of variable **a** by using the value of variable **b.**

Advantages of pointers

(i) pointers increase the execution speed of the program.

(ii) pointers reduce the length and complexity of a program.

(iii) pointers enable us to access a variable that is defined outside the function.

(iv) pointers are used to pass information back and forth between a function and its reference point.

(v) Pointers reduces wastage of memory while storing character strings.

Accessing the address of variables

The address of any variable can be accessed with the help of the **address operator &** available in C. The general form is

```
&variable
```

Let an integer variable quantity containts a value 79. This can be defined as

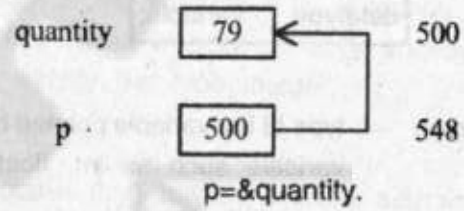
```
int quantity = 79;
```

The address of the variable quantity is got by the address operator & and can be assigned to an integer pointer variable p as

```
p=&quantity.
```

The operator & immediately preceding a variable gives the address of the variable associated with it.

Example




```
p=&quantity.
```

This statement will assign the address 500 (the location of the quantity) to the variable **p**. The **&** operator can be remembered as **address of.**

Example

(i) The following are valid address operators

address operator	reason
(i) &A	pointing a variable
(ii) &X[5]	pointing a array element

(ii) The following are some invalid address operator

address operator	reason
(i) &25	pointing a constant
(ii) &(A+B)	pointing an expression

Note: The & operator can be used only with a simple variable or an array element.

Declaring and initialising pointers

All pointer variables must be declared before it is used in the program like other variables. The general form is

```
datatype * variable;
```

where

datatype - type of the variable pointed by pointer variable such as int, float etc.

* - to identify the vairable as pointer

variable - valid C variable name

Rules

(i) Pointer variables should be a valid C- variable name.

(ii) Asterisk (*) is not the part of the variable name but it is to denote the type of the variable as pointer.

(iii) Data type refers to the type of the variable pointed by the pointer variable.

(iv) Initially the pointer does not point to anything but the programmer must assign it a value.

Example

(i) int *a;

declares the variable **a** as a pointer variable that points to an integer variable.

(ii) float *b;

declares the variable **b** as a pointer variable that points to an floating point variable.

(iii) int *a,b; or int b,*a;

declares the variable **a** as a pointer variable that points to an integer variable and the variable **b** as simply the type integer.

(iv) int *a, *b

declares the variables **a** and **b** as pointer variables that points to an integer variable.

Accessing the address of the variable through pointers

The address of the variables can be got with the help of the address operator **&**. The operator **&** immediately preceding a variable returns the address of the variable associated with it. The general form is

```
pointer variable = &variable;
```

where

Pointer variable - declared pointer variable
& - address operator
variable - declared ordinary variable

Example

(i)
```
int *a;
int x;
a = &x;
```

The memory address of the variable x is assigned to the pointer variable **a**. If 1001 is the address of the variable **x**, **a** got the value 1001.

(ii)
```
int *y;
int b;
b = *y;
```

The **content** of the pointer variable y is assigned to the variable **b**. **Not the memory address.**

(iii)
```
int a[100];
int *b;
b = &a[10]
```

The memory address of the 10th item in the array **a** is assigned to the pointer variable **b**.

(iv) The following are some **invalid pointer** declaration.

declaration	reason for invalidty
(i) int x; x = &a	pointer declaration must have a preceding *
(ii) int *x; x = &50;	& operator can be used only with variables
(iii) int a[100]; int *b; b = &a;	& operator can't be used with array names
(iv) int x,y; int *a; a = &(x+y);	& operator can't be used with expressions
(v) int x; char *y; y = &x;	mixed mode data type is not permitted

Accessing a variable through its pointer

The value of a variable can be accessed using pointer variable. The general form is

```
* pointer variable;
```

where

* - indirection operator
pointer variable - declared pointer variable

Example

```
int *x;   → declare x as pointer variable
int y;
```

y=200;

x=&y; → assigns the address of y to pointer variable x

printf("%d",*x); → prints 200, the value of variable y

printf("%d",x); → prints the address of the variable y

Programs

1. Write a program to display the contents of a pointer variable

```
#include <stdio.h>
main()
  {
    int a;
    int *b;
    a = 100;
    b = &a;
    printf("the content of the pointer
          b = %d \n", *b);
  }
```

The output will be

the content of the pointer b = 100

2. Write a program to display the contents and address of a pointer variable

```
#include <stdio.h>
main()
  {
    int a;
    int *b;
    a = 100;
```



```
    b = &a;
    printf("the content of the pointer
          b = %d \n", *b);
    printf("the address of the pointer
          b = %d \n", b);
  }
```

The output will be

the content of the pointer b = 100

the address of the pointer b = 45422

Pointer expression

Pointer expression is a linear combination of pointer variables, variables and operators (+, -, ++, --). The pointer expression gives either numerical output or address output.

Example

(i) *P + *Q

(ii) * (P + 1)

(iii) * (-- P) + Sum

(iv) P + Q

Where

P, Q - already defined pointer variable

Sum - already defined variable

Pointer assignment

The general form of pointer assignment is

variable = pointer expression

where

variable - either ordinary variable or pointer variable

= - assignment operator

The value of the right hand side of pointer expression is assigned to the variable on the left hand side.

Example

```
int a = 5, b;
int *p, *q;
p = &a;
q = &b;
*q = * p + 10 or b = * p + 10; ⇒ Pointer assignment.
```

In this assignment, the value of the variable b is calculated by the pointer expression *p + 10 and its value 15.

Pointer arithmetic

The following four arithmetic operations can be performed with pointers.

addition +

subtraction -

incrementation ++

decrementation --

The general rule about pointer arithmetic is, pointer performs the operation in bytes of the appropriate storage class or scale factor or length of data type.

Increments and scale factor

Increment means adding one to the pointer variable. Since pointer variables contains the address of other variables, adding one means changing the value (address) to the next value (address). Therefore this change (increment) takes places in terms of the size or length of the data type called the **scale factor** of the variable whose address is stored. The following table gives the various scale factors.

Data type	Scale factor
character	1 byte
integer	2 byte
floats	4 byte
long integer	4 byte
doubles	8 bytes

Example

Consider the following program segment.

```
int a;
int *x,*y;
x = &a;
```

1. Let x and y be pointer variables. In x, address of integer variable a is stored. Let the address of variable a is 1020. Then the operations.

(i) y = x+4

means the scale factor of integer ie 2 byte is added 4 times to the value of x and y pointer takes the value 1020 + 8 ie 1028.

(ii) y = x-4

means the scale factor ie 2 byte is subtracted 4 times to the value of x and Y pointer takes value 1020-8 ie 1012.

(ii) x++

means the scale factor ie 2 byte is added to the value of x ie 1020+2 = 1022.

(iv) x--

means the scale factor ie 2 byte is subtracted from the value of x i.e. 1020-2 = 1018.

2. Consider the following program segment

```
int x,y;
int *a;
x = 20;
a = &x;
```

then the following operations

(i) y = *a++

means the content of the pointer variable i.e. 20 is incremented by 1 and y takes the value 21.

(ii) y = *a--

y takes the value 19.

(iii) y = *a

y takes the value 25.

(iv) y = *a-5

y takes the value 15.

Programs

1. Write a program to display the pointer address before and after incrementation

```
# include    <stdio . h>
main ()
 {
      int    a;
      int    *b;
      a =    100;
      b =    &a;
      printf ("memory address before
                    incrementation  %d \n", *b);
      b++;
      printf ("memory address after
                    incrementation  %d \n", *b);
 }
```

The output will be

memory address before incrementation 45422

memory address after incrementation 45424

2. Write a program to display the content of a pointer variable before and after adding a value 5.

```
#include <stdio.h>
main()
  {
     int a;
     int *b;
     a = 100;
     b = &a;
```

```
    printf("Content of b before adding
                    5 = %d \n", *b);
    b = *b + 5;
    printf("Content of b after adding
                    5 = %d \n", *b);
}
```

The output will be

Content of b before adding 5 = 100

Content of b after adding 5 = 105

Pointers and arrays

Introduction

An **array name** can be defined as a **constant pointer** that points to the address of the first element in the array. Consider the array declaration

int a[10];

On seeing this computer reserves 10 contiguous memory locations by names such as a[0], a[1].....a[9] and sets up a pointer to the array by the array name **a** pointing to the location of a[0]. The location of a[0] (ie address) assigned to the pointer (ie array name 'a') is constant and it cannot be changed. But the values stored in the array can be accessed by means of subscripting ie by adding the appropriate number to the first name a[0]. So

(i) the value stored in first location is accessed by a[0]

(ii) the value stored in second location is accessed by a[0+1]

(iii) the value stored in nth location is accessed by a[0+n]

Difference between array name and pointer

Array name is a constant and it cannot appear on the left side of an assignment operator.

Therefore if **a** is the name of an integer array and x is an integer variable the expression a = &x is not valid.

Pointer is a variable and can appear on the left side of an assignment operator.

Pointers and one dimensional array

Any operation that can be done by array subscripting can be done with pointer. But operation using pointer is faster than that of subscripting. The **principle is,**

(i) the elements of an array are stored contiguously and are all of same type. Therefore only the address of the first element is needed to access the entire array.

(ii) the address of the first element is got by &a[0] where **a** is the name of the array. This is assigned to a **pointer variable x** as x = &a[0].

(iii) the address of the remaining elements are got by adding the **scale factor** to the pointer variable ie x+1 gives the address of a[1], x+2 gives the address of a[2] and x+n gives the address of a[n].

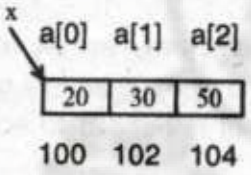
(iv) * x gives the content (value stored) of the first element a[0] *[x+1] gives the content of the second element a[1] and *(x+n) gives the value of the nth element. Where x is the pointer to the array **a**.

Example

Consider the following declaration having three elements in an integer array. Let the staring address of the array is 100. Each element in the array occupies two bytes.

```
static int a[3]   =  {20, 30, 50};
int *x;
int y;
```

The figure given below shows the memory occupation of above declaration.



consider the following operations

(i) x = a; or x = &a[0]'

is a valid assignment, because the property of array is that, it automatically sets an pointer by the name of the array **a**, which points to the address of the first element a[0] ie 100. Therefore the address of a[0] ie 100 is assigned to the pointer variable x.

(ii) x = &a[0];

This will assign the address of the element a[0] to the pointer x ie x = 100

(iii) x = &a[0];
x++;
This will increment the value of x by the scale factor. ie from 100 to 102 which is the address of a[1].

(iv) x = &a[0]
y = *(x+1)
The value stored in the location a[1] is assigned to the variable y. ie y takes the value 30

(v) x = &a[2];
y = *x+5
The value stored in the location a[2] is added with 5 and y takes the value 55.

Program

1. Write a program to display the contents of an array using pointer

```
#include<stdio.h>
main()
   {
      static int a[4] = {10,20,30,40};
      int *b;
      int i;
      b=&a[0];
      printf("Contents of the array \n");
      for(i=0; i<4; i++)
      printf("%d \n", *(b+i));
   }
```

The output is

10
20
30
40

2. Write a program to sum the contents of an array using pointer

```
#include<stdio.h>
main()
    {
        static int a[4] = {10,20,30,40};
        int *b;
        int i;
        int sum=0;
        b=&a[0];
        for(i=0; i<4; i++)
        sum=sum+*(b+i));
        printf("%d \n", sum);
    }
```

The output is 100

Pointers and two dimensional array

In one dimensional array, the address of the first element is assigned to the pointer variable. Like this, in two dimensional array the first element i.e., address of the zeroth row and zeroth column is assigned to the pointer variable.

Explanation

Consider the 2-D array declaration

```
static int a[3] [3] = { {10,20,30},
                        {40,50,60},
                        {70,80,90} };
int *x;
x = &a[0][0];   [OR]   x = a;
```

The logical view of the above declaration is

	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

The physical view of the above logical view is

pointer expression ⇒	x	x+1	x+2	x+3	x+4	x+5	x+6	x+7	x+8
	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]
	10	20	30	40	50	60	70	80	90
Address ⇒	α	α+2	α+4	α+6	α+8	α+10	α+12	α+14	α+16

where α- base address

Address calculation of each array elements using pointer

The address of the element a[i][j] in the 2-D array is got by the formula

x+[(n*i)+j)] .

where

x - pointer variable

n - number of columns in the declared array

The address of a[1][1] is

= x + [(3*1)+1]

= x + 4

The value stored in a [1][1] is *[x + 4) = 50

The address of a [2][1] is

= x + [(3*2)+1]

= x + 7

The value stored in a [2][1] is

*(x+7) = 80

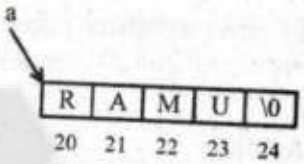
Pointers and character strings

String is a 1-D array of characters terminated by a null character '/0'. Each character occupies 1 byte memory. Each element in the array can be accessed using pointers.

Example

```
char *a;
a = "RAMU";
[OR]
char s[10] = "RAMU" ;
char * a ;
a = & s[0] ;
```

a is a pointer which points the starting address of the string "RAMU" and is shown below.



where

20, 21, 22, 23 and 24 are addresses

Like ordinary one dimensional array we can access the individual characters in the string by **pointer arithmetic**.

Program

1. Write a program to define a string using pointers and to print its content.

```
main()
  {
    char *a;
    a = "RAMU";
    while (*a! = '\0')
      {
        printf ("%c", *a);
        a++;
      }
  }
```

Output is

RAMU

2. Write a program to find the length of a given string

```
#include <stdio.h>
main()
  {
    char a[10];
```

```
    int l;
    char *p;
    gets(a);
    p = &a[0];
    l = 0;
    while(*p != '\0')
      {
        l++;
        p++;
      }
    printf("the length of the string
          = %d \n", l);
}
```

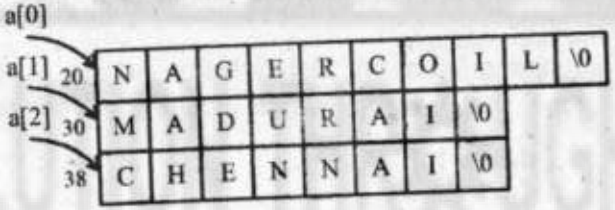
Array of pointers to strings

The collection of pointers which points different strings is called array of pointers to strings.

Example

```
static char *a[3] = { "NAGERCOIL",
                      "MADURAI",
                      "CHENNAI"
                    }
```

In the above example **a** is declared as are array of three pointers to strings. Each pointer points to a particular string as shown below.



Program

Write a program to define array of pointers to strings and print its content.

```
main()
  {
    static *a[3] = {
                     "NAGERCOIL",
                     "MADURAI",
                     "CHENNAI"
                   }
    int i;
    for(i = 0; i < 2; i++)
      printf("%s \n", a[i]);
  }
```

The output will be

NAGERCOIL

MADURAI

CHENNAI

Pointers and functions

In ordinary functions, when a function is called, the actual arguments are passed as **values** to the functions formal arguments. This method is called **call by value.**

But with the help of functions, the actual values to the functions formal arguments can be passed as addresses. Therefore any change made to the content (value) of the addresses will be seen in both the function and the calling function. This method is called **cally by reference.**

Pointers and functions can be divided into two groups. They are

(i) Pointers as function arguments

(ii) Function returning pointer

(i) Pointers as function arguments

Pointers can be used as functions formal arguments. These arguments receive addresses instead of values. Therefore the formal arguments must be declared as pointer variables.

The general form of function definition is

```
datatype functionname (par1, par2....parn)
    datatype *par1, datatype *par2 ...
                datatype *parn;
{
    - - - -
    - - - -
    - - - -
    return (value);
}
```

where

datatype - type of the function
function name - user defined name
par1....parn - pointer variables to receive actual value addresses.

The general form of function calling is

```
functionname (&arg1, &arg2....&argn)
```

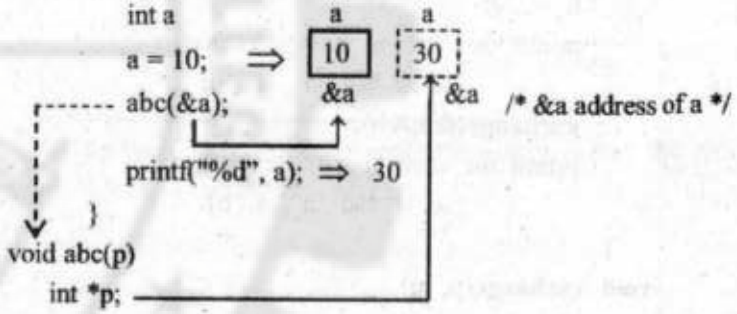

where

functionname - same user defined name as in function definition

&arg1....&argn - addresses of the actual values passed to the formal parameters par1, par2 ..... parn in function definition.

Example

```
main()
{
    int a
    a = 10;  ⇒
    abc(&a);
    printf("%d", a); ⇒ 30
}
void abc(p)
int *p;
{
    *p = *p +20; ⇒ 10 + 20 = 30
    printf ("%d, *p); ⇒ 30
}
```



When the function abc is called, the **address of the variable a** is passed to the function. Inside **abc** the variable p is defined as pointer and p holds the address of the variable **a. Thus the statement**

*p = *p+20

adds 20 to the content of the address p ie 10 and the output will be 30.

Program

Write a program to exchange the contents of two variables using pointers as function arguments

```
#include <stdio.h>
main()
  {
     int a;
     a = 10;
     b = 20;
     printf("the content before exchange
              %d \t %d \n", a, b);
     exchange(&a,&b);
     printf("the content after exchange
              %d \t %d \n", a, b);
  }
void exchange(p, q)
   int *p, *q;
      {
         int temp;
         temp = *p;
         *p = *q;
         *q = temp;
      }
```

The output will be

the content before exchange 10 20
the content after exchange 20 10

Function returning pointers

Functions can return a pointer value like int, float etc. The general form is

```
datatype * functionname (parameter list)
     declaration of parameter ;
     {
     local variable declaration
     - - - - - - - - - - - - - -
     - - - - - - - - - - - - - -
     - - - - - - - - - - - - - -
     return (pointer variable) ;
     }
```

The type of the pointer variable returned and the type of the function must be same.

Example - Program to add two numbers.

```
     int * add(a, b)
          int a, int b ;
     {
     int *p, *q, *r ;
     p = &a ;
     q = &b
     * r = *p+ *q ; or int c = *p+ *q ;
     return (r) ; or return (&c) ;
     }
main ( )
     {
     int * sum ;
```

```
int a = 5, b = 10 ;
sum = add (a, b) ;
printf("%d", * sum) ;
}
```

Pointers and structures

We have seen that pointers are used to point basic data type such as int, float or char. But pointers can be used to point structure data type also. The general form is

```
struct tag-field
  {
     member1;
     member2;
     - - - - - -
     - - - - - -
     membern
  }*structure variable;
```

Where the structure variable is a pointer variable which holds the address of the structure.

The members of the structure can be accessed by any one of the following methods.

(i) (*structure variable).membern

The parentheses required because period(.) has higher precedence than *.

(ii) structure variable ->membern

where

-> is called arrow operator and is made up of minus sign and a greater than.

Example

(i)
```
struct student
  {
     char name[20];
     int number;
  }*ps;
```

This declares a structure named as **student** with two members and **ps** is a pointer to the structure which holds the address of the structure student.

The members of the structure are accessed as

(*ps).name [OR] ps -> name

(*ps).number [OR] ps -> number

(ii)
```
struct student
  {
     char name[20];
     int number;
  };
struct student *ps;
```

This is also a valid declaration.

Program

Write a program to assign values to the member of the structure and print its contents.

```
#include <stdio.h>
main()
  {
     struct student
```

```
    {
        char name[20];
        int number;
    };
    struct student *ps;
    scanf("%s \n", ps-> name);
    scanf("%d \n", &ps -> number);
    printf("Name = %s\n", ps -> name);
    printf("Number = %d \n", ps -> number);
}
```

4.3. Dynamic memory management

Introduction

Memory management is a process of managing the computer memory. In this, the available memory is allocated to the variables in the programs and frees the memory when no longer needed by the programs.

Memory allocation process is divided into

(i) Static memory allocation

(ii) Dynamic memory allocation

(i) Static memory allocation

In this type of allocation, memory space needed by the variables in the program is allocated before loading and executing the program. In this type of allocation the allocated memory space will remain unchanged as long as program is running.

Example

```
int name[1000];
```

This declaration reserves 1000 memory locations in the name **name** for use in the program.

(ii) Dynamic memory allocation

In this type of allocation, the memory space needed by the variables is allocated during the execution of the program.

Advantages of dynamic memory allocation over static

(i) In static memory allocation, if we allocate 1000 memory locations as

```
int name[1000];
```

While running the program only half of this may be used. The rest is unused and idle. It is a wastage of memory.

(ii) If we want to change the size of the array in the program, it is possible by reediting the program. It is a time consuming process.

In dynamic memory allocation the above two problems won't occur because, the memory space for variables is allocated only during execution.

Dynamic memory allocation

Functions used in dynamic memory allocation

Dynamic memory allocation can be implemented in C with the help of the four library functions.

(i) malloc()
(ii) calloc()
(iii) realloc()
(iv) free()

These four functions are in the standard C-library under the header file **stdio.h.** So if we are using the above functions this header file should be included in the program as,

#include<stdbl.h> or #include "stodlb.h"

Allocating a block of memory

malloc() function is used to allocate a contiguous block of memory in bytes. The genral form is

```
pointer variable = (cast-type*) malloc(size);
```

where

pointer variable	-	valid C-pointer variable already defined
cast-type	-	type of the pointer return by malloc( ) such as int. char etc.
malloc	-	keyword
size	-	required size of the memory in bytes

The above function allocates memory of size **size** and returns the starting address of the memory through **pointer variable** of type **cast-type.** The allocated area is **not filled with zeroes.**

Example

(i) int *y;
y = (int *) malloc(20);

On execution of this function 20 bytes of memory are allocated and the starting address of the first byte is assigned to the pointer y of type 'int'.

(ii) int *y;
y = (int *) malloc(10 * sizeof(int);

On execution of this function 10 times the size of an 'int' ie 10 x 2 = 20 bytes is allocated and the starting address of the first byte is assigned to the pointer y of type 'int'

(iii)
```
stuct student
    {
int regno; ⇒  size2 byte
int age;   ⇒  size2 byte
char sex;  ⇒ size 1 byte
} stu;     → structure variable
void main()
{
student * x ; → pointer of type structure
                    student.
- - - - -
- - - - -
x = (student * ) malloc (size of (student));
}
```

When **malloc()** statement is executed, a memory area of size equal to the size of structure student (5 bytes) is allocated and the starting address of the first byte is

assigned to the pointer variable x of data type student.

Note: If the allocation is success it returns the starting address else NULL.

Allocation of multiple blocks

calloc() function is used to allocate multiple blocks of contiguous memory. All the blocks are of same size. The general form is

pointer variable = (cast - type *) calloc(n, size);

pointer variable	-	valid C-pointer variable already defined
cast-type	-	type of the pointer return by calloc( ) such as int. char etc.
calloc	-	keyword
n	-	number of blocks
size	-	required size of memory in bytes

The above function allocates n blocks of memory space of **size** bytes. The address of the first byte of the allocated area is returned through the **pointer - variable** of type **cast-type.** The allocated area is filled **with zeros.**

Example

```
y = (int *) calloc(3,10);
```

On execution the function allocates 3 memory blocks of size 10 bytes and returns the starting address of the area though the pointer y of type "int".

Releasing the used memory space

free() function is used to free (release) the block of unused memory.

Example

```
y = (int *) malloc(10);
free(y);
```

First statement allocates memory space of 10 bytes and returns the starting address of the allocated memory through the pointer variable y. The second statement frees the allocated memory.

Altering the size of the block

realloc() function is used to increase or decrease the size of the memory already allocated. The general form is

pointer variable = realloc(old pointer variable, new size);

where

pointer variable	-	valid C-variable
realloc	-	keyword
old pointer variable	-	name of pointer variable already defined
new size	-	size of the new memory area needed

The above function allocates a new memory space of **new size** bytes. If the allocation is successful it returns the address of the new area through the pointer. If the allocation is unsuccessful it returns NULL and the original block data is lost (freed)

Example

```
y = (int *) malloc (20);
y = realloc (y,30);
```

First statement allocates memory spcae of size 20 bytes and return the starting address of the memory through the pointer y. The second statement reallocates (increases) the already allocated space to 30 bytes.

REVIEW QUESTIONS

PART - A

1. Define pointer.
2. Give any one advantage of pointer.
3. Give the address operator.
4. Give the general form to access the values & variable using provides.
5. Give the general form to define pointer variable.
6. Give the list of operators that can be used in the pointer expression.
7. Name the operator to access the member of a structure variable defined a pointer.
8. Define memory management.
9. Define dynamic memory allocation.
10. What are the two types of memory allocation?
11. Name any two memory allocation functions.
12. What is the use of realloc ( )?
13. What is the use of calloc ( )?
14. What is the use of free ( )?

15. What is the are of malloc ( )?
16. Give any pme relation between pointer and array.
17. What are the operation that can be done using pointers?

PART - B

1. Give any three advantages of pointers.
2. How pointer variables are declared?
3. How to access the address of a variable through pointers?
4. Explain pointer expression.
5. Give the general form of function definition that user pointers as its arguments.
6. Give the general form of structure definition that user pointers.

PART - C

1. Explain declaration and initialization of a pointer with example.
2. Explain increment and scale factor in pointers.
3. Discuss pointers and one dimensional array.
4. With example explain pointer and characters string.
5. With example explain pointers as function arguments.
6. Explain the different functions used in dynamic memory allocation with example.

764 - Sri Ranganathar Institute of Polytechnic College
1/104 A, Athipalayam, Thudiyalur to Kovilpalayam Road,
Coimbatore – 641110
Tamil Nadu
Phone No: (0422)2904008,2904009
E-mail: sripoly@yahoo.co.in

NAME OF THE FACULTY	:	**SASTHI KUMAR C**
COURSE NAME	:	**1052 , DIPLOMA IN COMPUTER ENGINEERING**
SUBJECT CODE	:	**35233**
YEAR / SEMESTER	:	**II/III**
SUBJECT TITLE	:	**C PROGRAMMING**

Unit No.	Topics	No. of Hours
I	PROGRAM DEVELOPMENT AND INTRODUCTION TO C	15
II	DECISION MAKING, ARRAYS AND STRINGS	16
III	FUNCTIONS, STRUCTURES AND UNIONS	16
IV	POINTERS	17
V	FILE MANAGEMENT & PREPROCESSORS	16
	TEST AND REVISION	10
	TOTAL	**90**

DETAILED SYLLABUS

UNIT I PROGRAM DEVELOPMENT & INTRODUCTION TO C		**…. 12 HOURS**
1.1	**Program Algorithm & flow chart:** Program development cycle- Programming language levels & features. Algorithm – Properties & classification of Algorithm, flow chart – symbols, importance & advantage of flow chart.	2 Hrs
1.2	**Introduction C: - History of C** – features of C structure of C program –Compiling, link & run a program. Diagrammatic representation of program execution process.	2 Hrs

1.3.	**Variables, Constants & Data types:** C character set-Tokens- Constants- Key words – identifiers and Variables – Data types and storage – Data type Qualifiers – Declaration of Variables – Assigning values to variable - Declaring variables as constants-Declaration – Variables as volatile- Overflow & under flow of data	3 Hrs
1.4	**C Operators:** Arithmetic, Logical, Assignment Relational, Increment and Decrement, Conditional, Bitwise, Special Operator precedence and Associativity. C expressions – Arithmetic expressions – Evaluation of expressions- Type cast operator	3 Hrs
1.5	**I/O statements:** Formatted input, formatted output, Unformatted I/O statements	2 Hrs
UNIT II DECISION MAKING,ARRAYS and STRINGS HOURS		**…… 13**
2.1.	**Branching:** Introduction – Simple if statement – if –else – else-if ladder , nested if-else-Switch statement – go statement – Simple programs.	4 Hrs
2.2.	**Looping statements:** While, do-while statements, for loop, break & continue statement – Simple programs	3 Hrs
2.3.	**Arrays:** Declaration and initialization of One dimensional, Two dimensional and character arrays – Accessing array elements – Programs using arrays	3 Hrs
2.4	**Strings :** Declaration and initialization of string variables, Reading String, Writing Strings – String handling functions (strlen(),strcat(),strcmp()) – String manipulation programs	3 Hrs
UNIT III FUNCTIONS, STRUCTURES AND UNIONS HOURS		**…… 13**
3.1.	**Built –in functions:** Math functions – Console I/O functions – Standard I/O functions – Character Oriented functions – Simple programs	3 Hrs
3.2	**User defined functions:** Defining functions & Needs-, Scope and Life time of Variables, , Function call, return values, Storage classes, Category of function – Recursion – Simple programs	6 Hrs
3.3	**Structures and Unions:** Structure – Definition, initialization, arrays of structures, Arrays with in structures, structures within structures, Structures and functions – Unions – Structure of Union – Difference between Union and structure – Simple programs.	4 Hrs
UNIT IV POINTERS HOURS		**……. 14**
4.1.	**Pointers:** Definition – advantages of pointers – accessing the address of a variable through pointers - declaring and initializing pointers- pointers expressions, increment and scale factor- array of pointers- pointers and array - pointer and character strings – function arguments – pointers to functions – pointers and structures – programs using pointer.	10 Hrs
4.2.	**Dynamic Memory Management:** introduction – dynamic memory allocation – allocating a block memory (MALLOC) – allocating multiple blocks of memory (CALLOC) – releasing the used space: free – altering the size of a block (REALLOC) – simple programs	4 Hrs

UNIT V FILE MANAGEMENT AND PREPROCESSORS ..... 13 HOURS		
5.1	**File Management:** Introduction-Defining and opening a file-closing a file-Input/ Output operations on files—Error handling during I/O operations –Random Access to files—Programs using files	8 Hrs
5.2	**Command line arguments:** Introduction – argv and argc arguments – Programs using command Line Arguments –Programs	2 Hrs
5.3	**The preprocessor:** Introduction – Macro Substitution, File inclusion, Compiler control directives.	3 Hrs

VIDEO'S URL

UNIT –V FILE MANAGEMENT AND PREPROCESSORS

FILE MANAGEMENT	**https://youtu.be/XZSsp10FYtc**
Input/ Output operations on files	**https://youtu.be/EyRHGQgpVEE?t=171**
Error handling during I/O operations	**https://youtu.be/rMfp6gKR_kk?t=1**
Random Access to files	**https://youtu.be/SFZI6rtm0fo**
Command line arguments	**https://youtu.be/s9Mo7Y6dX3I?t=1**
The preprocessor	**https://youtu.be/zImgTJTrHRw**
	https://youtu.be/yN6GI1iioFM?t=1
Macro Substitution	**https://youtu.be/9GSbVZ7XYM0**
File inclusion	**https://youtu.be/qn2z06MGwmI**
Compiler control directives	**https://youtu.be/509GC9oKk-Q?t=49**

UNIT - V

FILE MANAGEMENT AND PREPROCESSOR

5.1. File management

Introduction

The **scanf** and **printf** functions are used to read and write data to and from the computers through the console. If computers deal with large amount of data as input and output (read, write) then both methods are not efficient. The best way is to use **files** to give data to the computers and get data from the computers.

Defining a file

A file is defined as **FILE** in the header file **stdio.h**. Therefore all files should be declared as type **FILE** before they are used. The general form is

```
FILE *printer variable;
```

where

FILE - data type

pointer variable - pointer to the data type FILE

Opening a file

Before we read from a file or write to a file, we must open the file. Opening establishes a link between the program and the operating system. The general form is


```
FILE *pointer variable;
pointer variable = fopen("filename", "mode");
```

where

(i) pointer variable - variable which contains the address of the type FILE

(ii) file name - name of the file

(iii) mode may be any one of the following

r : open the file for reading only
w : open the file for writing only
a : open the file for appending
r+ : open the file for reading and writing
w+ : open the file for reading and writing
a+ : open the file for reading and appending

Examples

(i) FILE *a;

a = fopen("mydata", "r");

The file **mydata** is opened for reading. If the file does not present an error will occur.

(ii) FILE *b;

b = fopen("test", "w");

The file **test** is opened for writing. If the file contains data, then it is deleted and new file will be created.

(iii) FILE *a, *b;

a = fopen("mydata", "r");
b = fopen("test", "w");

Closing the file

An opened file must be closed after all operations on it have been completed. The general form is

fclose(pointer variable);

where

pointer variable - pointer variable usedduring opening the file

Example

```
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
FILE * a, *b;
a = fopen("mydata", "r");
b = fopen("test", "w");
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
fclose(a);
fclose(b);
```

Input/Output operations on files

Once a file is opened, reading out of or writing to it is carried out using input/output functions present in the **stdio.h** header file. They are

(i) getc() (ii) putc() (iii) getw()
(iv) putw() (v) fprintf() (vi) fscanf()

(i) getc()

This function is used to read a single character from a file that has been opened in read mode. The reading stops when end of file (EOF) is reached. The general form is

c=getc(pointer variable);

where

c - variable which receives the character
pointer variable - pointer which contains the address of the FILE

Example

```
file *a;
char c;
a=fopen("mydata","r");
- - - - - - - - - - - -
c=getc (a);
while (c!=EOF)
  {
     putchar(c);
  }
```

(ii) putc()

This function is used to write a single character into a file that has been opened in write mode. The general form is

putc(c,pointer variable);

where

c - character to be written into the file
pointer variable - pointer which contains the address of the FILE

Example

```
file *a;
char x;
a=fopen("mydata","w");
- - - - - - - - - - - -
- - - - - - - - - - - -
putc(x,a);
```

(iii) getw()

This function is used to read an **integer** value from a file that has been opened in read mode. The reading stops when end of file (EOF) is reached. The general form is

```
getw(pointer variable);
```

where

pointer variable - pointer which contains the address of the FILE

Example

```
file *a;
int c;
a=fopen("mydata","r");
- - - - - - - - - - - -
- - - - - - - - - - - -
c=getw(a);
while (c!=EOF)
  {
    printf("%d\n",c);
    - - - - - - - -
    - - - - - - - -
  }
```


(iv) putw()

This function is used to write an **integer** value into a file that has been opened in write mode. The general form is

```
putw(c,pointer variable);
```

where

c - integer value to be written into the file

pointer variable - pointer which contains the address of the FILE

Example

```
file *a;
int i;
- - - - - - - - - - - -
a=fopen("mydata","w");
- - - - - - - - - - - -
putw(i,a);
- - - - - - - - - - - -
```

(v) fscanf()

This function is used to read data from a file. The general form is

```
fscanf(pointer variable, "control string", list);
```

where

pointer variable - pointer which contains the address of the FILE

control string - format commands such as %s, %d, %f etc.

list - list of variables to be read from the file.

Example

```
FILE *a;
int x,y;
a = fopen("mydata", "r");
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
fscanf (a, "%d %d", &x, &y);
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
```

The statement fscanf reads the items x and y from the file specified by the pointer **a** according to the format %d, %d.

(vi) fprintf()

This function is used to write data into a file. The general form is

```
fprintf(pointer variable, "control string", list);
```

where

able - pointer which contains the address of the file

control string - format commands such as %s, %d, %f etc.

list - list of variables to be written on the file

Example

```
FILE *a;
int x,y;
a = fopen("mydata", "w");
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
fprintf(a, "%d %d", x,y);
- - - - - - - - - - - - - -
- - - - - - - - - - - - - -
```

The statement fprintf writes the items x and y in the file specified by the pointer **a** according to the format %d %d.

Error handling during I/O operation

An error may occur during input/output operations on a file. Some of the error situations are

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Trying to write to a write-protected file.

The following are some important error handling functions

(i) feof() (ii) ferror()

(i) feof()

This function is used to test whether the end of the file is reached or not. The general form is

feof(pointer variable)

where

pointer variable - pointer which contains the address of the file

This function returns a non-zero value if the end of file is reached else zero.

Example

```
file *a;
- - - - - - - - - - - -
- - - - - - - - - - - -
a=fopen("mydata","r");
- - - - - - - - - - - -
- - - - - - - - - - - -
if(feof(a))
  {
     printf("end of file");
  }
```

(ii) ferror()

This function is used to test a given file for an error. The general form is

ferror(pointer variable)

where

pointer variable - pointer which contains the address of the file

This function returns a non-zero value if an error is encountred else zero.

Example

```
file *a;
- - - - - - - - - - - -
- - - - - - - - - - - -
if(ferror(a))
  {
     printf("error");
  }
```

Program

1. Write a program to create and display a file named as "employee" to store information such as name, age and salary of five employees .

```
#include <stdio.h>
main()
    {
      FILE *ptr;
      char name[20];
      int age;
      int salary;
      ptr = fopen("employee", "w");
      for(i=0; i<5; i++)
        {
          printf("\n enter the name, age
                 and salary \n");
          scanf("%s %d %d", name,
                 &age, &salary);
          fprintf(ptr, "%s %d %d", name,
                 age, salary);
        }
      fclose(ptr);
```

```
    ptr = fopen("employee", "r");
    while(feof(ptr) == 0)
      {
        fscanf(ptr, "%s %d %d", name,
                &age, &salary);
        printf("%s %d %d \n", name,
                age, salary);
      }
    fclose(ptr);
}
```

2. Write a program to create a file ptr and copy the content of ptr to another file prt1.

```
#include<stdio.h>
main()
{
   FILE *ptr,*ptr1;
   char name[20];
   int age;
   int salary;
   ptr = fopen("employee", "w");
   for(i=0; i<5; i++)
     {
       printf("\n enter the name, age
                and salary \n");
       scanf("%s %d %d", name,
                &age, &salary);
       fprintf(ptr, "%s %d %d", name,
                age, salary);
     }
   fclose(ptr);
```



```
    ptr = fopen("employee", "r");
    ptr1= fopen("employee1", "w");
    while(feof(ptr) == 0)
      {
        fscanf(ptr, "%s %d %d", name,
                &age, &salary);
        fprintf(ptr1, "%s %d %d", name,
                age, salary);
      }
    fclose(ptr);
}
```

3. Write a program to create a date file with the following details of employees

   Employee number
   Basicpay
   D.A. and total income for each employee (DA=51%of BP)

```
#include<stdio.h>
main()
{
   FILE *ptr;
   int emp_no;
   float BP,DA,T_income;
   ptr = fopen("employee", "w");
   for(i=0; i<5; i++)
     {
       printf("\n enter the name, age
                and salary \n");
       scanf("%d %f", &emp_no,&BP);
       fprintf(ptr,"%d %f", emp_no,BP);
     }
```

```
    fclose(ptr);
    ptr = fopen("employee", "r");
    while(feof(ptr) == 0)
        {
            fscanf(ptr, "%d %f", &emp_no,&BP);
            DA=BP*51/100;
            T_income=BP+DA;
            printf("%d %f %f %f \n", emp_no,
                        BP,DA,T_income);
        }
    fclose(ptr);
}
```

Random access files

Reading and writing operations on any file are normally sequential. This is achieved by a **system controlled pointer** which points the position immediately after the last character is read or written.

Random access files are files in which the user or programmer can move the file pointer randomally and can do read or write optionas. The functions given below are used to move the file pointer randomally

(i) ftell (ii) rewind (iii) fseek

(i) ftell()

This function is used to return the current position of the pointer in the given file. The general form is

```
n = ftell(pointer variable);
```


where

ftell	-	keyword
pointer variable	-	pointer which contains the address of the file
n	-	numeric variable to receive the position

The return position is always relative to the beginning of the file and is always a **long integer**.

Example

```
FILE *ptr;
long n;
- - - - - - - -
- - - - - - - -
n = ftell(ptr)
- - - - - - - -
- - - - - - - -
```

(ii) rewind()

This function is used to reset the position of the pointer to be beginning of the given file. The general form is

```
rewind(pointer variable);
```

where

rewind	-	keyword
pointer variable	-	pointer which contains the address of the file

Example

```
FILE *ptr;
long n;
- - - - - - - -
```

```
rewind(ptr);
n = ftell(ptr)
```

The above declaration would assign o to n because the file position has been set to the start of the file.

(iii) fseek()

This function is used to move the file pointer to any desired location within the file. The general form is

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

where

pointer variable - pointer which contains address of the file

offset - this gives the number of positions to be moved from the location given in **position**. It must be a **long integer.**

position - it is a integer number **position** can take any one of the following values

values	meaning
0	Begining of the file
1	Current position
2	End of the file

Example

(i) fseek (ptr, 0L, 0) - go to the beginning

(ii) fseek (ptr, 4, 0) - move to fourth byte in the file from the beginning.

Program

Write a program to create a file named as "employee" and to display the 2nd record. The file fields are name, age and salary of 5 employees.

```
#include<stdio.h>
#include<conio.h>
struct emp
   {
   char name[30];
   int age;
   float sal;
   } e1;
void main()
   {
   int n;
   FILE *fp;
   fp=fopen("employee","w");
   printf("\enter the name, age, salary \n");
   for (int i=0; i<5; i++)
      {
   scaf("%s%d%f", e1.name, &e1.age, &e1.sal);
   fprint (fp,"%d%d%f", e1.name, e1.age, e1.sal);
      }
   fclose (fp);
   fp=fopen ("employee", "r");
   fseek (fp, 2, 0);
   printf("%s",e1.name);
   printf("%d",e1.age);
   printf("%f",e1.sal);
   fclose(fp);
   getch();
   }
```

5.2. Command line arguments

Command line arguments are arguments that are passed from the command line or prompt to the **main function**. The general form of main function is

```
main (argc, *argv[ ])
in argc, char * argv[ ];
{
- - - -
- - - -
}
```

where

argc - an integer variable and it contains the number of parameters passed to main function from command line.

argv - an array of pointers to characters. Each string in this array contains the parameters passed to main function from command line

The general form for running the program and to give parameter from the command line is

```
C> programname string1 string2.....stringn
```

where

programname - name of the executable program ie with **.exe**

string1, string2, ... stringn - list of arguments

When this command is executed the variable **argc** automatically counts the number of arguments on the command line and it takes the value argc=(n+1). The pointer

variable argv takes the value of argc i.e. n+1 as its **size**. The first location argv[0] contains the first parameter **programname**, the second location contains the string1, the thrid location contains string2 and so on. That is

```
argc    = n+1
argv[0] = programname
argv[1] = string1
argv[2] = string2
- - - - - - - - - -
- - - - - - - - - -
argv[n] = stringn
```

Example

Let **prg** be the executable file. Let **GOD** and **LOVES** be the values to pass. The command line argument is

```
C> prg.exe GOD LOVES
```

The main() function takes the form

```
main(argc, argv)
int argc, char * argv[];
{
- - - - - - - -
- - - - - - - -
}
```

In this argc = 3

```
argv[0] = prg.exe
argv[1] = GOD
argv[2] = LOVES
```

Program :

1. Write a program to add two numbers using command arguments

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void main (int argc, char*argv[ ])
{
int a,b,c;
a=atoi (argv[1]);// converting string to int
b=atoi (argv[2]); // converting string to int
c=a+b;
printf("%d",c);
getch();
}
```

Execution procedure :

Let add be the name of the program.

```
C>add 10 20
```

Here

argc = 3
argv[0]= add
argv[1]= 10
argv[2]= 20

2. Write a program to sort any data file using command line arguments and to store the sorted data in another file.

```
#include<stdio.h>
main(argc, argv)
```

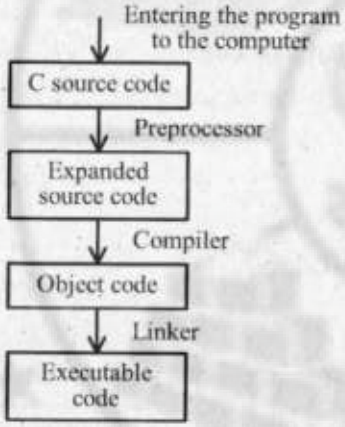


```
int argc;
char *argv[];
  {
    FILE *fptr1, *fptr2;
    int i, a[100], item;
    fptr1 = fopen(argv[1], "r");
    i = 0;
while(!feof(fptr1)
{
fscanf(fptr1, "&d", &item);
a[i++] = item;
}
fclose (fptr1);
for(j=0; j<i-2; j++)
for(k=j+1; k<i-1; k++)
if(a[j] > a[k])
{
t=a[j];
a[j] = a[k];
a[k] = t;
}
fptr2 = fopen(argv[2]; "w");
for(j=0; j<i-1; j++)
{
fprintf(argv[2], "%d \n", a[j]);
}
fclose(ptr2);
}
```

5.3. The preprocessor

Introduction

"Preprocessor is a process that reads our source program and performs some operation on it before it is passed to the computer". This facility is not available in many other high level languages. The flow-diagram given below shows the various steps from writing a C program till its execution.



Preprocessor commands can be considered as a language with in C language. It operates under the control of preprocessor command line or directives. The preprocessor directives are divided into three types. They are,

(i) Macro substitution directive
(ii) File inclusion directive
(iii) Compiler control directive

Advantages

(i) Programs easier to develop
(ii) Easier to read
(iii) Easier to modify
(iv) Portable.

Rules for writing preprocessor directive

(i) it should begin with # symbol from column one.
(ii) semicolon is not needed at the end.
(iii) only one directive can appear on a line.
(iv) it can appear at any place in the source code. But commonly placed before main() or before the beginning of a particular function.

Macro substitution

The macro substitution is used to define a global symbol in the C program that represents a value. It is done with the help of **#define** preprocessor directive. Macros can be classified into two types they are,

(a) Simple macro definition
(b) Macro definition with arguments.

(a) Simple macro definition

The general form is

```
#define identifier symbol
```

where

#define → preprocessor directive
identifier → valid C name used in the source program

symbol → this value is substituted in the source program instead of the **identifier**. This may be any text.

If we include this line in the begining of our C program, the preprocessor replaces every occurance of the **identifier** in the source code by the **symbol**. There should be atleast one blank between the identifier and the symbol.

Example

(i)
```
#define VAL 200
main()
{
    int M [VAL];
    int N [VAL];
    - - - - - -
    - - - - - -
}
```

When we compile this program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definition and replaces all the occurance of the identifier VAL by 100. Only after this procedure is over the program is passed to the compiler.

Rules

(i) in the macro definition the symbol is assigned to the identifier as string.

Example

```
#define COUNT 100
```

This really assigns the string "100" to COUNT.

(ii) the symbol will not be substituted to the identifier with in the string.

Example

```
#define COUNT 100
- - - - - - -
- - - - - - -
printf("COUNT");
```

Here "COUNT" will not be substituted by 100.

(iii) in the macro definition expression can be included as symbols.

Example

```
#define A 3.14 * 5 * 5 * 3
```

(iv) in the macro definition literal text can be included as symbols

Example

```
#define TEST if (a>b)
#define xy GOODLUCK
```

(b) Macro definition with arguments

Macros can be defined with arguments, just like function. The general form is,

#define identifier $(a_1,a_3,....a_n)$ symbol

Where

#define → preprocessor directive
identifier → valid C name used in the source program.
$a_1,a_3,....a_n$ → formal macro arguments
symbol → this value is substituted to the formal arguments during processing.

Example

1. #define CUBE(x) (x*x*x)

   If this macro is called in the program as

   Volume = CUBE(10);

   then the preprocessor expands this statement as

   Volume = (10*10*10)

2. #define DECREMENT(y) if(y<0) Y=10

   If this macro is called in the program as

   DECREMENT(x);

   then the preprocessor expands this statement as

   if(x<0) x = 10

The difference between function arguments and this argument is : function argument is a value and macro argument is a string.

File inclusion directive

This directive is used to include the content of a file into the source code of the program. The general form is

#include"filename" [or] #include<filename>

Where,

#include → preprocessor directive

filename → name of the file to be included into the source code.

Whenever the #include directive is encountered, the preprocessor inserts the entire content of the filename into the source code of the program, Usually file inclusion directives are given in the begining of the program.

Example

1. #include<stdio.h>
2. #include"myfile.c"

Difference between "filename" and <filename>

If the filename is given within double quotation marks, the file is searched first in the current directory and then in the available standard directories.

If the filename is given with in < > the file is searched only in the standard directories.

Compiler control directives

This directive is used to direct the compiler to skip a part of source code. The preprocessor commands used are

(i) #ifdef . . . . . #endif

(ii) #ifdef . . . . . #else . . . . #endif

(iii) #if . . . . . #endif

(iv) #if . . . . . #else . . . . . # endif.

(i) #ifdef . . . . # endif

The general form is

```
#ifdef macroname
    Statement A;
#endif
```

If macroname has been defined in the included header file then statement A will be compiled.

Example

```
- - - - - - - - -
- - - - - - - - -
# ifdef MGP
   Statement 1,
   - - - - - - -
   - - - - - - -
   Statement n;
# endif
   - - - - - - -
   - - - - - - -
```

Here statements 1 to n will be compiled if the macro MGP is defined in the included file.

(ii) #ifdef . . . . #else . . . . #endif

The general form is

```
#ifdef macroname
   Statement A;
#else
   Statement B;
#endif
```

If macroname has been defined in the included header file statement A will be compiled else statement B.

Example

```
- - - - - - -
- - - - - - -
#ifdef MGP
      Statement 1;
      - - - - - - -
      Statement n;
#else
```



```
      Statement A;
      - - - - - - -
      Statement f;
#endif
```

Here statements 1 to n will be compiled if the macro MGP is defined in the header file else statements A to F will be compiled.

(iii) #if . . . . . #endif

The general form is

```
#if condition
   Statement 1;
   - - - - -
   Statement n;
#endif
```

If the result of the condition is true then the statements between #if and #endif will be compiled. If not these are skipped.

(iv) #if . . . . #else . . . . #endif

The general form is

```
#if condition
      Statement 1;
      - - - - - - -
      Statement n;
#else
      Statement A;
      - - - - - - -
      Statement F;
#endif
```

If the result of the condition is true then the statements 1 to n will be compiled else statements A to F will be compiled.

REVIEW QUESTIONS

PART - A

1. Give the general form to define a file.
2. Give the general form of open a file in write mode.
3. Give any four mode of opening a file.
4. Give the general form to close a file.
5. Write the syntax of fscanf ( ).
6. Give the error handling fuctions.
7. What is the use of feof ( ) function?
8. What is the use of ferror ( ) function?
9. Define random access file.
10. Define fseek ( ) function.
11. Define command line arguments.
12. What is preprocessor?
13. Give the general form of # define.
14. Define file inclusions directive.
15. Give the general form of # ifedf...#enddef.

PART - B

1. Explain the general form to open a file.
2. List the various I/O functions in files.
3. List any three errors in files.
4. Discuss fseek ( ).
5. Give the different preprocessor directives.

PART - C

1. Define a file. How a file in opened and closed?
2. Explain the various I/O operations on files.
3. How to handle errors during I/O operations.
4. Explain the use of command line arguments.
5. Explain about random access files.
6. How will you append items to a file.