STEP TOWARDS SUCCESS

AKASH'S

Guru Gobind Singh Indra Prastha University Series

SOLVED PAPERS

[PREVIOUS YEARS SOLVED QUESTION PAPERS]

[B.Tech]

FIRST/SECOND SEMESTER

Programming in 'C'

(ES-101/102)

Rs.81.00/-

AKASH BOOKS
NEW DELHI

SYLLABUS (From Academic Session 2021-22)

Programming in 'C' [ES-101/102]

Marking Scheme

(a) Teacher Continuous Evaluation: 25 marks

(b) Term End Theory Examination : 75 marks

UNIT I

Introduction to Prgramming: Computer system, components of a computer system, computing environments, computer languages, creating and running programs, Preprocessor, Compilation process, role of linker, idea of invocation and execution of a programme. Algorithms: Representation using flowcharts, pseudocode.

Introduction to C Language: History of C, basic structure of C programs, process of compiling and running a C program, C tokens, keywords, identifiers, constants, strings, special symbols, variables, data types, I/O statements. Interconversion of variables.

Operators and Expressions: Operators, arithmetic, relational and logical, assignment operators, increment and decrement operators, bitwise and conditional operators, special operators, operator precedence and associativity, evaluation of expressions, type conversions in expressions. **[8 Hrs.] [T2]**

UNIT II

Control Structures: Decision statements; if and switch statement; Loop control statements: while, for and do while loops, jump statements, break, continue, goto statements.

Arrays: Concepts, One dimensional array, declaration and initialization of one dimensional arrays, two dimensional arrays, initialization and accessing, multi dimensional arrays.

Functions: User defined and built-in Functions, storage classes, Parameter passing in functions, call by value, Passing arrays to functions: idea of call by reference, Recursion.

Strings: Arrays of characters, variable length character strings, inputting character strings, character library functions, string handling functions.

[8 Hrs.] [T2]

UNIT III

Pointers: Pointer basics, pointer arithmetic, pointers to pointers, generic pointers, array of pointers, functions returning pointers, Dynamic memory allocation. Pointers to functions, Pointers and Strings.

Structures and Unions: Structure definition, initialization, accessing structures, nested structures, arrays of structures, structures and functions, self referential structures, unions, typedef, enumerations.

File handling: Command line arguments, File modes, basic file operations read, write and append. Scope and life of variables, multi-file programming.

[8 Hrs.] [T2]

UNIT IV

C99 extensions. 'C' Standard Libraries: stdio.h, stdlib.h, assert.h, math.h, time.h, ctype.h, setjmp.h, string.h, stdarg.h, unistd.h **[3 Hrs.] [T1, R8]**

Basic Algorithms: Finding Factorial, Fibonacci series, Linear and Binary Searching, Basic Sorting Algorithms. Bubble sort, Insertion sort and Selection sort. Find the square root of a number, array order reversal, reversal of a string.

[7 Hrs.] [T1]

SYLLABUS

ACADEMIC SESSION : 2015-2016

INTRODUCTION TO PROGRAMMING (ETCS-108)

Instruction to Paper Setters: **Maximum Marks : 75**

1. *Question No. 1 should be compulsory and cover the entire syllabus. This question should have objective or short answer type questions. It should be of 25 marks.*
2. *Apart from Question No. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions. However, student may be asked to attempt only 1 question from each unit. Each question should be 12.5 marks*

UNIT I

Concept of algorithms, Flow Charts, Overview of the compiler (preferably GCC) , Assembler, linker and loader, Structure of a simple Hello World Program in C, Overview of compilation and execution process in an IDE (preferably Code Block)

[T1],[T2], [R4][R5][No. of hrs 8]

UNIT II

Programming using C: Preprocessor Directive, C primitive input output using get char and put char , simple I/O Function calls from library , data type in C including enumeration , arithmetic, relational and logical operations, conditional executing using if, else, switch and break .Concept of loops , for, while and do-while , Storage Classes: Auto, Register, Static and Extern.

[T1], [T2], [R7][No. of hrs 8]

UNIT III

Arrays (one and two dimensional), 2-d arrays used in matrix computation. Concept of Sub-programming, functions. Parameter transmission schemes i.e. call by value and call by reference, Pointers, relationship between array and pointer, Argument passing using pointers, Array of pointer, passing arrays as arguments.

[T2], [R1], [R7][No. of hrs 8]

UNIT IV

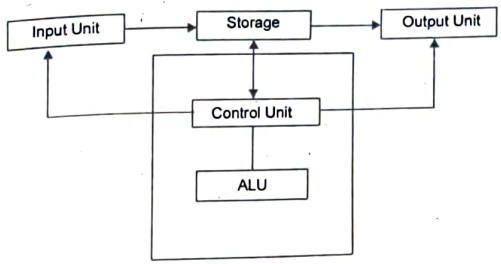
Structure and unions , Strings and C string library, File Handling in C Using File Pointers,fopen( ), fclose( ),Input and Output using file pointers, Character Input and Output with Files , String Input / Output Functions , Formatted Input / Output Functions,Block Input / Output Functions, Sequential Vs Random Access Files , Positioning the File Pointer.

[T1], [T2],[R2][R7][No. of hrs 8]

NEW TOPICS ADDED FROM ACADEMIC SESSION 2021-22
FIRST/SECOND SEMESTER
PROGRAMMING IN 'C' (ES -101/102)

UNIT I

Q. (a) Discuss the computer system with block diagram. **(2.5)**



Ans. A computer can process data, pictures, sound and graphics. They can solve highly complicated problems quickly and accurately.

1. Input Unit: Computers need to receive data and instruction to solve any problem. Therefore we need to input the data and instructions into the computers. The input unit consists of one or more input devices. Keyboard is the one of the most commonly used input device. Other commonly used input devices are the mouse, floppy disk drive, magnetic tape, etc. • Accept the data and instructions from the outside world. • Convert it to a form that the computer can understand. • Supply the converted data to the computer system for further processing.

2. Storage Unit: The storage unit of the computer holds data and instructions that are entered through the input unit, before they are processed. It preserves the intermediate and final results before these are sent to the output devices. It also saves the data for the later use. The various storage devices of a computer system are divided into two categories.

(a) Primary Storage: It is used to hold the program being currently executed in the computer, the data being received from the input unit, the intermediate and final results of the program. The primary memory is temporary in nature. The data is lost, when the computer is switched off. In order to store the data permanently, the data has to be transferred to the secondary memory. The cost of the primary storage is more compared to the secondary storage. Therefore most computers have limited primary storage capacity.

(b) Secondary Storage: It stores several programs, documents, data bases etc. The programs that you run on the computer are first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory. Some of the commonly used secondary memory devices are Hard disk, CD, etc.,

3. Output Unit: The output unit of a computer provides the information and results of a computation to outside world. Printers, Visual Display Unit (VDU) are the commonly used output devices. Other commonly used output devices are floppy disk drive, hard disk drive, and magnetic tape drive.

4. Arithmetic Logical Unit: All calculations are performed in the Arithmetic Logic Unit (ALU) of the computer. It also does comparison and takes decision. The ALU can perform basic operations such as addition, subtraction, multiplication, division, etc and does logic operations viz, >, <, =, 'etc. Whenever calculations are required, the control unit transfers the data from storage unit to ALU once the computations are done, the results are transferred to the storage unit by the control unit and then it is send to the output unit for displaying results.

5. Control Unit: It controls all other units in the computer. The control unit instructs the input unit, where to store the data after receiving it from the user. It controls the flow of data and instructions from the storage unit to ALU. It also controls the flow of results from the ALU to the storage unit. The control unit is generally referred as the central nervous system of the computer that control and synchronizes its working.

Central Processing Unit: The control unit and ALU of the computer are together known as the Central Processing Unit (CPU).The CPU is like brain performs the following functions:

- It performs all calculations.
- It takes all decisions.
- It controls all units of the computer.

Q. Differentiate between system and application software with suitable example **(2.5)**

Ans. Application software (an application) is a set of computer programs designed to permit the user to perform a group of coordinated functions, tasks, or activities. Application software cannot run on itself but is dependent on system software to execute. Example: MS-word, Excel, Spreadsheet, Games etc.

System software (systems software) is computer software designed to operate and control the computer hardware and to provide a platform for running application software. System software can be separated into two different categories, operating systems and utility software.

	Application Software	System Software
Definition	Application software is computer software designed to help the user to perform specific tasks.	System software is computer software designed to operate the computer hardware and to provide a platform for running application software.
Purpose	It is specific purpose software.	It is general-purpose software.
Classification	• Package Program, • Customized Program	• Time Sharing, • Resource Sharing, • Client Server • Batch Processing Operating System • Real time Operating System • Multi-processing Operating System • Multi-programming Operating System • Distributed Operating System
Environment	Application Software performs in a environment which created by System/ Operating System	System Software Create his own environment to run itself and run other application.
Execution Time	It executes as and when required.	It executes all the time in computer.
Essentiality	Application is not essential for a computer.	System software is essential for a computer

Q. What are high level languages and give five example? **(2.5)**

Ans. A machine independent programming language which lets the programmer concentrate on the logic of the problem to be solved rather than the intricacies of the machine architecture such as is requiredwith low-level assembly languages.

High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture.

High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

Programs written in a high-level language must be translated into machine language by a compiler or interpreter. The first high-level programming languages were designed in the 1950s. Example: Ada, Algol, BASIC, COBOL, C, C++, FORTRAN, LISP, Pascal, and Prolog.

Q. Define an operating system with an example

Ans. An operating system (OS) is a set of computer program that manages the hardware and software resources of a computer. Operating System is a software program that acts as an interface between the user and the computer. It is a software package which allows the computer to function.

The OS performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files.

Functions of Operting System:

1. Program creation
2. Program execution
3. Access to Input/Output devices
4. Controlled access to files
5. System access
6. Error detection and response
7. Interpreting the commands
8. Managing peripherals
9. Memory management
10. Processor management

11. Information management

12. Process communication

13. Networking

Examples of operating systems: MS-DOS, LINUX, SOLARIS, MAC OS, UNIX, WINDOWS

Q. Arrange the following in ascending order

Nibble, Kilobyte, Petabyte, Megabyte, Terabyte, Byte, Bit, Gigabyte

Ans. Arranged in Ascending order:

Bit

Nibble

Byte

Kilobyte

Megabyte

Gigabyte

Terabyte

Petabyte

Q. What are compilers and Interpreters used ?Is there any difference between Interpreter and Compiler? (5 + 5)

Ans. A high-level language is one which is understandable by us humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code. A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished my compilers and interpreters. Hence, a compiler or an interpreter is a program that converts program written in high-level language into machine code understood by the computer.

Difference between interpreter and compiler

	Compiler	Interpreter
1	Compiler Takes Entire program as input	Interpreter Takes Single instruction as input
2	Intermediate Object Code is Generated	No Intermediate Object Code is Generated
3	Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
4	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
5	Program need not be compiled every time	Every time higher level program is converted into lower level program
6	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
7	Example: C Compiler	Example : BASIC

Q. Briefly discuss the importance of cache memory

Ans. Cache Memory: The Cache Memory is the memory which is very nearest to the CPU , all the recent Instructions are Stored into the Cache Memory. The Cache Memory is attached for storing the input which is given by the user and which is necessary for the CPU to perform a Task. But the Capacity of the Cache Memory is too low in compare to Memory and Hard Disk.

Importance of Cache memory: The cache memory lies in the path between the processor and the memory. The cache memory therefore, has lesser access time than memory and is faster than the main memory. A cache memory have an access time of 100ns, while the main memory may have an access time of 700ns.

The cache memory is very expensive and hence is limited in capacity. Earlier cache memories were available separately but the microprocessors contain the cache memory on the chip itself.

The need for the cache memory is due to the mismatch between the speeds of the main memory and the CPU. The CPU clock as discussed earlier is very fast, whereas the main memory access time is comparatively slower. Hence, no matter how fast the processor is, the processing speed depends more on the speed of the main memory (the strength of a chain is the strength of its weakest link). It is because of this reason that a cache memory having access time closer to the processor speed is introduced.

The cache memory stores the program (or its part) currently being executed or which may be executed within a short period of time. The cache memory also stores temporary data that the CPU may frequently require for manipulation.

It acts as a high speed buffer between CPU and main memory and is used to temporary store very active data and action during processing since the cache memory is faster then main memory, the processing speed is increased by making the data and instructions needed in current processing available in cache. The cache memory is very expensive and hence is limited in capacity.

Q. Why we need the device drivers?

Ans.Computer is made up with various visible components (ie. cpu, memory stick, hard disk, peripherals). But they cannot work without the invisible operating system, which acts like a general commander and tells them what actions to take in order to accomplish tasks for us.. However, devices and operating system cannot communicate with each other directly. Drivers are therefore created to help them understand each other. Drivers work like translators and liaisons to convey operating system's meaning into different languages that different devices can comprehend, without which, devices cannot receive instructions from operating system and accordingly cannot function properly at all.

However, some devices, such as CPU, memory stick, monitor, keyboard and mouse, seem to be able to work directly without the need to install drivers for them. This is because these devices are must-have components for a computer to work in basic function. So codes were built in BIOS (software that is loaded immediately when a PC is powered on) at early stage so that they could be supported directly by operating system. Other than that, most devices (like graphic card, audio card, network card) require a specific driver being installed to work properly. For example When we connect a scanner, camera, printer to our computer, we're always asked to install a certain driver first.

Drivers are mostly released by official device manufacturers, who keep updating them not only to ensure the best compatibility with different operating system, but also for enhanced performance of devices. Nvidia, for example, a noted brand for graphics chips, averagely upgrade video card driver 2-3 times every month. By installing updated drivers, one will usually get better performance and new features out of our devices.

Input devices which are used in a computer "

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Track Ball
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader(OMR)

Keyboard: Keyboard is the most common and very popular input device which helps to input data to the computer. The layout of the keyboard is like that of traditional typewriter, although there are some additional keys provided for performing additional functions.

Keyboards are of two sizes 84 keys or 101/102 keys, but now keyboards with 104 keys or 108 keys are also available for Windows and Internet.

Mouse: Mouse is the most popular pointing device. It is a very famous cursor-control device having a small palm size box with a round ball at its base, which senses the movement of the mouse and sends corresponding signals to the CPU when the mouse buttons are pressed.

Generally, it has two buttons called the left and the right button and a wheel is present between the buttons. A mouse can be used to control the position of the cursor on the screen, but it cannot be used to enter text into the computer.

Output Devices

Monitors: Monitors, commonly called as Visual Display Unit (VDU), are the main output device of a computer. It forms images from tiny dots, called pixels that are arranged in a rectangular form. The sharpness of the image depends upon the number of pixels.

There are two kinds of viewing screen used for monitors.

- Cathode-Ray Tube (CRT)
- Flat-Panel Display

Printers: Printer is an output device, which is used to print information on paper.

There are two types of printers

- Impact Printers
- Non-Impact Printers

Impact Printers

Impact printers print the characters by striking them on the ribbon, which is then pressed on the paper.

Characteristics of Impact Printers are the following-

- Very low consumable costs
- Very noisy
- Useful for bulk printing due to low cost
- There is physical contact with the paper to produce an image

These printers are of two types

- Character printers
- Line printers

Character Printers

Character printers are the printers which print one character at a time.

These are further divided into two types:

- Dot Matrix Printer(DMP)
- Daisy Wheel

Dot Matrix Printer: In the market, one of the most popular printers is Dot Matrix Printer. These printers are popular because of their ease of printing and economical price. Each character printed is in the form of pattern of dots and head consists of a Matrix of Pins of size (5*7, 7*9, 9*7 or 9*9) which come out to form a character which is why it is called Dot Matrix Printer.

Advantages

- Inexpensive
- Widely Used
- Other language characters can be printed

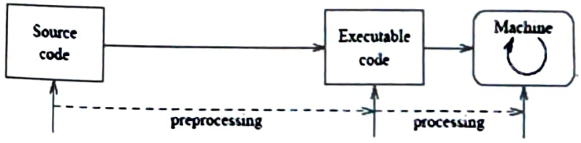
Disadvantages

- Slow Speed
- Poor Quality

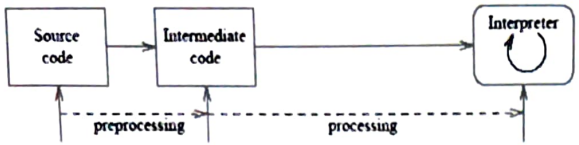
character set is embossed on the track. Different character sets available in the market are 48 character set, 64 and 96 characters set. One rotation of drum prints one line. Drum printers are fast in speed and can print 300 to 2000 lines per minute.

Q. Differentiate between Assembler, compiler, Interpreter. (2.5)

Ans. Assembler: Assembler converts source code written in assembly language into machine code and then that machine code is executed by a computer.



Compilation



Interpretation

Compiler: A compiler converts high-level language program code into machine language and then execute it. High-level languages are C and C++.

Interpreter: Interpreter converts source code into the intermediate form and then convert that intermediate code into machine language. The intermediate code looks same as assembler code.

During source code conversion variables and arithmetic operators like +, -, *, / are extracted from the code and then calculated.

In a compiler, all the code is converted at one time and the error report is generated for the whole program. In interpreter, the code is converted line by line and if an error occurs in the code then program execution stops.

In java language, compiler and interpreter work together to generate machine code.

Q. HLL and LLL (Highlevel language and low level language) (2.5)

Ans. High Level Language

- HLLs are programming languages that look like natural language text.
- They make programming easier and more abstract, i.e. the programmer does not need to come up with the detailed machine instructions.
- HLL programs are machine independent. They can run on different hardware platforms (i.e. different computers with different instruction sets)

Low Level Language : Low level computer languages are machine codes or close to it. Computer cannot understand instructions given in high level languages or in English. It can only understand and execute instructions given in the form of machine language i.e. language of 0 and 1. There are two types of low level languages:

Machine Language: It is the lowest and most elementary level of Programming language and was the first type of programming language to be Developed. Machine Language is basically the only language which computer Can understand. In fact, a manufacturer designs a computer to obey just one Language, its machine code, which is represented inside the computer by a String of binary digits(bits) 0 and 1.

Assembly Language: It was developed to overcome some of the many inconveniences of machine language. This is another low level but a very important language in which operation codes and operands are given in the form of alphanumeric symbols instead of 0's and 1's. These alphanumeric symbols will be known as mnemonic codes and can have maximum up to 5 letter combination e.g. ADD for addition, SUB for subtraction, START,LABEL etc. Because of this feature it is also known as 'Symbolic Programming Language'. This language is also very difficult and needs a lot of practice to master it because very small English support is given to this language. The language mainly helps in compiler orientations. The instructions of the Assembly language will also be converted to machine codes by language translator to be executed by the computer.

Q. Define cache memory. Why it is required. Discuss the principle on which cache memory works. (2.5)

Ans. Cache Memory : The Cache Memory is the Memory which is very nearest to the CPU , all the recent instructions are stored into the cache Memory. The Cache Memory is attached for storing the input which is given by the user and which is necessary for the CPU to perform a Task. But the capacity of the cache memory is too low in compare to memory and hard Disk.

The need for the cache memory is due to the mismatch between the speeds of the main memory and the CPU. The CPU clock as discussed earlier is very fast, whereas the main memory access time is comparatively slower. Hence, no matter how fast the processor is, the processing speed depends more on the speed of the main memory (the strength of a chain is the strength of its weakest link). It is because of this reason that a cache memory having access time closer to the processor speed is introduced.

Principle on which cache memory works is Locality of reference.

Q. What is Cache Memory? (2.5)

Ans. Cache memory, also called CPU memory, is high-speed static random access memory (SRAM) that a computer microprocessor can access more quickly than it can access regular random access memory (RAM). This memory is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU. The purpose of cache memory is to store program instructions and data that are used repeatedly in the operation of programs or information that the CPU is likely to need next. The computer processor can access this information quickly from the cache rather than having to get it from computer's main memory. Fast access to these instructions increases the overall speed of the program.

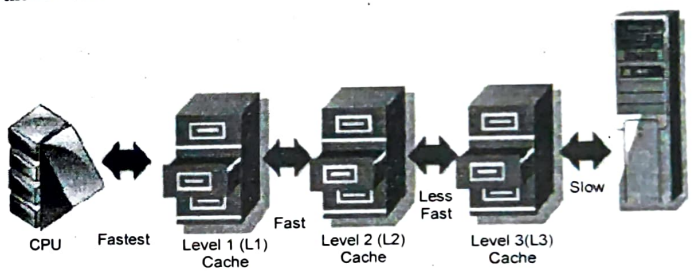


Diagram for Cache Memory

Q. What are Language Processors? Explain with examples?

Ans. A language processor is a software program designed or used to perform tasks, such as processing program code to machine code. Language processors are found in languages such as Fortran and COBOL.

There are two main types of language processors:

Interpreter: allows a computer to interpret, or understand, what a software program needs the computer to do, what tasks to perform.

Translator: takes a program's code and translates it into machine code, allowing the computer to read and understand what tasks the program needs to be done, in its own native code. An Assembler and a Compiler are examples of a translator. .

The difference between an interpreter and a translator is that an interpreter is telling the computer what to do by interpreting the program's code for the computer. A translator takes the program's code and converts it to machine code, allowing the computer to read that machine code itself. The interpreter tells the computer what to do and the translator lets the computer figure out what to do by itself

UNIT - IV

Q.1. Explain the Extensions to the C Language Family (C99).

Ans. GNU C provides several language features not found in ISO standard C. (The -pedantic option directs GCC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro __GNUC__, which is always defined under GCC.

These extensions are available in C and Objective-C. Most of them are also available in C++. Some features that are in ISO C99 but not C90 or C++ are also, as extensions, accepted by GCC in C90 mode and in C++.

• Statement Exprs:	Putting statements and declarations inside expressions.
• Local Labels:	Labels local to a block.
• Labels as Values:	Getting pointers to labels, and computed gotos.
• Nested Functions:	Nested function in GNU C.
• Nonlocal Gotos:	Nonlocal gotos.
• Constructing Calls:	Dispatching a call to another function.
• Typeof:	typeof: referring to the type of an expression.
• Conditionals:	Omitting the middle operand of a '?:' expression.
• __int128:	128-bit integers—__int128.
• Long Long:	Double-word integers—long long int.
• Complex:	Data types for complex numbers.
• Floating Types:	Additional Floating Types.
• Half-Precision:	Half-Precision Floating Point.
• Decimal Float:	Decimal Floating Types.
• Hex Floats:	Hexadecimal floating-point constants.
• Fixed-Point:	Fixed-Point Types.
• Named Address Spaces:	Named address spaces.
• Zero Length:	Zero-length arrays.
• Empty Structures:	Structures with no members.
• Variable Length:	Arrays whose length is computed at run time.
• Variadic Macros:	Macros with a variable number of arguments.
• Escaped Newlines:	Slightly looser rules for escaped newlines.
• Subscripting:	Any array can be subscripted, even if not an lvalue.
• Pointer Arith:	Arithmetic on void-pointers and function pointers.
• Variadic Pointer Args:	Pointer arguments to variadic functions.

• Pointers to Arrays:	Pointers to arrays with qualifiers work as expected.
• Initializers:	Non-constant initializers.
• Compound Literals:	Compound literals give structures, unions or arrays as values.
• Designated Inits:	Labeling elements of initializers.
• Case Ranges:	'case 1 ... 9' and such.
• Cast to Union:	Casting to union type from any member of the union.
• Mixed Labels and Declarations:	Mixing declarations, labels and code.
• Function Attributes:	Declaring that functions have no side effects, or that they can never return.
• Variable Attributes:	Specifying attributes of variables.
• Type Attributes:	Specifying attributes of types.
• Label Attributes:	Specifying attributes on labels.
• Enumerator Attributes:	Specifying attributes on enumerators.
• Statement Attributes:	Specifying attributes on statements.
• Attribute Syntax:	Formal syntax for attributes.
• Function Prototypes:	Prototype declarations and old-style definitions.
• C++ Comments:	C++ comments are recognized.
• Dollar Signs:	Dollar sign is allowed in identifiers.
• Character Escapes:	'\e' stands for the character ESC.
• Alignment:	Determining the alignment of a function, type or variable.
• Inline:	Defining inline functions (as fast as macros).
• Volatiles:	What constitutes an access to a volatile object.
• Using Assembly Language with C:	Instructions and extensions for interfacing C with assembler.
• Alternate Keywords:	__const__, __asm__, etc., for header files.
• Incomplete Enums:	enum foo;, with details to follow.
• Function Names:	Printable strings which are the name of the current function.
• Return Address:	Getting the return or frame address of a function.
• Vector Extensions:	Using vector instructions through built-in functions.
• Offsetof:	Special syntax for implementing offsetof.

• __sync Builtins:	Legacy built-in functions for atomic memory access.
• __atomic Builtins:	Atomic built-in functions with memory model.
• Integer Overflow Builtins:	Built-in functions to perform arithmetics and arithmetic overflow checking.
• x86 specific memory model extensions for transactional memory:	x86 memory models.
• Object Size Checking:	Built-in functions for limited buffer overflow checking.
• Other Builtins:	Other built-in functions.
• Target Builtins:	Built-in functions specific to particular targets.
• Target Format Checks:	Format checks specific to particular targets.
• Pragmas:	Pragmas accepted by GCC.
• Unnamed Fields:	Unnamed struct/union fields within structs/ unions.
• Thread-Local:	Per-thread variables.
• Binary constants:	Binary constants using the '0b' prefix.

Q.2. Explain the different types of C Standard Libraries.?

Ans. In C language, header files contain the set of predefined standard library functions. The "#include" preprocessing directive is used to include the header files with ".h" extension in the program.

Here is the table that displays some of the header files in C language,

<assert.h>	Conditionally compiled macro that compares its argument to zero
<complex.h> (C99)	Complex number arithmetic
<ctype.h>	Functions to determine the type contained in character data
<errno.h>	Macros reporting error conditions
<fenv.h> (C99)	Floating-point environment
<float.h>	Limits of floating-point types
<inttypes.h> (C99)	Format conversion of integer types
<iso646.h> (C95)	Alternative operator spellings
<limits.h>	Ranges of integer types
<locale.h>	Localization utilities
<math.h>	Common mathematics functions
<setjmp.h>	Nonlocal jumps
<signal.h>	Signal handling

<stdalign.h> (C11)	alignas and alignof convenience macros
<stdarg.h>	Variable arguments
<stdatomic.h> (C11)	Atomic operations
<stdbit.h> (C23)	Macros to work with the byte and bit representations of types
<stdbool.h> (C99)	Macros for boolean type
<stdckdint.h> (C23)	macros for performing checked integer arithmetic
<stddef.h>	Common macro definitions
<stdint.h> (C99)	Fixed-width integer types
<stdio.h>	Input/output
<stdlib.h>	General utilities: memory management, program utilities, string conversions, random numbers, algorithms
<stdnoreturn.h> (C11)	noreturn convenience macro
<string.h>	String handling
<tgmath.h> (C99)	Type-generic math (macros wrapping math.h and complex.h)
<threads.h> (C11)	Thread library
<time.h>	Time/date utilities
<uchar.h> (C11)	UTF-16 and UTF-32 character utilities
<wchar.h> (C95)	Extended multibyte and wide character utilities
<wctype.h> (C95)	Functions to determine the type contained in wide character data

Q.3. . Write an algorithm to find the factorial of a number?

Ans. Algorithm of Factorial Program in C

The algorithm of a C program to find factorial of a number is:

- Start program
- Ask the user to enter an integer to find the factorial
- Read the integer and assign it to a variable
- From the value of the integer up to 1, multiply each digit and update the final value
- The final value at the end of all the multiplication till 1 is the factorial
- End program

Q.4. Write an algorithm to print the Fibonacci series up to n?

Ans. Step 1: Start

Step 2: Declare variable a,b,c,n,i

Step 3: Initialize variable a=1, b=1, i=2

Step 4: Read n from user

Step 5: Print a and b

Step 6: Repeat until i<n

6.1 c=a+b

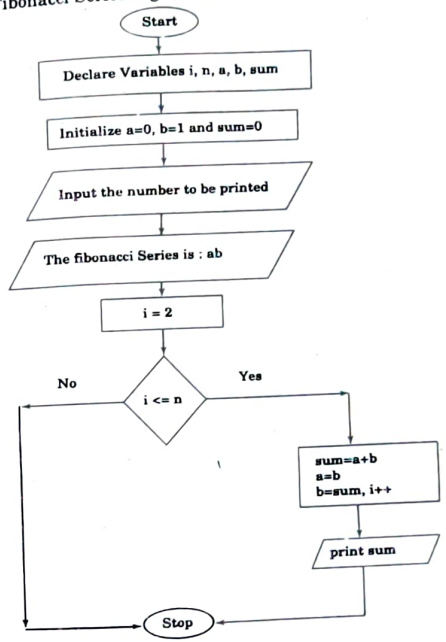
6.2 print c

6.3 a=b, b=c

6.4 i=i+1

Step 7: Stop

Flowchart for Fibonacci Series Algorithm:



Q.5. Write the algorithm for Linear Search & Binary Search?

Ans. Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Linear Search Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Binary Search : Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Binary Search Algorithm

1. Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search
2. Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
3. Step 2: repeat steps 3 and 4 while beg <=end
4. Step 3: set mid = (beg + end)/2
5. Step 4: if a[mid] = val
6. set pos = mid
7. print pos
8. go to step 6
9. else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit

Q.6.Write an algorithm for Bubble sort?

Ans. Algorithm: In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

1. begin BubbleSort(arr)
2. **for** all array elements
3. **if** arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end **if**
6. end **for**

7. **return** arr
8. end BubbleSort

Q.7. Write an algorithm for Insertion sort?

Ans. Algorithm:

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step 2 - Pick the next element, and store it separately in a **key**.

Step 3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Q.8. Write an algorithm for Selection sort?

Ans. Algorithm:

1. SELECTION SORT(arr, n)
2.
3. Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1
4. Step 2: CALL SMALLEST(arr, i, n, pos)
5. Step 3: SWAP arr[i] with arr[pos]
6. [END OF LOOP]
7. Step 4: EXIT
8.
9. SMALLEST (arr, i, n, pos)
10. Step 1: [INITIALIZE] SET SMALL = arr[i]
11. Step 2: [INITIALIZE] SET pos = i
12. Step 3: Repeat **for** j = i+1 to n
13. **if** (SMALL > arr[j])
14. SET SMALL = arr[j]
15. SET pos = j
16. [END OF if]
17. [END OF LOOP]
18. Step 4: RETURN pos

Q.9. Write an algorithm to find out the square root of a number?

Ans. Algorithm of program (Square root in C)

1. Start
2. Accept one number from user.
3. Use the function sqrt() to find square root.
4. Print the result.
5. End

Q.10. Write an algorithm for reversal of array?

Ans. Steps

1. Place the two pointers (let start and end) at the start and end of the array.

2.Swap arr[start] and arr[end]

3.Increment start and decrement end with 1

4.If start reached to the value length/2 or start ≥ end , then terminate otherwise repeat from step 2.

Pseudo Code

```
void rvereseArray(int[] arr) {
  start = 0
  end = arr.length - 1
  while (start < end) {
    // swap arr[start] and arr[end]
    int temp = arr[start]
    arr[start] = arr[end]
    arr[end] = temp
    start = start + 1
    end = end - 1
  }
}
```

Q.11. Write an algorithm for reverse a string?

Ans. Algorithm to reverse a string:

Step 1. Start

Step 2. Read the string from the user

Step 3. Calculate the length of the string

Step 4. Initialize rev = " " [empty string]

Step 5. Initialize i = length - 1

Step 6. Repeat until i>=0:

6.1: rev = rev + Character at position 'i' of the string

6.2: i = i – 1

Step 7. Print rev

Step 9. Stop

Q.12. What do you mean by the command line arguments?

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

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv**[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –


```
#include <stdio.h>

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

  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

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

```
$./a.out testing
The argument supplied is testing
```

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

```
$./a.out testing1 testing2
Too many arguments supplied.
```

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

```
$./a.out
One argument expected
```

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

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes –

UNIT - III

FIRST TERM EXAMINATION [APRIL-MAY–2016]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMING [ETCS–108]

Time : 1.30 Hrs. **M.M. : 30**

Note: *Attempt any three questions including Q.No. 1 which is compulsory.*

Q.1.(a)What are Flowchart and Algorithm.

Ans. A flowchart is a pictorial representation of an algorithm. It is a program planning tool for visually organizing a sequence of steps necessary to solve a problem using a computer. It uses boxes of different shapes to denote different types of instructions.

The main advantage of using flowchart is that a programmer need not pay attention to the details of the elements of the programming language. The process of drawing a flowchart for an algorithm is **flowcharting.**

Algorithm: Planning a program involves defining its logic. Thus, the algorithm refers to the logic of a program. It is a step by step description of how to arrive at a solution to a given problem. It is defined as a sequence of instructions that when executed in the specified sequence the desired results are obtained.

The characteristics of an algorithm are :

1. Each instruction should be precise & unambiguous.
2. Each instruction should be executed in a finite time.
3. One or more instructions should not be repeated infinitely. This ensures that the algorithm will ultimately terminate.
4. After executing the instructions, desired results are obtained.

Example: Write an algorithm to print the total number of students who passed in First Division. Fifty students appeared for test in Final Exam.

Algo :

Step 1 : Initialize Total_F_Div and Total_Mksheet to zero
Step 2 : Take the mark sheet of next student.
Step 3 : Check the division column of the marksheet to see if it is FIRST. If no, Goto Step 5.
Step 4 : Add 1 to Total_F_Div
Step 5 : Add 1 to Total_Mksheet
Step 6 : Is total_Mksheet = 50? If no, goto step 2
Step 7 : Print Total_F_Div
Step 8 : Stop.

Q.1.(b)What do you mean by Program?Write a small code for any program.

Ans. A computer program is a collection of instructions that performs a specific task when executed by a computer. A computer requires programs to function, and typically executes the program's instructions in a central processing unit.

A computer program is usually written by a computer programmer in a programming language. From the program in its human-readable form of source code, a compiler can derive machine code—a form consisting of instructions that the computer can directly execute. Alternatively, a computer program may be executed with the aid of an interpreter.

A part of a computer program that performs a well-defined task is known as an algorithm. A collection of computer programs, libraries and related data are referred to as software. Computer programs may be categorized along functional lines, such as application software or system software.

```
#include <stdio.h>
main()
{
  int i = 10;
  for(i=1;i<=10;i++)
    printf("Hello %d\n", i );
}
```

Q.1.(c)What is the difference between while and dowhile loop

Ans. Do while loop, execute the statements in the loop first before checks for the condition. At least one iteration takes places, even if the condition is false that means do while executes the body atleast once even if the condition is false.

```
#include <stdio.h>
main()
{
   int i = 10;
   do{
    printf("Hello %d\n", i );
    i = i -1;
  }while ( i > 0 );
}
```

This will produce following output:

```
Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
Hello 5
Hello 4
Hello 3
Hello 2
Hello 1
```

For loop is similar to while loop. If the condition is false at first iteration, the for loop doesn't execute. Secondly, the counter variable initialization, counter variable increment or decrement and condition testing syntax is written in single line. Example given below:

```
#include<stdio.h>
int main()
    {
    int i;
    for (i = 0; i < 10; i++)
    {
     printf ("Hello\n");
     printf ("World\n");
     }    return 0;}
```


Minimum number of times a while loop is executed is Zero(nil).If the condition is false at first iteration the while loop doesn't execute.

Write a program in C to print the multiplication table of a number using do while loop.

```
#include<stdio.h>
#include<conio.h>
 void main()
{
 clrscr();
 int i=1,n;
 printf("Enter a number whose table is to be printed");
scanf("%d",&n);
 do
{
J=n*i;
Printf("%d*%d=%d",n,i,j);
i++;
}while(i<=10);
getch();
}
```

Q.1.(d) What is the use of getchar() and putchar()?

Ans. The C library function **int getchar(void)** gets a character (an unsigned char) from stdin. This is equivalent to **getc** with stdin as its argument.

Declaration for getchar() function:

```
int getchar(void)
```

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getchar() function.

```
#include<stdio.h>
int main ()
{
char c;
  printf("Enter character: ");
  c = getchar();
  printf("Character entered: ");
  putchar(c);
return(0);
}
```

Output

Enter character: a

Character entered: a

The putchar macro is used to write a single character on the standard output stream (i.e., display). A call to this macro takes the following form: putchar (c), where c is an integer expression representing the character being written. Although this macro expects an argument of type int, we usually pass a character to it. The argument value (c) is converted to unsigned char and written to the standard output stream.

Q.1.(e) Explain continue ,break and switch statement

Ans. (e) (i) The break statement is used to terminate the execution of the nearest enclosing loop in which it appears.

- The break statement is used with for loop, while loop and do while loop.
- Syntax: break; [Just write break & terminate it with a semicolon].
- When compiler encounters a break statement, the control passes to the statement that follows the loop in which the break statement appears.

Example 1 :

```
while (...)
{
.......
if (condition)
break;
Transfer control out of the while loop)
}
.....
```

Example 2 :

```
for (...)
{
......
if (condition)
break;
..... Transfer control out of
| the For Loop
-----
```

Example 3 :

```
#include <stdio.h>
int main ()
{
init i = 0;
while (i<10)
{
if (i==5)
break;
Printf ("%d", i);
i = i+1;
}
return 0;
}
```


O/P: The code is meant to print first 10 numbers using a while loop, but it will actually print only numbers from 0 to 4. As soon as 'i' becomes equal to 5, the break statement is executed & the control jumps to the statement following the while loop.

***(ii)* Continue Statement:** When the compiler encounters a continue statement, then the rest of the statements in the loop are skipped and the control is unconditionally transferred to the loop-continuation portion of the nearest loop.

Syntax: continue;

Example 1 :

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

Example 2 :

```
#include <stdio.h>
int main ()
{
int i;
for (i=0; i<=10; i++)
{
if (i==5)
continue;
printf ("/t%d", i);
i=i+1;
}
return 0;
```

O/P : 0 1 2 3 4 5 6 7 8 9 10

The code given here is meant to print nos from 0 to 10. But as soon as i becomes equal to 5, the continue strnt. is encountered, so rest of the statements in the for loop are skipped & control passes to the expression that increments the value of i.

Q.2.Design an Algorithm and flowchart for finding largest of three numbers

Ans. Algorithm:

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

If a>c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b>c

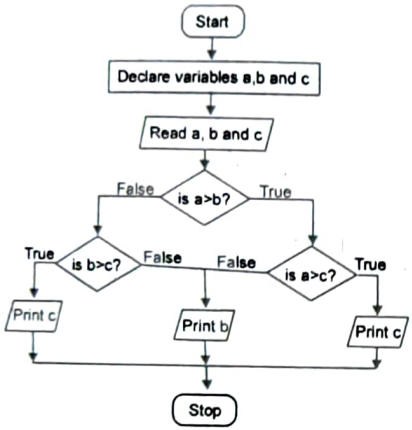
Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

Flowchart:



Q.2.(b)Explain different types of programming languages

Ans. Classification of Programming Languages are

1. Machine language
2. Assembly language
3. High level language (also known as Third Generation Language)
4. Very high level language

1. Machine Language: It is the only language that computer understands. All the commands & data values are expressed using 1's & 0's.

This is the lowest level of programming language.

Advantage of M/c L/g: The code can run very fast & efficiently since it is directly executed by CPU.

2. Assembly Language: These are symbolic programming language that use symbolic notation to represent machine language instruction. It uses symbolic codes also known as Mnemonic codes that are easy to remember abbreviations.

An assembly language statement consists of a label, an operation code & one or more operands.

Advantage: The code will again execute very fast.

Disadvantage: 1. Less portable, 2. Code not machine independent.

3. High Level Language: Programs were written in English like statements, making them more convenient to use & giving the programmer more time to address a clients problem.

Example: COBOL, FORTRAN, BASIC, C++, C.

But C & FORTRAN combine some of the flexibility of assembly language with the power of high level language.

Advantages: 1. It is easier to write and debug a program.

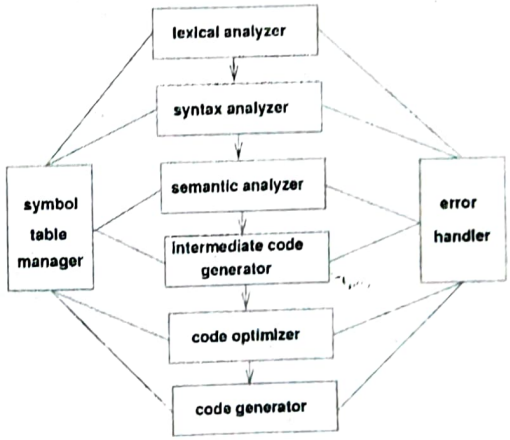
2. The programs written in HLL are portable between machines.

Q.3.(a) What is a compiler?What does it do?Explain structure of compiler

Ans A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an *editor*. The file that is created contains what are called the *source statements*. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called *object code* or sometimes an *object module* .

Structure of Compiler.



Q.3.(b)What are basic Assembler functions.Explain Machine dependent and Machine independent Assembler features

Ans. The basic assembler functions are:

- Convert mnemonic operation codes to their machine language equivalents.
- Convert symbolic operands to their equivalent machine addresses ü
- Build the machine instructions in the proper format.

- Convert the data constants to internal machine representations.
- Write the object program and the assembly listing.

Machine independent Assembler features

Literals

Symbol Defining Statements

Expressions

Program Blocks

Control Sections and Program Linking

Machine dependent Assembler features:

Instruction Format and Addressing Mode

Program Relocation

Q.4. What is a data type. Explain different data types in detail with examples.

Ans. C has the concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

1. Fundamental data Type

Integer: Integer data type is used to store numeric values without any decimal point e.g. 7,

–101, 107, etc.

A variable declared as 'int' must contain whole numbers e.g. age is always in number.

Syntax: intc

Int variable name;

E.g. int roll, marks, age;

Float: Float data type is used to store numeric values with decimal point. In other words, float data type is used to store-real values, e.g. 3.14, 7.67 etc. A variable declared as float must contain decimal values e.g. percentage, price, area etc. may contain real values.

Syntax: Float variable name;

E.g. float per, area;

Character: Char (Character) data type is used to store single character, within single quotes e.g. 'a', 'z', 'e' etc. A variable declared as 'char' can store only single character e.g. yes or No

Choice requires only 'y' or 'n' as an answer.

Syntax: Char variable name;

E.g. char chi='a', cha;

Void: Void data type is used to represent an empty value. It is used as a return type if a function does not return any value.

Type modifier: Type modifier is used to change the meaning of basic or fundamental data types so that they can be used in different situations. Various type modifiers are-signed, unsigned, short and long.

C language offers 3 different 'int' type modifiers-short, int, and long that represents three different integer sizes. C language also offers 3 different floating point typemodifiers-float, double, and long double. Two special 'char'type modifiers are-signed, unsigned.

Type Size (in bytes) Range

Int (signed/short) 2-32768 to 32767

In (unsigned) 2 0 to 65535

Long (signed) 4-2, 147,483,648 to 2, 147, 483, 648

Long (unsigned) 4 0 to 4,294,467,295

10 Second Semester Introduction to Programming, May-June 2013

Float 4 3.4*10-34 to 3.4*10-34

Double (long float) 8 1.7*10-308 to 1.7*10-308-1

Long double 10 3.4*10-4932 to 3.4*10-4932-1

Char 1 –128 to 127

Signed char 1 0 to 255

Unsigned char 1 –128 to 127

2. Derived Data Types

Data types that are derived from fundamental data types are called derived data types. Derived data types don't create a new data type; instead, they add some functionality to the basic data types. Two derived data type are - Array & Pointer.

Array: An array is a collection of variables of same type i.e., collection of homogeneous data referred by a common name. In memory, array elements are stored in a continuous location.

Syntax: [ ];

E.g. int a[10];

char ch [20];

According to the rules of C language, 1st element of array is stored at location a[0], 2nd at a[1] & so on.

Pointer: A pointer is a special variable that holds a memory address (location in memory) of another variable.

E.g. if we want to store the address of an 'int' data type variable into a pointer variable, it is done in the following manner:

int a, *b;

b=&a;

In the above case, variable 'b' stores the address of variable 'a'.

3. User Defined Data Type

User defined data type is used to create new data types. The new data types formed are fundamental data types. Different user defined data types are; struct, union, enum, typedef.

Struct: A struct is a user defined data type that stores multiple values of same or different data types under a single name. In memory, the entire structure variable is stored in sequence.

Syntax:

Struct< structure name>

{

var1;

Var2;

.....

.....

};

Union: A union is a user defined data type that stores multiple values of same or different data types under a single name. In memory, union variables are stored in a common memory location.

Syntax:

```
Union < tag name>
{
Var1;
Var2;
.....
.....
};
```

Enum: An enumeration is a data type similar to a struct or a union. Its members are constants that are written as variables, though they have signed integer values. These constant represent values that can be assigned to corresponding enumeration variables.

Syntax:

Enum {value1, value2, ..., value n};

E.g. :Enum colors {black, blue, cyan, green, red, yellow};

Color foreground, background;

Typedef: The 'typedef' allows the user to define new data-types that are equivalent to existing data types. Once a user defined data type has been established, then new variables, array, structures, etc. can be declared in terms of this new data type.

A new data type is defined as:

Typedef type new-type;

Type refers to an existing data type

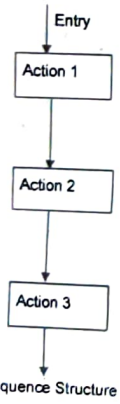
Q.4.(b)What are control structures.Explain the essential of repetition

Ans. A statement that is used to control the flow of execution in a program is called Contro. Structure. It combines instructions into logical unit.Logical unit has one entry and one exit point.

Types of Control Structure:

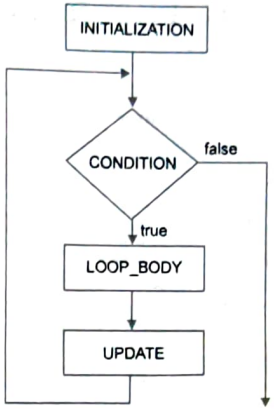
(a) Sequence (b) Selection (c) Repetition (d) Function Call

All the 3 control structures and its flow of execution is represented in the flow charts given below:



Sequence Structure

Repetition: In this structure the statements are executed more than one time. It is also known as Iteration or loop. For eg While, For, Do while etc.




```
#include<stdio.h>
#include<conio.h>
 void main()
{
 clrscr();
 int i=1,n;
 printf("Enter a number whose table is to be printed");
 scanf("%d",&n);
 do
{
J=n*i;
Printf("%d*%d=%d",n,i,j);
i++;
}while(i<=10);
getch();
```

END TERM EXAMINATION [MAY-JUNE–2016]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMING [ETCS–108]

Time : 1.30 Hrs. **M.M. : 75**

Note: *Attempt Q no. 1 which is compulsory and any five more questions including Q.No.1.*

Q.1.(a) Write Brief note on Structures Vs Union.

Ans.

	Structure	Union
Memory Allocation	Members of structure do not share memory. So A structure need separate memory space for all its members i.e. all the members have unique storage.	A union shares the memory space among its members so no need to allocate memory to all the members. Shared memory space is allocated i.e. equivalent to the size of member having largest memory.
Member Access	Members of structure can be accessed individually at any time.	At a time, only one member of union can be accessed.
Keyword	To define Structure, **'struct'** keyword is used.	To define Union, **'union'** keyword is used.
Initialization	**All** members of structure can be initialized.	**Only** the **first** member of Union can be initialized.
Size	Size of the structure is > to the sum of the each member's size.	Size of union is equivalent to the size of the member having **largest** size.
Syntax	struct struct_name { structure ele 1; structure ele 2; ——— ——— structure ele n; }struct_variable_name;	union union_name { union ele 1; union ele 2; ——— ——— union ele n; }union_variable_name;
Change in Value	Change in the value of one member can not affect the other in structure.	Change in the value of one member can affect the value of other member.

Q.1.(b) Write Brief note on Switch Statement Vs Multiple If.

Ans. 1. The main difference between switch and if-else is that the expression in if-else can result to a value of any data type, while the expression in switch should result to an integer value or to a value of datatype char.

2. The second difference is that the switch statement can check only for equality while the if-else statement can check any condition like equal to, less than, greater than or equal to etc.

3. In a practical sense, the switch statement may be thought of as an alternative to the use of nested if-else statements or else-if ladder. However, it can only replace those if-else statements that test for equality. If many such equality tests are to be carried out, then instead of using complex nesting of if-else or else-if ladder it is more convenient to use the switch statement.

Q.1.(c) Write Brief note on Break Vs Continue.

Ans.

S.No.	Break	Continue
1.	Break statement is used in switch and loops.	Continue statement is used in loops only.

2.	When break is encountered the switch or loop execution is immediately stopped.
	Example:
	#include<stdio.h>
	int main(){
	int i;
	for(i=0;i<5;++i){
	if(i==3)
	if(i==3)
	break;
	printf("%d",i);
	}
	return 0;}
	}
		When continue is encountered, the statements after it are skipped and the loop control jump to next iteration.
		Example:
		#include<stdio.h>
		int main(){
		int i;
		for(i=0;i<5;++i){
		continue;
		printf("%d",i);
		}
		return 0;}
		}

3.	**Output** 012	**Output:** 0 1 2 4

Q.1.(d) Write Brief note on Linker, Loader vs Compiler.

Ans. Compiler: It is a program which translates a high level language program into a machine language program. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program run time is more and occupies a larger part of the memory. It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes. If a compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler. On the other hand, if a compiler runs on a computer and produces the machine codes for other computer then it is known as a cross compiler.

Linker: In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not built in libraries, it also links the user defined functions to the user defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

Loader: Loader is a program that loads machine codes of a program into the system memory. In Computing, a **loader** is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

Q.1.(e) Write Brief note on Formal vs Actual Arguments.

Ans. Actual arguments: The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Formal arguments: The formal arguments are the parameters/arguments in a function declaration. The scope of formal arguments is local to the function definition in which they are used. Formal arguments belong to the called function. Formal arguments are a copy of the actual arguments. A change in formal arguments would not be reflected in the actual arguments.

Example:

```
#include <stdio.h>
void sum(int i, int j, int k);
/* calling function */
int main() {
int a = 5;
// actual arguments
sum(3, 2 * a, a);
return 0;
}
/* called function */
/* formal arguments*/
void sum(int i, int j, int k) {
int s;
s = i + j + k;
printf("sum is %d", s);
}
```

Here **3,2*a,a** are actual arguments and **i,j,k** are formal arguments.

Q.2.(a) Differentiate between auto , static, extern local and global variable with respect to their lifetime, scope and access rights.

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

- auto
- register
- static
- extern

The auto Storage Class

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

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

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

The Register Storage Class

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

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

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

The static Storage Class

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

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

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

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

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

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

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

The extern Storage Class

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

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

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

First File: main.c

```
#include <stdio.h>
int count ;
extern void write_extern();
main() {
  count = 5;
  write_extern();
}
```

Second File: support.c

```
#include <stdio.h>
extern int count;
void write_extern(void) {
  printf("count is %d\n", count);
}
```

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

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

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

```
count is 5
```

Q.2.(b) Differentiate between while and while-do loop with the help of an example.

Ans. Do-while loop is similar to while loop, however there is one basic difference between them – do-while runs at least one even if the test condition is false at first place. let's understand this with an example –

Using while loop:

```
main()
{
   int i=0
   while(i==1)
   {
        printf("while vs do-while");
   }
   printf("Out of loop");
}
```

Output:

Out of loop

Same example using do-while loop

```
main()
{
   int i=0
   do
   {
     printf("while vs do-while\n");
   }while(i==1);
   printf("Out of loop");
}
```

Output:

while vs do-while
Out of loop

Explanation: As I mentioned above do-while runs at least once even if the condition is false because compiler checks the condition after execution of its body.

Example:

```
int main()
{
   int j=0
   do
   {
      printf("Value of variable j is: %d", j);
      j++;
   }while (j<=8);
   return 0;
}
```

Output:

Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2

```
Value of variable j is: 3
Value of variable j is: 4
Value of variable j is: 5
Value of variable j is: 6
Value of variable j is: 7
Value of variable j is: 8
Value of variable j is: 9
```

Q.3.(a) Write a program in C which reads a string and after finding its length, program print back the string in reverse order. Do not use any built function

Ans:

```
#include <stdio.h>
int main()
{
  char s[1000], i;
  printf("Enter a string: ");
  scanf("%s", s);
  for(i = 0; s[i] != '\0'; ++i);
  printf("Length of string: %d", i);
  return 0;
}
```

Output

```
Enter a string: Programiz
Length of string: 9
```

C program to reverse string without using function

```
#include <stdio.h>
#include <string.h>
int main()
{
  char s[100], r[100];
  int n, c, d;
  printf("Input a string\n");
  gets(s);
  n = strlen(s);
  for (c = n - 1, d = 0; c >= 0; c--, d++)
    r[d] = s[c];
  r[d] = '\0';
  printf("%s\n", r);
  return 0;
}
```

Q.3.(b) What do you mean by enumerated data types? Are they different from strings? Explain their use?

Ans. An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword enum is used.


```
enum flag { const1, const2, ..., constN };
```

Here, name of the enumeration is flag.

And, const1, const2,...., constN are values of type flag.

By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).

```
// Changing default values of enum
enum suit {
  club = 0,
  diamonds = 10,
  hearts = 20,
  spades = 3,
};
```

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };
enum boolean check;
```

Here, a variable check of type enum boolean is created.

Here is another way to declare same check variable using different syntax.

```
enum boolean
{
  false, true
} check;
```

Use :

Enum variable takes only one value out of many possible values. Example to demonstrate it,

```
#include <stdio.h>
enum suit {
  club = 0,
  diamonds = 10,
  hearts = 20,
  spades = 3
} card;

int main()
{
    card = club;
printf("Size of enum variable = %d bytes", sizeof(card));
                                return 0;
}
```

Output

```
Size of enum variable = 4 bytes
```

It's because the size of an integer is 4 bytes.

Q.4. (a) Differentiate between Sequential/Random files in terms of the time taken for insertion, deletion and updating of records.

Ans: Relative data and information is stored collectively in file formats. A file is a sequence of records stored in binary format. A disk drive is formatted into several blocks that can store records. File records are mapped onto those disk blocks.

Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

File Operations

Operations on database files can be broadly classified into two categories "

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open**—A file can be opened in one of the two modes, **read mode** or **write mode**. In read mode, the operating system does not allow anyone to alter data. In other words, data is read only. Files opened in read mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.
- **Locate**—Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.
- **Read**—By default, when files are opened in read mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Write**—User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so.
- **Close**—This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
  - — removes all the locks (if in shared mode),
  - — saves the data (if altered) to the secondary storage media, and
  - — releases all the buffers and file handlers associated with the file.

Q.4.(b) Explain different classifiers for formatted outputs with example.

Ans. Printf Background

The printf function is not part of the C language, because there is no input or output defined in C language itself. The printf function is just a useful function from the standard library of functions that are accessible by C programs. The behavior of printf is defined in the ANSI standard. If the compiler that you're using conforms to this standard then all the features and properties should be available to you.

Format Specifiers

There are many format specifiers defined in C. Take a look at the following list:

%i or %d	int
%c	char
%f	float (see also the note below)
%s	string

Note: %f stands for float, but C language has also a thing called "default argument promotions".

Default argument promotions happen in variadic functions. Variadic functions are functions (e.g. printf) which take a variable number of arguments. When a variadic function is called, after lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions, each argument that is a part of the variable argument list undergoes additional conversions known as **default argument promotions**:

- float arguments are converted to double as in floating-point promotion
- bool, char, short, and unscoped enumerations are converted to int or wider integer types as in integer promotion

So for example, float parameters are converted to doubles, and char's are converted to int's. If you actually needed to pass, for example, a char instead of an int, the function would have to convert it back.

Let us take a look at an example of printf formatted output (that why you here, isn't it?):

```
#include<stdio.h>
main()
{
        int a,b;
        float c,d;
        a = 15;
        b = a / 2;
        printf("%d\n",b);
        printf("%3d\n",b);
        printf("%03d\n",b);
        c = 15.3;
        d = c / 3;
        printf("%3.2f\n",d);
        }
```

Output of the source above:

```
7
  7
007
5.10
```

As you can see in the first printf statement we print a decimal. In the second printf statement we print the same decimal, but we use a width (%3d) to say that we want three digits (positions) reserved for the output.

The result is that two "space characters" are placed before printing the character. In the third printf statement we say almost the same as the previous one. Print the output with a width of three digits, but fill the space with 0.

In the fourth printf statement we want to print a float. In this printf statement we want to print three position before the decimal point (called width) and two positions behind the decimal point (called precision).

The \n used in the printf statements is called an escape sequence. In this case it represents a newline character. After printing something to the screen you usually want to print something on the next line. If there is no \n then a next printf command will print the string on the same line. Commonly used escape sequences are:

- \n (newline)
- \t (tab)
- \v (vertical tab)
- \f (new page)
- \b (backspace)
- \r (carriage return)
- \n (newline)

Let's take another look at a printf formatted output in a more application like example:

```
#include<stdio.h>
main()
{
    int Fahrenheit;
    for (Fahrenheit = 0; Fahrenheit <= 300; Fahrenheit = Fahrenheit + 20)
        printf("%3d %06.3f\n", Fahrenheit, (5.0/9.0)*(Fahrenheit-32));
}
```

Output of the source above:

```
  0 -17.778
 20 -6.667
 40 04.444
 60 15.556
 80 26.667
100 37.778
120 48.889
140 60.000
160 71.111
180 82.222
200 93.333
220 104.444
240 115.556
260 126.667
280 137.778
300 148.889
```


As you can see we print the Fahrenheit temperature with a width of 3 positions. The Celsius temperature is printed with a width of 6 positions and a precision of 3 positions after the decimal point. Let's recap:

- %d (print as a decimal integer)
- %6d (print as a decimal integer with a width of at least 6 wide)
- %f (print as a floating point)
- %4f (print as a floating point with a width of at least 4 wide)
- %.4f (print as a floating point with a precision of four characters after the decimal point)
- %3.2f (print as a floating point at least 3 wide and a precision of 2)

Formatting other Types

Until now we only used integers and floats, but there are more types you can use. Take a look at the following example:

```
#include<stdio.h>
main()
{
    printf("The color: %s\n", "blue");
    printf("First number: %d\n", 12345);
    printf("Second number: %04d\n", 25);
    printf("Third number: %i\n", 1234);
    printf("Float number: %3.2f\n", 3.14159);
    printf("Hexadecimal: %x\n", 255);
    printf("Octal: %o\n", 255);
    printf("Unsigned value: %u\n", 150);
    printf("Just print the percentage sign %%\n", 10);
}
```

Output of the source example:

```
The color: blue
First number: 12345
Second number: 0025
Third number: 1234
Float number: 3.14
Hexadecimal: ff
Octal: 377
Unsigned value: 150
Just print the percentage sign %
```

Note: In the last printf statement only the percentage sign is printed.

The number 10 in this statement doesn't matter; it's not used in the output. So if you want to print a percentage number you would use something like this: printf("%2d%%\n", 10); (The output will be 10%)

Q.5. Give function name, syntax and corresponding header file to achieve the following:

(a) To open a file in read mode.

(b) To clear the input reader.

(c) To clear the output buffer.

(d) To read the buffered strings.

(e) To read the character Un-buffered.

(f) To read the whole line from input.

Ans. (a) The C library function **FILE *fopen(const char *filename, const char *mode)** opens the **filename** pointed to, by filename using the given **mode**.

Declaration

Following is the declaration for fopen() function.

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

Parameters

- **filename** " This is the C string containing the name of the file to be opened.
- **modo** " This is the C string containing a file access mode.

Ans. (b) getchar() returns the first character in the input buffer, and removes it from the input buffer. But other characters are still in the input buffer (\n in your example). You need to clear the input buffer before calling getchar() again:

```
void clearInputBuffer() // works only if the input buffer is not empty
{
  do
  {
    c = getchar();
  } while (c != '\n' && c != EOF);
}
```

Ans. (c) The C library function **int fflush(FILE *stream)** flushes the output buffer of a stream.

Declaration

Following is the declaration for fflush() function.

int fflush(FILE *stream)

Parameters

- **stream** " This is the pointer to a FILE object that specifies a buffered stream.

Return Value

This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set (i.e. feof).

Example

The following example shows the usage of fflush() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char buff[1024];
  memset( buff, '\0', sizeof( buff ));
  fprintf(stdout, "Going to set full buffering on\n");
  setvbuf(stdout, buff, _IOFBF, 1024);
  fprintf(stdout, "This is tutorialspoint.com\n");
```



```
  fprintf(stdout, "This output will go into buff\n");
  fflush( stdout );
  fprintf(stdout, "and this will appear when programm\n");
  fprintf(stdout, "will come after sleeping 5 seconds\n");
  sleep(5);
  return(0);
}
```

Let us compile and run the above program that will produce the following result. Here program keeps buffering into the output into **buff** until it faces first call to **fflush()**, after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before program comes out.

```
Going to set full buffering on
This is tutorialspoint.com
```

This output will go into buff and this will appear when programm will come after sleeping 5 seconds

Ans. (*d*) fgets (buffer, BUFSIZ, stdin);

or

scanf ("%128[^\n]%*c", buffer);

Here you can specify the buffer length 128 bytes as %128.. and also include all the blankspace within the string.

And then calculate the length and allocate new buffer with:

```
len = strlen (buffer);
string = malloc (sizeof (char) * len + 1);
strcpy (string, buffer);
•
•
•
free (string);
```

Ans. (*e*) *All* stdio.h functions for reading from a FILE may exhibit *either* "buffered" or "unbuffered" behavior, and *either* "echoing" or "non-echoing" behavior. What controls these things is not which function you use, but settings on the stream and/or its underlying file descriptor.

- The standard library function setvbuf can be used to enable or disable buffering of input (and output) by the C library. This has no effect on buffering by the operating system. There are three possible modes: "fully buffered" (read or write in substantial chunks); "line buffered" (buffer until a '\n' character is read or written, but not beyond that); and "unbuffered" (all reads and writes go to the OS immediately).

- The default buffering for new FILE objects (including stdin and friends) is implementation-defined. Unixy C libraries generally default all FILEs to fully buffered, with two exceptions. stderr defaults to unbuffered. For any other FILE, if setvbuf has not been used on it at the time of the first actual read or write, and isatty is true for the underlying file descriptor, then the FILE becomes line-buffered.

- Some C libraries provide extension functions, e.g. __flbf and friends on Linux and Solaris, for reading back *some* of the settings controlled by setvbuf. Keep in mind that,

(d) To read the buffered strings.

(e) To read the character Un-buffered.

(f) To read the whole line from input.

Ans. (a) The C library function **FILE *fopen(const char *filename, const char *mode)** opens the **filename** pointed to, by filename using the given **mode**.

Declaration

Following is the declaration for fopen() function.

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

Parameters

- **filename** − This is the C string containing the name of the file to be opened.
- **mode** − This is the C string containing a file access mode.

Ans. (b) getchar() returns the first character in the input buffer, and removes it from the input buffer. But other characters are still in the input buffer (\n in your example). You need to clear the input buffer before calling getchar() again:

```
void clearInputBuffer() // works only if the input buffer is not empty
{
  do
  {
    c = getchar();
  } while (c != '\n' && c != EOF);
}
```

Ans. (c) The C library function **int fflush(FILE *stream)** flushes the output buffer of a stream.

Declaration

Following is the declaration for fflush() function.

int fflush(FILE *stream)

Parameters

- **stream** − This is the pointer to a FILE object that specifies a buffered stream.

Return Value

This function returns a zero value on success. If an error occurs, EOF is returned and the error indicator is set (i.e. feof).

Example

The following example shows the usage of fflush() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char buff[1024];
  memset( buff, '\0', sizeof( buff ));
  fprintf(stdout, "Going to set full buffering on\n");
  setvbuf(stdout, buff, _IOFBF, 1024);
  fprintf(stdout, "This is tutorialspoint.com\n");
```



```
  fprintf(stdout, "This output will go into buff\n");
  fflush( stdout );
  fprintf(stdout, "and this will appear when programm\n");
  fprintf(stdout, "will come after sleeping 5 seconds\n");
  sleep(5);
  return(0);
}
```

Let us compile and run the above program that will produce the following result. Here program keeps buffering into the output into **buff** until it faces first call to **fflush()**, after which it again starts buffering the output and finally sleeps for 5 seconds. It sends remaining output to the STDOUT before program comes out.

```
Going to set full buffering on
This is tutorialspoint.com
```

This output will go into buff and this will appear when programm will come after sleeping 5 seconds

Ans. (*d*) fgets (buffer, BUFSIZ, stdin);

or

scanf ("%128[^\n]%*c", buffer);

Here you can specify the buffer length 128 bytes as %128.. and also include all the blankspace within the string.

And then calculate the length and allocate new buffer with:

```
len = strlen (buffer);
string = malloc (sizeof (char) * len + 1);
strcpy (string, buffer);
•
•
•
free (string);
```

Ans. (*e*) *All* stdio.h functions for reading from a FILE may exhibit *either* "buffered" or "unbuffered" behavior, and *either* "echoing" or "non-echoing" behavior. What controls these things is not which function you use, but settings on the stream and/or its underlying file descriptor.

• The standard library function setvbuf can be used to enable or disable buffering of input (and output) by the C library. This has no effect on buffering by the operating system. There are three possible modes: "fully buffered" (read or write in substantial chunks); "line buffered" (buffer until a '\n' character is read or written, but not beyond that); and "unbuffered" (all reads and writes go to the OS immediately).

• The default buffering for new FILE objects (including stdin and friends) is implementation-defined. Unixy C libraries generally default all FILEs to fully buffered, with two exceptions. stderr defaults to unbuffered. For any other FILE, if setvbuf has not been used on it at the time of the first actual read or write, and isatty is true for the underlying file descriptor, then the FILE becomes line-buffered.

• Some C libraries provide extension functions, e.g. __flbf and friends on Linux and Solaris, for reading back *some* of the settings controlled by setvbuf. Keep in mind that,

as described above, the buffering mode can change upon the first actual read or write, if it hasn't been explicitly set.

Ans. (*f*) Use fgets to read string from standard input

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

int main(void)
{
  char str[80];
  int i;

  printf("Enter a string: ");
  fgets(str, 10, stdin);

  /* remove newline, if present */
  i = strlen(str)-1;
  if( str[ i ] == '\n')
     str[i] = '\0';

  printf("This is your string: %s", str);

  return 0;
}
```

Q.6.(a) Write a program in C which reads an array of size n, find how many numbers in the array are even, odd and count them and print the information.

Ans. This C Program prints the number of odd & even numbers in an array.

Here is source code of the C Program to print the number of odd & even numbers in an array. The C program is successfully compiled. The program output is also shown below.

```
1. /*
2.  * C Program to Print the Number of Odd & Even Numbers in an Array
3.  */
4. #include <stdio.h>
5.
6. void main()
7. {
8.    int array[100], i, num;
9.
10.       printf("Enter the size of an array \n");
11.       scanf("%d", &num);
12.       printf("Enter the elements of the array \n");
13.       for (i = 0; i < num; i++)
```



```
14.       {
15.          scanf("%d", &array[i]);
16.       }
17.       printf("Even numbers in the array are - ");
18.       for (i = 0; i < num; i++)
19.       {
20.          if (array[i] % 2 == 0)
21.          {
22.             printf("%d \t", array[i]);
23.          }
24.       }
25.       printf("\n Odd numbers in the array are -");
26.       for (i = 0; i < num; i++)
27.       {
28.          if (array[i] % 2 != 0)
29.          {
30.             printf("%d \t", array[i]);
31.          }
32.       }
33.    }
```

```
$ cc pgm70.c
$ a.out
Enter the size of an array
6
Enter the elements of the array
12
19
45
69
98
23
Even numbers in the array are - 12   98
Odd numbers in the array are  - 19   45   69   23
```

Q.6.(b) Write a program in C which reads a array of integers. Each element of the array is replaced by its square. Print back the array.

Ans.

```
#include<stdio.h>
#include<conio.h>
#define MAX_ROWS 3
#define MAX_COLS 4
void print_square(int[ ]);
void main(void) {
int row;
```

```
int num[MAX_ROWS][MAX_COLS] = { { 0, 1, 2, 3 },
{ 4, 5, 6, 7 },
{ 8, 9, 10, 11 } };
for (row = 0; row < MAX_ROWS; row++)
print_square(num[row]); }
void print_square(int x[]) {
int col;
for (col = 0; col < MAX_COLS; col++)
 printf("%d\t", x[col] * x[col]);
printf("\n");}
```

Output :

0	1	4	9
16	25	36	49
64	81	100	121

Q.7.(a) Differentiate between call by value and call by reference parameter passing mechanism with the help of an example.

Ans. Call By Value : In Call By Value method, the arguments are passed to a function using the **value of the variable**. There is a second copy of the variable is formed while passing values from one function to another.

Therefore, the original values or the **original variables do not change**. The call by value and call by reference approach are mostly used while passing values to functions in C programming.

The original variables are called **Actual Parameters** whereas the new values that are passed to the function are called as **Formal Parameters**.

In case of call by value approach, the copy of the values is passed to the function. The actual parameters and formal parameters are thereby created in different memory locations. Therefore, the original value is not modified in call by value method in C programming.

The values passed to the **User Defined Function** are created on **Stack** memory location. Therefore, the changes made within the user defined function is not visible in the **main()** method or any other method for that matter.

The disadvantage with pass by value method is that there are two copies created for the same variable which is not **memory efficient**. However, the advantage of call by value approach is that this method doesn't change the original variables thereby preserving data. The following program will demonstrate how to write a C code to swap two variables using call by value method.

In the following C program to swap variables with Call By Value approach, the variables **var1** and **var2** are passed to the **swap()** method by using their values and not the addresses.

When the user defined function **swap()** receives the values of var1 and var2, two local copies are created for that variables in the function. These local values **val1** and **val2** are manipulated by the program itself and there is no effect on the actual parameters var1 and var2 that were passed from the main() function.

C Program To Swap Variables using Call by Value Method

```
1  include<stdio.h>
2  int swap(int a, int b);
3   int main()
4   {
5  int main()
6  {
7  int num1, num2;
8  printf("\nEnter The First Number:\t");
9   scanf("%d", &num1);
10   printf("\nEnter The Second Number:\t");
11  scapf("%", &num2);
12  swap(num1, num2);
13  printf("/nOld values\n");
14   printf("\nFirst Number = %d\nSecond Number = %d\n", num1, num2);
15  printf("\n");
16   return 0;
17  }
18  int swap(int a, int b)
19  {
20  int temp;
21   temp = a;
22   a = b;
23   b = temp;
24  printf("\nNew Values\n");
25   printf("\nFirst Number = %d\nSecond Number = %d\n", a, b);
26  }
```

Call By Reference Method

In Call By Reference method, the arguments are passed to another function using the **address of the variables**. This address is also known as **Reference Pointers**. There is no second copy of the variable while passing values from one function to another.

Since, the original variables' address is passed, the modifications done to the variable in the User Defined will **change the original variable** as well. Therefore, the Original Values will change.

The original variables are called **Actual Parameters** whereas the new values that are created within the user defined function are called **Formal Parameters**. In case of call by reference approach, the address of the original values are passed to the function.

The actual parameters and formal parameters are thereby created in the **same memory location**. All the modification is done on actual parameters. Therefore, the original value is modified in Pass By Reference method in C programming.

The changes made within the user defined function are visible in the **main()** method and any other method for that matter. We have demonstrated below a C code for call by reference in C using Pointers.

The Advantage of using pass by reference approach is that it does not create duplicate data for holding only one value. Therefore, this helps to **save memory space**. The following program will demonstrate how to write a C program to swap two variables using pass by reference method.

In the C program to swap variables with passing by reference approach, the variables **var1** and **var2** are passed to the **swap()** method by using their addresses and not the values.

When the User Defined Function **swap()** receives the addresses of var1 and var2, the pointers ***ptr1** and ***ptr2** holds the values of the var1 and var2 since the addresses are passed to them. Therefore, no local copies are created. The original variables var1 and var2 will be directly modified by prt1 and ptr2.

C Program To Swap Variables using Call By Reference Method

```
#include<stdio.h>
int swap(int *ptr1, int *pt2);
int main()
{
  int var1, var2;
   printf("Enter The First Number:\t");
   scanf("%d", &var1);
    printf("Enter The Second Number:\t");
   scanf("%d", &var2);
  swap(&var1, &var2);
  printf("The Swapped Values are:\n");
   printf("First Number = %d\nSecond Number = %d\n", var1, var2);
   return 0;
    }
   int swap(int *ptr1, int *ptr2)
  {
    int temp;
    temp = *ptr1;
  *ptr1 = *ptr2;
    *ptr2 = temp;}
  }
```

Q.7.(b) In C, when arrays array are passed to the function, which mechanism do they follow? Is there any relation between pointers and array name? Explain.

Ans: In C programming, a single array element or an entire array can be passed to a function.

This can be done for both one-dimensional array and a multi-dimensional array.

Passing One-dimensional Array In Function

Single element of an array can be passed in similar manner as passing variable to a function.

C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int age)
{
  printf("%d", age);
}
int main()
{
  int ageArray[] = { 2, 3, 4 };
  display(ageArray[2]); //Passing array element ageArray[2] only.
  return 0;
}
```

Output

4

While passing arrays as arguments to the function, only the name of the array is passed (,i.e, starting address of memory area is passed as argument).

C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
float average(float age[]);
int main()
{
  float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
  avg = average(age); /* Only name of array is passed as argument. */
  printf("Average age=%.2f", avg);
  return 0;
}
float average(float age[])
{
  int i;
  float avg, sum = 0.0;
  for (i = 0; i < 6; ++i) {
    sum += age[i];
  }
  avg = (sum / 6);
  return avg;
}
```

Output

Average age=27.08

Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example: Pass two-dimensional arrays to a function

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
```

```
int main()
{
  int num[2][2], i, j;
  printf("Enter 4 numbers:\n");
  for (i = 0; i < 2; ++i)
    for (j = 0; j < 2; ++j)
      scanf("%d", &num[i][j]);
  // passing multi-dimensional array to displayNumbers function
  displayNumbers(num);
  return 0;
}
void displayNumbers(int num[2][2])
{
  // Instead of the above line,
  // void displayNumbers(int num[][2]) is also valid
  int i, j;
  printf("Displaying:\n");
  for (i = 0; i < 2; ++i)
    for (j = 0; j < 2; ++j)
      printf("%d\n", num[i][j]);
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Relation between Pointer and Array Name:

A pointer variable is used to store the address of another variable while an array is used to group related data items of same data type.

The name of the array is a pointer to the first element of the array. That means it holds the address of the very first element of the array.

For eg, if we declare an array - int arr[10];

Then - arr = &arr[0]

FIRST TERM EXAMINATION [FEB. 2017]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING
[ETCS–108]

Time : 1½ Hrs. **M.M. : 30**

Note: *Attempt any three questions including Q. No. 1 which is compulsory.*

Q.1. (a) What is the output of the following programs? Justify your answer. (2)

(a) #define square(x) x*x
void main() { int a=5; printf("%d", square(a+1)); }

Ans. 11

[5 + 1 × 5 + 1

5 + 5 + 1 [because multiplication has priority over Addition]

= 11]

Q.1. (b) enum colors {BLACK, BLUE=2,GREEN}; (2)
void main() { printf("%d -> %d -> %d", BLACK, BLUE, GREEN); }

Ans. 0, 2, 3

(In enum, values are 0,1,2. So, Black will be given value 0. Value of blue has been assigned to 2. Thus Blue value = 2 will get printed and value of Green will be incremented by 1 and 3 will be get printed for green.

Q.1. (c) void main() {int x=0; switch(x) { (2)
case 1: printf("one");
case 0 : printf("two");
default: printf("wrong");
case2:printf("three");}}

Ans. two wrong three

[As value of x = 0, therefore case 1 case 1 won't get executed. Control will go directly to case 0 & there is no break statement. Therefore, all cases after that will get executed.]

Q.1. (d) void main (){ int *x,y,z;* (2)
z = x++ + ++y - x- - + - -y ; printf("%d %d %d",x,y,z); }

Ans. 1, 1, 2

[Assume x = 1, y = 1, z = 1

$z = 1 + 2 - 2 + 1$

$\therefore z = 1$

$y = 1$

$x = 1$ (After decrementing)]

Q.1. (e) void main(){ int x=4,y=2,z=l; (2)
x=10 + ((z== -1 || z<=0) ? y/z : ++y); printf("%d %d %d",x,y,z); }

Ans. 3, 3, 1

[x = 10 + (If this then (?) y/z else ++y)

$\therefore x = 10 +$ (else part)

$= 10 + 3$

$= 13$

$z = 1$

$y = 3$ (because incremented)

Q.2. (a)What is the difference between entry control loop and exit control loop? Explain giving an example program from each category. (6)

Ans. Loops are the technique to repeat set of statements until given condition is true. C programming language has three types of loops - **1) while loop, 2) do while loop** and **3) for loop**.

These loops controlled either at entry level or at exit level hence loops can be controlled in two ways:

1. Entry Controlled Loop
2. Exit Controlled Loop

Entry Controlled Loop: Loop, **where test condition is checked before entering the loop body**, known as **Entry Controlled Loop**.

Example: while loop, for loop

Exit Controlled Loop: Loop, **where test condition is checked after executing the loop body**, known as **Exit Controlled Loop**.

Example: do- while loop

Using while loop

```
int count=100;
while(count<50)
printf("%d",count++);
```

Example: using for loop

```
int count;
for(count=100; count<50; count++)
printf("%d",count);
```

In both code snippets **value of count is 100** and condition is **count<50**, which will check first, hence there is no output.

Example: using do while loop

```
int count=100;
do
{
printf("%d",count++);}
while(count<50);
```

In this code snippet **value of count is 100** and test condition is **count<50** which is false yet loop body will be executed first then condition will be checked after that. Hence output of this program will be 100.

Entry Controlled Loop	Exit Controlled Loop
1. Test condition is checked first, condition is checked.	Loop body will be executed first, and then and then loop body will be executed.
2. If Test condition is false, loop body will not be executed.	If Test condition is false, loop body will be executed once.
3. For loop and while loop are the examples of Entry Controlled Loop.	do while loop is the example of Exit controlled loop.
4. Entry Controlled Loops are used when checking of test condition is mandatory before executing loop body.	Exit Controlled Loop is used when checking of test condition is mandatory after executing the loop body.

Q.2. (b)WAP to evaluate a^b without using library functions. (4)

Ans.

```
#include <stdio.h>
int main()
{
 int base, exponent;
long long power = 1;
int i;
/* Input base and exponent from user */
 printf("Enter base: ");
 scanf("%d", &base);
 printf("Enter exponent: ");
 scanf("%d", &exponent);
/* Multiply base, exponent times*/
for(i=1; i<=exponent; i++)
{
  power = power * base;
}
printf("%d ^ %d = %lld", base, exponent, power);
return 0;
}
```

Q.3. (a)Briefly explain different forms of decision control structures. Draw a flowchart to find largest of three numbers. (6)

Ans. C has three major decision making instructions—the if statement, the if-else statement, and the switch statement.

1. The if Statement

C uses the keyword if to implement the decision control instruction. The general form of if statement looks like this:

```
//for single statement
if(condition)
  statement;
//for multiple statement
if(condition)
{
  block of statement;
}
```

The more general form is as follows:

```
//for single statement
if(expression)
  statement;
//for multiple statement
if(expression)
{
  block of statement;
}
```

Here the expression can be any valid expression including a relational expression. We can even use arithmetic expressions in the if statement.

2. The if-else Statement

The if statement by itself will execute a single statement, or a group of statements, when the expression following if evaluates to true. It does nothing when the expression

evaluates to false. We can execute one group of statements if the expression evaluates to true and another group of statements if the expression evaluates to false. This is what is the purpose of the else statement that is demonstrated as

```
if (expression)
{
  block of statement;
}
else
  statement;
```

3. Nested if-else

If we write an entire if-else construct within either the body of the If statement or the body of an else statement. This is called *"nesting"* of ifs. This is demonstrated as -

```
if (expression1)
  statement;
else
{
  if (expression2)
    statement;
  else
  {
    block of statement;
  }
}
```

4. The if-else Ladder/else-if Clause

The else-if is the most general way of writing a multi-way decision.

```
if(expression1)
  statement;
else if(expression2)
  statement;
else if(expression3)
  statement;
else if(expression4)
{
  block of statement;
}
else
  statement;
```

The *expressions* are evaluated in order; if an expression is true, the *"statement"* or *"block of statement"* associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces. The last else part handles the *"none of the above"* or *default case* where none of the other conditions is satisfied.

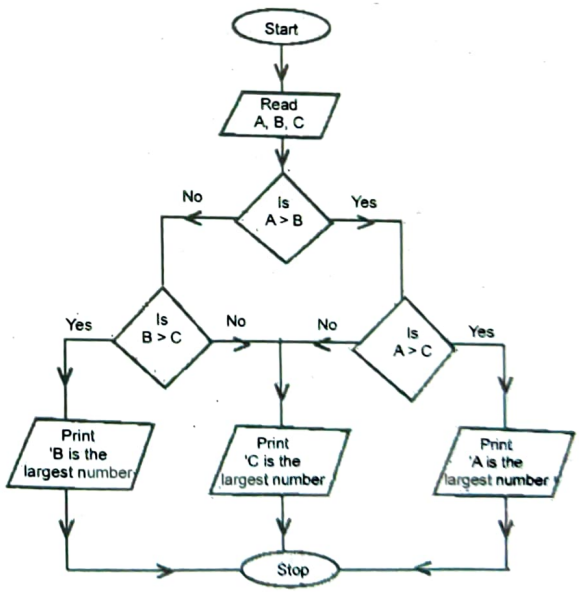
5. Switch Statements or Control Statements

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. The switch statement that allows us to make a decision from the number of choices is called a switch, or switch-case-default, since these three keywords go together to make up the switch statement.

```
switch (expression)
{
case constant-expression:
  statement1;
  statement2;
  break;
case constant-expression:
  statement;
  break;
...
default:
  statement;
}
```


- In switch..case command, each case acts like a simple label. A label determines a point in program which execution must continue from there. Switch statement will choose one of case sections or labels from where the execution of the program will continue. The program will continue execution until it reaches break command.
- break statements have vital rule in switch structure. If you remove these statements, program execution will continue to next case sections and all remaining case sections until the end of switch block will be executed (while most of the time we just want one case section to be run).
- default section will be executed if none of the case sections match switch comparison.

Flowchart to find largest of 3 numbers



Q.3. (b)WAP to determine whether input year is a leap year or not using logical operators. **(4)**

Ans.

```
#include<stdio.h>
main()
{
int year;
printf("Enter the year to check for a leap year:");
scanf ("%d", &year);
if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
{
printf("The entered year is a leap year.");
}
else
{
printf("The entered year is not a leap year.");
}
```

Q.4. Explain any two with suitable example.

(a) Formatted Console I/O and Unformatted Console I/O **(5)**

Ans. THE CONSOLE I/O FUNCTIONS

The screen and keyboard together are called a console. The console functions can be further divided into the following :

- **Formatted console I/O functions :** These allow the input read from the keyboard or the output displayed on the VDU to be formatted as per the requirements. e.g. scanf(), printf()
- **Unformatted console I/O functions :** These do not allow formatted input or output. e.g. getch(), getche(), getchar(), gets, putch(), putchar(), puts()

FORMATTED CONSOLE I/O FUNCTIONS

These functions allow us to supply the input in a fixed format . scanf() and printf() are its example.

The syntax of printf() looks like :

printf ("format string", list of variables) ;

where the format string can contain :

- Characters that are simply printed as they are
- Conversion specifications that begin with a % sign
- Escape sequences that begin with a \ sign

The syntax of scanf() looks like :

scanf ("format string", list of addresses of variables) ;

For example :

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

UNFORMATTED CONSOLE I/O FUNCTIONS

In this, we have functions that deal with either a single character or a string of characters

FUNCTIONS THAT DEAL WITH A SINGLE CHARACTER

A weakness of *scanf()* is that we have to hit ENTER before the function can digest what we have typed. **getch()** and **getche()** are two functions that overcome this weakness. These functions return the character that has been most recentiy typed. The 'e' in getche( ) function means it echoes (displays) the character that you typed to the screen. As against this getch( ) just returns the character that you typed without echoing it on the screen.

getchar() works similarly and echoes the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed.

```
main()
{
        char ch;
    ch = getchar();
}
```

putch() and **putchar()** print a character on the screen. Their working is the same. This is shown in the example:

```
main( )
{
        char c = 'C';
      putch ( ch );
    putchar ( ch );
      putch ( 'A' );
    putchar ( 'M' );
}
```

The output is CCAM

FUNCTIONS THAT DEAL WITH A STRING OF CHARACTERS

Another weakness of *scanf()* is that it reads a string until it encounters a space. i.e. you cannot have strings having spaces input using this function. This is overcome by using **gets()** function. It is terminated when an Enter key is hit. Thus, spaces and tabs are perfectly acceptable as part of the input string.

The **puts()** function outputs a string to the screen.

The following program shows the use of these two functions:

```
main( )
{
    char footballer[40] ;
    puts ( "Enter name" ) ;
    gets ( footballer ) ; /* sends base address of array */
    puts ( "Happy footballing!" ) ;
    puts ( footballer ) ;
}
```

Q.4. (b)Beak and continue with example. **(5)**

Ans. The break statement is used to terminate the current enclosing loop or conditional statements in which it appears. We use the break statement to come out of switch statements.

The continue statement is used to alter the sequence of execution. Instead of coming out of the loop like the break statement did, the continue statement stops the current iteration and simply returns control back to the top of the loop.

Example of Break Statement:

```
for(i=1 ;i<=1 0;i++)
{
if(i%5==0)'
break;
else
Printf("%d",&i);
}
O/P:1 2 3 4
```

Example of Continue Statement:

```
for(i=1 ;i<=1 ; i ++)
{
if(i%5==0)
 continue;
else
printf("%d",&i);
}
O/P:1 2 3 4 6 7 8 9
```

Q.4. (c) Storage classes in C (5)

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program.

There are four storage classes which can be used in a C Program

Auto

register

static

extern

Auto Storage Class is the default storage class for all local variables.

```
{
int Month;
//or
auto int Month; //Both declarations are same
}
```

Auto can only be used within functions, i.e. local variables.

Register Storage Class

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

Example: register int month;

Static Storage Class is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
{
printf("%d\n", Road);
}
```

Static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

Static can also be defined within a function. If this is done, the variable is initalised at run time but is not reinitalized when the function is called. Thus, inside a function static variable retains its value during vairous calls.

External storage class: External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

Example:

```
main()
{
extern int x;
x=10;
printf("%d",x);
}
int x; // Global variable x;
```

Q.4. (d)linker vs loader. (5)

Ans. A linker is a program used with a compiler or assembler to provide links to the libraries needed for an executable program. A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

A loader reads the executable code into memory, does some address translation and tries to run the program resulting in a running program or an error message (or both).

END TERM EXAMINATION (MAY-JUNE 2017)
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING
[ETCS–108]

Time : 3 Hrs. **M.M. : 75**

Note: *Attempt any five questions including Q. No. 1 which is compulsory.*

Q.1. Differentiate between the following:

Q.1.(a) Assembler and Linker **(5)**

Ans. Assembler: An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer. An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter. An assembler primarily serves as the bridge between symbolically coded instructions written in assembly language and the computer processor, memory and other computational components. An assembler works by assembling and converting the source code of assembly language into object code or an object file that constitutes a stream of zeros and ones of machine code, which are directly executable by the processor.

Linker - A **linker** is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program. The program as a whole refers to these separately-compiled object files using symbols. The linker combines these separate files into a single, unified program; resolving the symbolic references as it goes along.

Dynamic linking is a similar process, available on many operating systems, which postpones the resolution of some symbols until the program is executed. When the program is run, these dynamic link libraries are loaded as well. Dynamic linking does not require a linker.

Q.1. (b) Internal and External Documentation. **(5)**

Ans. Internal documentation is written in a program as comments. External documentation is written in a place where people who need to use the software can read about how to use the software. External documentation can be broken down into *library documentation*, which describes tools that a programmer can use, and user documentation, which is intended for users of an application.

The line between internal and library documentation is not clearcut because the trend is to write library documentation inside a program as comments, relying on software that extracts the documentation and puts it into a form suitable for people who only want to use the library. For example, the extensive Java library documentation is created by software called *javadoc* that reads Java programs, including comments, and writes documentation.

Q.1. (c)Break and Continue. **(5)**

Ans. Refer Q.4. (c) First Term Examination Feb. 2017

Q.1. (d)Malloc() and Calloc() **(5)**

Ans. When calloc is used to allocate a block of memory, the allocated region is initialized to zeroes. In contrast, malloc does not touch the contents of the allocated

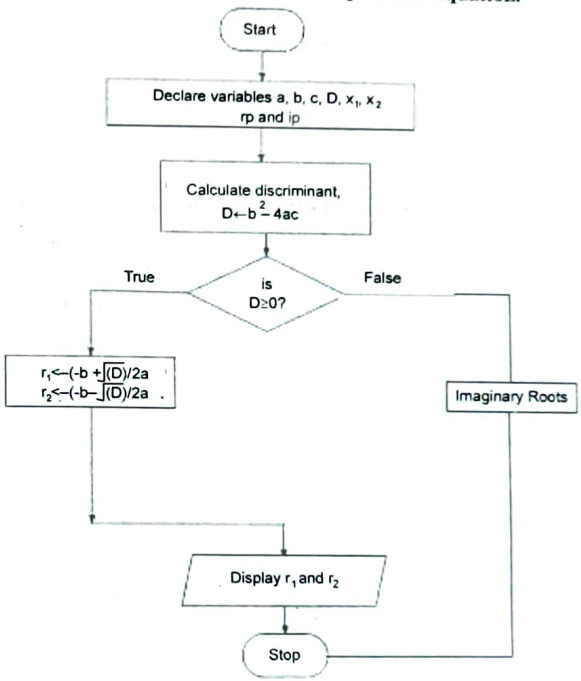
block of memory, which means it contains garbage values. This could potentially be a security risk because the contents of memory are unpredictable and programming errors may result in a leak of these contents.

malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory.

Q.1. (e)while and do-while **(4)**

Ans. Refer Q.1. (c) First Term Examination April-May 2016

Q.2. (a) Draw a flowchart to find roots of quadratic equation. **(5)**



Q.2. (b)WAP to print the pattern **(7.5)**

Ans.

```
#include <stdio.h>
int main()
{
   int i, j, rows;
   printf("Enter number of rows:" );
   scanf("%d",&rows);
   for(i=1; i<=rows; ++i)
   {
      for(j=1; j<=i; ++j)
      {
         printf("* ");
      }
      printf("\n");
   }
   return 0;
}
```

Q.3. (a)Explain difference between static and constant with example. (4)

Ans. A variable declared as const can only be initialised once. The program should not alternate the value of a variable declared as const. A variable declared as const should never appear to the left of an assignment operator throughout the program. The reader may refer the C literature for appreciating the use const variable. Now consider a static variable. Static variables are permanent variables within their on function. Unlike global variables they are not known outside their function or file, but they maintain their values between function calls.

For example:

```
#include<stdio.h>
void display (void)
{
int j=10;
++j;
printf("J=%d \n",j);
}
main(void)
{
int i;
for (i=0;i<10;++i)
display();
}
```

Output:

J=11
J=11
J=11

J=11
J=11
J=11
J=11
J=11
J=11

In the above program consider the variable 'j'. 'j' is declared as a an ordinary local variable. It's life is only though out the function display(). After the execution of the function variable 'j' is automatically destroyed. Thus when the function is called again variable 'j' is again defined and is initialised to value 10. Thus in all cases the the value of j displayed is 11.

Consider the code below

```
#include<stdio.h>
void display (void)
{
static int j=10;
++j;
printf("J=%d \n",j);
}
main(void)
{
int i;
for (i=0;i<10;++i)
display();
}
```

Output:

J=11
J=12
J=13
J=14
J=15
J=16
J=17
J=18
J=19
J=20

Here the difference is that the variable 'j' is declared as static. In this case the variable 'j' will not be destroyed when the function execution is over and in addition, it also retains its last value. Thus the declaration and initialisation of 'j' will occur only once when the function is called for the first time. Thus each time the function display() is called value of 'j' is incremented.

Q.3. (b) What are enumerated data types. Explain with example. **(4)**

Ans. An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword enum is used.

```
enum flag { const1, const2, ..., constN };
```

Here, name of the enumeration is flag

And, const1, const2, ..., constN are values of type flag.

By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary). For example:

```
// Changing default values of enum
enum suit {
  club = 0,
  diamonds = 10,
  hearts = 20,
  spades = 3,
};
```

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
Enum Boolean { false, true };
enum Boolean check;
```

Here, a variable check of type enum oolean is created.

Q.3. (c) Explain Preprocessor directives with examples. **(4.5)**

Ans. The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if

Preprocessors Examples:

```
#define MAX_ARRAYJLENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

Q.4. (a) Explain dangling pointer and NULL pointer with example. **(3)**

Ans. A dangling pointer is one that has a value (not NULL) which refers to some memory which is not valid for the type of object you expect. For example if you set a pointer to an object then overwrote that memory with something else unrelated or freed the memory if it was dynamically allocated. Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally, dangling references and wild references are references that do not resolve to a valid destination, and include such phenomena as link rot on the internet.

Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. The system may reallocate the previously freed memory, and if the program then dereferences the (now) dangling pointer, *unpredictable behavior may result*, as the memory may now contain completely different data. If the program writes to memory referenced by a dangling pointer, a silent corruption of unrelated data may result, leading to subtle bugs that can be extremely difficult to find. If the memory has been reallocated to another process, then attempting to dereference the dangling pointer can cause segmentation faults (UNIX, Linux) or general protection faults(Windows). If the program has sufficient privileges to allow it to overwrite the bookkeeping data used by the kernel's memory allocator, the corruption can cause system instabilities. In object-oriented languages with garbage collection, dangling references are prevented by only destroying objects that are unreachable, meaning they do not have any incoming pointers; this is ensured either by tracing or reference counting. However, a finalizer may create new references to an object, requiring object resurrection to prevent a dangling reference.

Eg. Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

```
int main()
{
  int *p;  /* wild pointer */
    /* Some unknown memory location is being
corrupted.
       This should never be done. */
       *p = 12;
}
```

The **null pointer** is used to denote the end of a memory search or processing event. In computer **programming**, a **null pointer** is a **pointer** that does not point to any object or function. A nil **pointer** is a false value. For example, 1 > 2 is a nil statement

Q.4. (b) Explain argument passing using pointers. **(3)**

Ans. When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value. This is because, pointer stores the location of the memory, and not the value.

```
/* C Program to swap two numbers using pointers and function. */
#include <stdio.h>
void swap(int *n1, int *n2);
int main(){
  int num1 = 5, num2 = 10;
```

```
// address of num1 and num2 is passed to the swap function
swap( &num1, &num2);
printf("Number1 = %d\n", num1);
printf("Number2 = %d", num2);
return 0;
}
void swap(int * n1, int * n2)
{
// pointer n1 and n2 points to the address of num1 and num2 respectively
int temp;
temp = *n1;
*n1 = *n2;
*n2 = temp;
}
```

Output

Number1 = 10

Number2 = 5

Q.4. (c) Explain various storage classes in C. Difference between primitive and non-primitive data types. **(4 + 2.5 = 6.5)**

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program. There are following storage classes which can be used in a C Program:

auto

register

static

extern

Auto Storage Class is the default storage class for all local variables.

```
{
int Month;
//or
auto int Month; //Both declarations are same
}
```

Auto can only be used within functions, i.e. local variables.

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

Example: register int month;

Static Storage Class is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

Static variables can be 'seen' within all functions in this source file. Static can also be defined within a function. If this is done, the variable is initialised at run time but is not reinitialized when the function is called. Thus, inside a function static variable retains its value during various calls.

External storage class: External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

```
Example:
main()
{
extern int x;
x=10;
printf("%d",x);
}
int x; // Global variable x;
```

Primitive and non-primitive data types

A data type is a classification of data, which can store a specific type of information. Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain. The term "data type" and "primitive data type" are often used interchangeably. Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types.

- For eg. Character (character , char);
- Integer (integer , int , short , long , byte) with a variety of precisions;
- Floating-point number (float , double , real , double precision);
- Fixed-point number (fixed) with a variety of precisions and a programmer-selected scale.

Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "lastname" and define it as a string data type. The variable will then store data as a string of characters.

- Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data. The data types that are derived from primary data types are known as **non-primitive data types.**
- The non-primitive datatypes are used to **store group of values.**
- The non-primitive data types are :
  - Arrays
  - String
  - Structures

Q.5. (a) Differentiate between Structure and Union with example. **(2.5)**

Ans. Structures are used to group together different types of variables under the same name. For example you could create a structure "student": which is made up of a string (that is used to hold the name of the student) and an integer (that is used to hold the student. For example:

```
Struct student
{
    char name[20];
```

```
    int id;
} s1,s2;
```

Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct. For example:**

```
Union car
{
    char name[50];
     int price;
}c1,c2;
```

There is difference in memory allocation between union and structure as suggested in above example. **The amount of memory required to store a structure variables is the sum of memory size of all members.**

Q.5. (b)Explain call by value and call by reference in C. **(4)**

Ans. Call by Value: If data is passed by value, the data is copied from the variable used in for example mainO to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include<stdio.h>
#include<conio.h>
int swap(int, int);                 // Declaration of function
main( )
{
int a = 10, b = 20;                 // call by value
swap(a.b);                          // a and b are actual parameters
printf ( "\na = %d b = %d", a, b ;
getch();
}
int swap( int x, int y)             // x and y are formal parameters
{
int t ;
t = x;
x = y;
y = t;
printf ( "\nx = %d y = %d", x, y ) ;
}
```

Call by Reference: In call by reference mechanism, instead of passing values to the function being called, references/pointers to the original variables are passed.

```
#include <stdio.h>
void swap (int*, int*);
int main()
{
int x, y;
```



```
printf("Enter the value of x and y\n");
scanf("%d%d",&x,&y);
printf("Before Swapping\nx = %d\ny = %d\n", x, y);
swap(&x, &y);
printf("After Swapping\nx = %d\ny = %d\n", x, y);
return 0;
}
void swap(int *a, int *b)
{
int temp;
temp = *b;
*b =*a;
*a = temp;
}
```

Q.5. (c)Write a program to find the factorial of a number through recursion. **(6)**

Ans.

```
#include <stdio.h>
int factorial(int);
int main()
{
      int num;
  int result;
  printf("Enter a number to find it's Factorial:");
  scanf("%d", &num);
  if (num < 0)
  {
    printf("Factorial of negative number not possible\n");
  }
  else
  {
    result = factorial(num);
    printf("The Factorial of %d is %d.\n", num, result);
  }
  return 0;
}
int factorial(int num)
    {
 if (num == 0 || num == 1)
       {
    return 1;
  }
  else
  {
```

```
    return(num * factorial(num - 1));
  }
}
```

Enter a number to **find** it's Factorial: 6

The Factorial of 6 is 720.

Q.6. (a)Write a program that copies the file called "abc.txt" to another file called "new txt". **(6)**

Ans.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main() {
  FILE *fp1, *fp2;
  char ch;
  clrscr();
  fp1 = fopen("abc.txt", "r");
  fp2 = fopen("new.txt", "w");
  while (1) {
    ch = fgetc(fp1);
    if (ch == EOF)
      break;
    else
      putc(ch, fp2);
  }
  printf("File copied Successfully!");
  fclose(fp1);
  fclose(fp2);
}
```

Q.6. (b)Explain formatted I/O functions for files. **(2.5)**

Ans. Formatted input reads characters from the input file and converts them to internal form. Formatted I/O can be either "Free" format or "Explicit" format.Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.) Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

However, formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text. Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

Q.6. (c) Differentiate between sequential and random access files. **(4)**

Ans. Refer Q4 (a) End Term Examination May-June 2016

Q.7. (a) Write a program to check whether an input string is palindrome or not. **(6.5)**

Ans.

```
#include <stdio.h>
 #include <string.h>
int main()
{
chartext[100];
int begin, middle, end, length = 0;
gets(text);
while (text [length] != '\0')
length++;
end = length - 1;
middle = length/2;
for( begin = 0 ; begin < middle ; begin++ )
{
if ( text[begin] != text[end] )
{
      printf("Not a palindrome.\n");
      break;
   }
     end — ;
}  if( begin == middle )
     printf("Palindrome.\n");
 return 0;
}
```

Q.7. (b) Write a program to count the number of vowels and words in the string "Introduction to Programming". **(6)**

Ans.

```
#include<stdio.h>
 int main(){
          char words[200];
          int vowels=0, letters=0, word=0, digits=0, spaces=0;
          int flag=0, i;
          clrscr();
          printf("Enter a line of Text :\n");
          gets(words);
          for(i=0;words[i]!='\0';++i)
          {
           if(words[i]== 'a' || words[i]== e' || words[i]=='i' || words[i]=='o' || words[i]=='u' || words[i]=='A' || words[i]=='E' || words[i]=='I' || words[i]=='O' || words[i]=='U')
          ++vowels;
          else
          if(words[i]==' '){
```

```
++spaces;
flag=0;}
if (words[i] != ' '&& flag==0){
++word;
flag=1;
}
}
letters += vowels;
printf("\n\n Number of words: %d", word);
printf("\n Number of Vowels: %d", vowels);
getch();
return 0;
}
```

Q.8. (a) Write a program to multiply two matrices. (4)

Ans.

```
# include <stdio.h>
int raain()
{
int m, n, p, q, c, d, k, sum = 0;
intfirst[10][10], second[10][10], multiply[10][10];
printf("Enter the number of rows and columns of first matrix\n");
scanf("%d%d", &m, &n);
printf("Enter the elements of first matrix\n");
for ( c = 0 ; c < m ; c++ )
for ( d = 0 ; d < n ; d++ )
scanf("%d", &first[c][d]);
printf("Enter the number of rows and columns of second matrix\n");
scanf("%d%d", &p, &q);
if ( n != p )
printf("Matrices with entered orders can't be multiplied with each other.\n");
else
{
printf("Enter the elements of second matrix\n");
for ( c = 0 ; c < p ; c++ )
for ( d = 0 ; d < q ; d++ )
scanf("%d", &second[c][d]);
for (c = 0 ; c < m ; c++ )
{
for ( d = 0 ; d < q ; d++ )
{
```



```
for ( k = 0 ; k < p ; k++)
{
sum = sum + first[c][k]*second[k][d];
}
multiply[c][d] = sum;
sum = 0;
}
}
printf("Product of entered matrices:\n");
for (c = 0 ; c < m ; c++ )
{
for ( d = 0 ; d < q ; d++ )
printf("%d\t", multiply[c][d]);
printf("\n");
}
}
return 0;
}
```

Q.8. (b) Explain ternary operator with example. (3.5)

Ans. The conditional operator in C is also known as ternary operator. It is called ternary operator because it takes three arguments. The conditional operator evaluates an expression returning a value if that expression is true and different one if the expression is evaluated as false.

Syntax for conditional operator in C:

```
1 condition ? result 1 : result 2;
```

If the condition is true, result1 is returned else result2 is returned.

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a = 10, b = 11;
int c;
c = (a < b)? a : b;
printf("%d", c);
}
```

Q.8. (c) Explain bitwise operators available in C. (5)

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

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12 *i.e.*, 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A # B) = 61 *i.e.*, 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49 *i.e.*, 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61 *i.e.*, 1100 0011 in 2's Complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	(A << 2) = 240 *i.e.*, 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	(A >> 2) = 15 *i.e.*, 0000 1111

Example:

```
#include<stdio.h>

main(){

unsignedint a =60;  /* 60 = 0011 1100 */
unsignedint b =13;  /* 13 = 0000 1101 */
int c =0;

 c = a & b;/* 12 = 0000 1100 */
printf("Line 1 - Value of c is %d\n", c );

 c = a | b;/* 61 = 0011 1101 */
printf("Line 2 - Value of c is %d\n", c );

 c = a ^ b;/* 49 = 0011 0001 */
printf("Line 3 - Value of c is %d\n", c );
}
```

Output:

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49

FIRST TERM EXAMINATION [FEB. 2018]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING [ETCS–108]

Time : 1.30 hrs. **M.M. : 30**

Note: Question 1 is compulsory. Attempt any two more questions from the rest.

Q.1. What is the output of the following programs? Justify your answer.

Q. 1. (a)
```
void main()
{ int n;
for(n = 7; n!=0; n--)
printf("n = %d",n--); }
```
(2)

Ans. infinite loop

Q. 1. (b)
```
enum {true, false};
void main()
{int i = 1;
do{ printf("%d\n", i); i++;
        if (i < 15) continue;
} while (true); }
```
(2)

Ans. 1

Q. 1. (c)
```
void main()
{ int d, a = 1 , b = 2;
d = a++ + ++b;
printf("%d %d %d", d, a, b);
}
```
(2)

Ans. d = 4, a = 2,b = 3

Q. 1. (d)
```
void main()
{ int x = 2, y = 0, k,z;
z = y = (0, 1, -1); k = x && y;
printf("%d %d %d %d \n", x,y,z,k);
}
```
(2)

Ans. $x = 2$, $y = -1$, $z = -1$, k=1

Q. 1. (e)
```
void main() {int x=0;
switch(x) { case 1: printf("one");
case 0 : printf("two");
default: printf("wrong");
case 2 : printf("three"); }}
```
(2)

Ans. two,wrong, three

Q. 2. (a) What is the difference between while loop and do-while loop? Explain giving an example program from each category. **(6)**

Ans. Do-while loop is similar to while loop, however there is one basic difference between them - do-while runs at least one even if the test condition is false at first place, let's understand this with an example -

Using while loop:

```
main()
{
  int i=0
  while(i==1)
  {
    printf("while vs do-while");
  }
    printf("Out of loop"); }
```

Output:

Out of loop

Same example using do-while loop

```
main()
{
   int i=0
   do
  {
   printf("while vs do-while\n");
  }while(i==1);
  printf("Out of loop");
  }
```

Output:

while vs do-while

Out of loop

Explanation: As I mentioned above do-while runs at least once even if the condition is false because compiler checks the condition after execution of its body.

Example:

```
/ int main()
{
   int j=0
   do
  {
    pintf("Value of variable j is: %d", j);
   j++;
  }while (j<=8);
 return 0;
}
```

Output:

Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
Value of variable j is: 4
Value of variable j is: 5

Value of variable j is: 6
Value of variable j is: 7
Value of variable j is: 8
Value of variable j is: 9

Q. 2. (b) Write a program to find factorial of a number. (4)

Ans.

```
#include <stdio.h>
int main()
{
  int n, i;
  unsigned long long factorial = 1;
  printf("Enter an integer:");
  scanf("%d",&n);
  // show error if the user enters a negative integer
  if (n < 0)
    printf("Error! Factorial of a negative number doesn't exist.");
  else
  {
    for(i=1; i<=n; ++i)
    {
      factorial *= i;          // factorial = factorial*i;
    }
    printf("Factorial of %d = %llu", n, factorial);
  }
  return 0;
}
```

Q. 3. (a) Briefly explain the different category of operators supported in 'C' with an example of each. [4]

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

- **Arithmetic Operators**

+	addition or unary plus	
–	subtraction or unary minus	
*	multiplication	
/	division	
%	remainder after division (modulo division)	

- **Relational Operators**

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

<	Checks if the value of left operand is less than the value of right operand if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A>=B)is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

• Logical Operators

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

• Bitwise Operators

&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 00111101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 00110001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A<< 2 will give 240 which is 11110000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 00001111

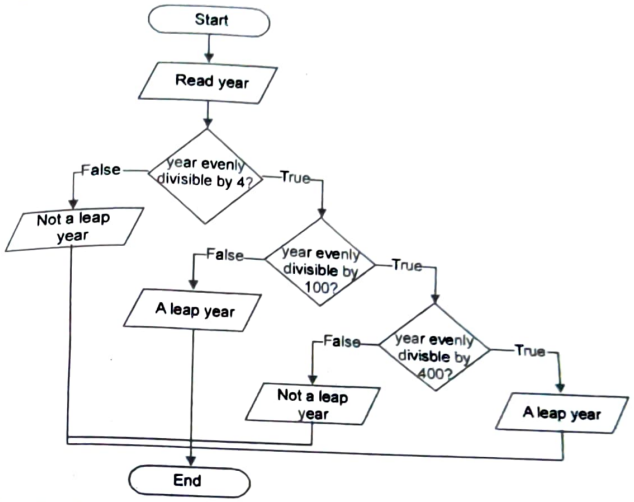
The parenthesis operator and increment decrement operator has highest precedence whereas comma operator has lowest precedence.

Q. 3. (b) WAP to determine whether input year is leap year or not using logical operators. Draw a flowchart of same. (6)

Ans. (b)

```
#include <stdio.h>
int main()
{
 int year;
 printf("\n Please Enter any year you wish \n");
 scanf("%d", &year);
 if (( year%400 == 0) || (( year%4 == 0 ) &&( year%100 != 0)))
   printf("\n %d is a Leap Year. \n", year);
 else
   printf("\n %d is not the Leap Year. \n", year);
 return 0;
}
```


Flowchart for Leap Year



Q. 4. Explain any two with suitable example

(a) Formatted console I/O and Unformatted console I/O (5)

Ans. Refer Q. 4. (a) First Term Exam feb 2017.

Q. 4. (b) Break and Continue with example program (5)

Ans.

S.No.	Break	Continue
1.	Break statement is used in switch and loops.	Continue statement is used in loops only.

2.	When break is encountered the switch or loop execution is immediately stopped.
	Example:
	#include<stdio.h>
	int main(){
	int i;
	for(i=0;i<5;++i){
	if(i==3)
	if(i==3)
	break;
	printf("%d",i);
	}
	return 0;}
	}
		When continue is encountered, the statements after it are skipped and the loop control jump to next iteration.
		Example:
		#include<stdio.h>
		int main(){
		int i;
		for(i=0;i<5;++i){
		continue;
		printf("%d",i);
		}
		return 0;}
		}

3.	**Output** 012	**Output:** 0124

Q. 4. (c) Storage classes in C (5)

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program.There are following storage classes which can be used in a C Program:

auto
register
static
extern

Auto Storage Class is the default storage class for all local variables.

```
{
int Month;
//or
auto int Month; //Both declarations are same
}
```

Auto can only be used within functions, i.e. local variables.

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

Example: register int month;

Static Storage Class is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

Static variables can be 'seen' within all functions in this source file. Static can also be defined within a function. If this is done, the variable is initialised at run time but is not reinitialized when the function is called. Thus, inside a function static variable retains its value during various calls.

External storage class: External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

Example:

```
main()
{
extern int x;
x=10;
printfl"%d",x);  .
I
int x; // Global variable x;
```

Q. 4. (d) Linkers vs Loaders (5)

Ans. In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not built in libraries, it also links the user defined functions to the user defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

Loader: Loader is a program that loads machine codes of a program into the system memory. In Computing, a **loader** is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

END TERM EXAMINATION [MAY-JUNE 2018]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING
[ETCS–108]

Time : 3 hrs. **M.M. : 75**

Note: Attempt any five questions including Q.No 1 which is compulsory.

Q.1. Answer the following:-

(a) Define break statement. How does continue statement differ from break statement? (5)

Ans. Refer Q.4.(b) First term exam Feb 2017

Q. 1. (b) What is meant by function? List its characteristics and name and five inbuilt functions with their purpose. (5)

Ans. A function declaration tells the compiler about a function's name, **return type**, and **parameters**. A function **definition** provides the actual body of the function. The C standard library provides numerous built-in functions that your program can call.

Functions Provides these features

- **Reusability of Code :** Means Once a Code has Developed then we can use that Code any Time.
- **Remove Redundancy:** Means a user doesn't need to Write Code Again and Again.
- **Decrease Complexity:** Means a Large program will be Stored in the Two or More Functions. So that this will makes easy for a user to understand that Code.

There are Two Types of Functions

- Built in Functions
- User Defined functions

The Functions those are provided by C Language are refers to the Built in Functions For example. cin and cout, getch , clrser are the examples of built in Functions so that all the functions those are provided by the C++ Language are pre-defined and stored in the Form of header Files so that a user doesn't need to know how this Function has developed and a user just use that Function. Examples are

- char *strcpy (char *dest, char *src);

Copy src string into dest string.

- char *strncpy(char *string1, char *string2, int n);

Copy first n characters of string2 to string 1 .

- int strcmp(char *string1, char *string2);

Compare string1 and string2 to determine alphabetic order.

- int strncmp(char *string1, char *string2, int n);

Compare first n characters of two strings.

- int strlen(char *string);

Determine the length of a string.

On the other hand the Functions those are developed by the user for their Programs are known as User Defined Programs.

Q. 1. (c) Describe the various phases of compilations. Is there any relation between compilation and pre processor directives? Justify your answer. (5)

Ans. Compiling a C program is a multi-stage process. The process can be split into four separate stages: **Preprocessing, compilation, assembly**, and **linking**.

Preprocessing is the first stage of compilation. In this stage, lines starting with a #character are interpreted by the preprocessor as preprocessor commands. These commands form a simple macro language with its own syntax and semantics. This language is used to reduce repetition in source code by providing functionality to inline files, define macros, and to conditionally omit code.

Compilation

The second stage of compilation is confusingly enough called compilation. In this stage, the preprocessed code is translated to assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

Assembly

During this stage, an assembler is used to translate the assembly instructions to object code. The output consists of actual instructions to be run by the target processor.

Linking

The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

C PREPROCESSOR DIRECTIVES:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Q. 1. (d) Differentiate between array and structure. Define an array of structure with an example. (5)

Ans. The following are the differences between structures and arrays:

1. An array is a collection of related data elements of same type.
2. An array is a derived data type.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it.

Structures

1. Structure can have elements of different types
2. A structure is a programmer-defined data type
3. In structure, first we have to design and declare a data structure before the variable of that type are declared and used.

```
#include <stdio.h>
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
```

```
} book[3];
int main(int argc, char *argv[])
{
int i;
for (i=0;i<3;i++)
{
printf ("\nEnter the Name of Book :");
gets (book[i]. bname);
printf ("\n Number of Pages:");
scanf ("%d" ,book[i]. pages) ;
printf ("\nEnter the Price of Book :");
scanf("%f",book[i],price);
}
printf("\n------- Book Details ---------");
for(i=0;i<3;i++)
{
printf("\n Name of Book :%s", book[i].bname);
printf ("\Number of Pages :%d", book [i]. pages);
printf ("/nPrice of Book: %f", book [i] price);
}
return 0;
}
```

Output of the Structure Example:

```
Enter the Name of Book   : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book   : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book   : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500
----------Book Details --------
Name of Book       : ABC
Number of Pages    : 100
Price of Book      : 200
Name of Book       : EFG
Number of Pages    : 200
Price of Book      : 300
Name of Bool       : HIJ
Number of Pages    : 300
Price of Book      : 500
```


Q. 1. (c) Define associativity of operators. Write the output of the following statements; All are integer values.

(i) i= 3; j =7 i* 2*j' print f ("%d", i)' output : error

(ii) i=j = k = 3; print f ("%d", (i + 5) % (j + 2)/k); output : zero.

Ans. If two operators of same precedence (priority) is present in an expression, Associativity of operators indicate the order in which they execute.

Example of associativity

1 == 2 != 3

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression on the left is executed first and moves towards the right.

Thus, the expression above is equivalent to :

((1 == 2) != 3)

i.e, (1 == 2) executes first resulting into 0 (false)

then, (0 != 3) executes resulting into 1 (true)

Q. 2. (a) Define an algorithm and list its characteristics. .Write an algorithm for measuring 4 liter of water if only 5 liter and 3 liter jugs are available. (6)

Ans. Algorithm for Water Jug Problem

An **algorithm** should have 1 or more well-**defined** outputs, and should match the desired output.

The characteristics of a good algorithm are:

- Precision – the steps are precisely stated(defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X–d, Y+d)

Here, (X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2

Examples: Input is 4 3 2

Output will be ((0,0), (5,0), (2,3), (2,0), (0,2),(5,2), (4,3),(4,0))

The output can be explained as:

- Fill the 5 litre can from the tap
- Empty the 5 litre can into the 3 litre can - leaving 2 litres in the 5 litre can.
- Pour away the contents of the 3 litre can.
- Fill the 3 litre can with the 2 litres from the 5 litre can - leaving 2 litres in the 3 litre can.

- Fill the 5 litre can from the tap.
- Fill the remaining 1 litre space in the 3 litre can from the 5 litre can. Leaving 4 **litres** in the 5 litre can.

Q. 2. (b) What is logical error? How does logical error differ from run time error and syntax error? Support your answer with examples. **(6.5)**

Ans. The program **with a runtime error** can be executed but fails under certain conditions. Example: You have a division statement **in** your program. **Syntax errors** are static **error** that can be detected by the compiler. **Runtime errors** are dynamic **error** that cannot be detected by the compiler.

A runtime error is a program error that occurs while the program is running. The term is often used in contrast to other types of program errors, such as syntax errors and compile time errors.

There are many different types of runtime errors. One example is a logic error, which produces the wrong output. For example, a miscalculation in the source code or a spreadsheet program may produce the wrong result when a user enters a formula into a cell. Another type of runtime error is a memory leak. This type of error causes a program to continually use up more RAM while the program is running. A memory leak may be due to an infinite loop, not deallocating unused memory, or other reasons.

A program crash is the most noticeable type of runtime error, since the program unexpectedly quits while running. Crashes can be caused by memory leaks or other programming errors. Common examples include dividing by zero, referencing missing files, calling invalid functions, or not handling certain input correctly.

A logic error is an error in a program's source code that gives way to unanticipated and erroneous behavior. A logic error is classified as a type of runtime error that can result in a program producing an incorrect output. It can also cause the program to crash when running.

Logic errors are not always easy to recognize immediately. This is due to the fact that such errors, unlike that of syntax errors, are valid when considered in the language, but do not produce the intended behavior. These can occur in both interpreted and compiled languages.

A logic error is also known as a logical error.

Q.3. (a) Describe storages classes in C with suitable examples. **(6)**

Ans. Refer Q. 4. (c) of First Term Exam 2018.

Q. 3. (b) Describe various operators used in C language. Which operator has the highest precedence?

Ans. Refer Q. 3. (a) First Term Exam 2018. **(6.5)**

Q. 4. (a) Using the for loop, write a program to print the sum of first n odd numbers. **(4)**

Ans.
```
#include<stdio.h>
int main()
{
 int i, number, Sum = 0;
 printf("\n Please Enter the Maximum Limit Value :");
 scanf("%d", &number);
 printf("\n Odd Numbers between 0 and %d are :", number);
 for(i = 1; i <= number; i++)
```



```
 {
        if ( i%2 != 0 )
            {
                printf("%d", i);
    Sum = Sum + i;
            }
 }
 printf("\n \n The Sum of Odd Numbers from 1 to %d  = %d", number, Sum);

 return 0;
}
```

Q. 4. (b) Using the do-while loop, write a program that prints the factorial of n. **(4)**

Ans.
```
#include<stdio.h>
#include<conio.h>
void main()
{
  int n,i=1,f=1;
  clrscr();
  printf("\n Enter The Number:");
  scanf("%d",&n);
  //LOOP TO CALCULATE THE FACTORIAL OF A NUMBER
  do
  {
    f=f*i;
    i++;
  }while(i<=n);
  printf("\n The Factorial of %d is %d",n,f);
  getch();
}
```

Q. 4. (c) Using the while statement, write a program that reverses the digits n. **(4.5)**

Ans.
```
/* C program to all natural numbers in reverse from n to 1 using while loop */
#include <stdio.h>
int main()
{
  int n;
  /*
   * Input a number from user
   */
  printf("Enter value of n:");
  scanf("%d", &n);
  while(n>=1)
```

```
    {
      printf("%d\n", n);
      n--;
    }
    return 0;
}
```

Q. 5. (a) Describe the methods of parameter passing with the help of suitable examples. (6)

Ans. Call By Value : In Call By Value method, the arguments are passed to a function using the **value of the variable.** There is a second copy of the variable is formed while passing values from one function to another.

Therefore, the original values or the **original variables do not change.** The call by value and call by reference approach are mostly used while passing values to functions in C programming.

The original variables are called **Actual Parameters** whereas the new values that are passed to the function are called as **Formal Parameters.**

In case of call by value approach, the copy of the values is passed to the function. The actual parameters and formal parameters are thereby created in different memory locations. Therefore, the original value is not modified in call by value method in C programming.

The values passed to the **User Defined Function** are created on **Stack** memory location. Therefore, the changes made within the user defined function is not visible in the **main**() method or any other method for that matter.

The disadvantage with pass by value method is that there are two copies created for the same variable which is not **memory efficient.** However, the advantage of call by value approach is that this method doesn't change the original variables thereby preserving data. The following program will demonstrate how to write a C code to swap two variables using call by value method.

In the following C program to swap variables with Call By Value approach, the variables **var1 and var2** are passed to the **swap**() method by using their values and not the addresses.

When the user defined function **swap**() receives the values of **var1** and **var2**, two local copies are created for that variables in the function. These local values **val1** and **val2** are manipulated by the program itself and there is no effect on the actual parameters var1 and var2 that were passed from the main() function.

C Program To Swap Variables using Call by Value Method

```
1.  include<stdio.h>
2   int swap(int a, int b);
3   int main()
4   {
5   int main()
6   {
7   int num 1, num2;
8   printf("\nEnter The First Number:\t");
9   scanf("%d", &num1);
10  printf("\nEnter The Second Number: \t");
```



```
11  scapf("%", &num2);
12  swap(num1,num2);
13  printf("/nOld values\n");
14  printf("\nFirst Number = %d\nSecond Number = %d\n", num 1, num2);
15  printf("\n");
16  return 0;
17  }
18  int swap(int a, int b)
19  {
20  int temp;
21  temp = a;
22  a = b;
23  b = temp;
24  printf("\nNew Values\n");
25  printf("\nFirst Number = %d\nSecond Number = %d\n", a, b);
26  }
```

Call By Reference Method

In Call By Reference method, the arguments are passed to another function using the **address of the variables.** This address is also known as **Reference Pointers.** There is no second copy of the variable while passing values from one function to another.

Since, the original variables' address is passed, the modifications done to the variable in the User Defined will **change the original variable** as well. Therefore, the Original Values will change..

The original variables are called **Actual Parameters** whereas the new values that are created within the user defined function are called **Formal Parameters.** In case of call by reference approach, the address of the original values are passed to the function.

The actual parameters and formal parameters are thereby created in the **same memory location.** All the modification is done on actual parameters. Therefore, the original value is modified in Pass By Reference method in C programming.

The changes made within the user defined function are visible in the **main**() method and any other method for that matter. We have demonstrated below a C code for call by reference in C using Pointers.

The Advantage of using pass by reference approach is that it does not create duplicate data for holding only one value. Therefore, this helps to **save memory space.** The following program will demonstrate how to write a C program to swap two variables using pass by reference method.

In the C program to swap variables with passing by reference approach, the variables **var1** and **var2** are passed to the **swap**() method by using their addresses and not the values.

When the User Defined Function **swap**() receives the addresses of var1 and var2, the pointers ***ptr1 and *ptr2** holds the values of the var1 and var2 since the addresses are passed to them. Therefore, no local copies are created. The original variables var1 and var2 will be directly modified by prt1 and ptr2.

C Program To Swap Variables using Call By Reference Method

```
1   #include<stdio.h>
2   int swap(int *ptr1, int *pt2);
3   int main()
4   {
5   int var1, var2;
```

```
6   printf("Enter The First Number: \t");
7   scanf("%d", &var1);
8   printf("Enter The Second Number:\t");
9   scanf("%d", &var2);
10  swap(&var1, &var2);
11  printf("The Swapped Values are:\n");
12  printf("First Number = %d\nSecond Number = %d\n", var1, var2);
13  return 0;
14  }
15  int swap(int *ptr1, int *ptr2)
16  {
17  int temp;
18  temp = *ptr1;
19  *ptr1 = *ptr2;
20  *ptr2 = temp;)
21  }
```

Q. 5. (b) What do you mean by recursion? Write a program using a recursive function. **(6.5)**

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

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

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

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

Program Factorial

The following example calculates the factorial of a given number using a recursive function

```
Live Demo
#include <stdio.h>
unsigned long long int factorial(unsigned int i) {

  if(i <= 1) {
    return 1;
  }
  return i * factorial(i - 1);
}
int main() {
  int i = 12;
```



```
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

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

```
Factorial of 12 is 479001600
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function

```
Live Demo
#include <stdio.h>
int fibonacci(int i) {
  if(i == 0) {
    return 0;
  }
  if(i == 1) {
    return 1;
  }
  return fibonacci(i - 1) + fibonacci(i - 2);
}
int main() {
  int i;
  for (i = 0; i < 10; i++) {
    printf("%d\t\n", fibonacci(i));
  }
  return 0;
}
```

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

```
0
1
1
2
3
5
8
13
21
34
```

Q. 6. (a) What are pointers? Why are they needed? Explain with an example. **(6)**

Ans. Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Pointer Syntax : data_type *var_name; Example : int *p; char *p;

Where, * is used to denote that "p" is pointer variable and not a normal variable.

The main points about pointers:

Normal variable stores the value whereas pointer variable stores the address of the variable.

- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

EXAMPLE PROGRAM FOR POINTERS IN C:

```
#include <stdio.h>
int main()
{
int *ptr, q;
q = 50;
/* address of q is assigned to
ptr */
ptr = &q;
 /* display q's value using ptr
 variable */
 printf("%d", *ptr);
 return 0;}
```

The reason is that pointers are used to bodge into C some vital features which are missing from the original language: arrays, strings, & writeable function parameters. They can also be used to optimize a program to run faster or use less memory that it would otherwise.

One of the complications when reading C programs is that a pointer could be being used for any several or all of these different reasons with little or no distinction in the language so, unless the programmer has put in helpful comments, one has to follow through the program to see what each pointer is used for in order to work out why it is there instead of a plain simple variable.

Q. 6. (b) Give two matrices $A[\,]_{m \times r}$ and $A[\,]_{r \times n}$. Write a program to calculate product of two matrices such that: (6.5)

$C[\,]_{m \times n} = A[\,]_{m \times r} * B[\,]_{r \times n}$

Ans.

```
#include <stdio.h>
int main()
    {
        int m, n, p, q, c, d, k, sum = 0;
        int first[10][10], second[10][10], multiply[10][10];
printf("Enter the number of rows and columns of first matrix\n");
scanf("%d%d", &m, &n);
printf("Enter the elements of first matrix\n");
for (c = 0 ; c < m ; c ++ )
for (d = 0 ; d < n ; d++)
scanf("%d",&first[c]fd]);
priritf("Enter the number pf rows and columns of second matrix\n");
scanf("%d%d", &p, &q);
```



```
if(n!=p)
   printf("Matrices with entered orders can't be multiplied with each other.\n");
     else
        {
       printf("Enter the elements of second matrix\n");
for (c = 0 ; c < p ; C++)
for (d = 0 ; d < q ; d++)
 scanf("%d", &second[c][d]);
for (c = 0 ; c < m ; C++)
  {
for (d = 0 ; d < q ; d++)
  {
for (k = 0 ; k < p ; k++)
  {
       sum = sum + first[c][k]*second[k][d];
     }
 multiply[c][d] = sum;
 sum = 0;
 }
 }
 printf("Product of entered matrices:-\n");
 for (c == 0 ; c < m ; c++)
 {
 for ( d = 0 ; d < q ; d++)
 printf("%d\t", multiply[c][d]);
 printf("\n");
 }
 }
 return 0;
 }
```

Q. 7. (a) Declare a structure for employee's record. Also determine size of this structure. (6.5)

Ans. Structure for employee's Record will be as follows:

```
struct details
{
 char name[30];
 int age;
 char address[500];
 float salary;
};
```

The size will be : 30+2+500+4 = 536 bytes

Program for employee Record will be as follows:

```
#include <stdio.h>
#include <conio.h>
struct details
{
 char name[30];
```

```
int age;
char address[500];
float salary;
};
int main()
{
 struct details detail;
 clrscr();
 printf("\nEnter name:\n");
 gets(detail.name);
 printf("\nEnter age:\n");
 scanf("%d",&detail.age);
 printf("\nEnter Address:\n");
 gets(detail.address);
 printf("\nEnter Salary:\n");
 scanf("%f",&detail.salary);
 printf("\n\n\n");
 printf("Name of the Employee : %s \n",detail.name);
 printf("Age of the Employee : %d \n",detail.age);
 printf("Address of the Employee : %s \n",detail.address);
 printf("Salary of the Employee : %f \n",detail.salary);
 getch();
}
```

Q. 7. (b) Define a random access file. What are the various operations commonly used in file handling? Discuss with examples. (6.5)

Ans. The terms random access and sequential access are used to describe data files. A random-access data file enables you to read or write information anywhere in the file. In a sequential-access file, you can only read and write information sequentially, starting from the beginning of the **file Writing data randomly** to a **Random-Access File**. The program writes data to the **file** "credit.dat". It uses the combination of fseek and fwrite to store data at specific locations in the **file**. Function **fseek sets the file** position pointer to a specific position in the **file**, then fwrite writes the data.

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file file *fp.

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	Description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file

rewind() set the position to the begining point

For Example : Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

```
#include<stdio.h>
int main()
{
  FILE *fp;
  char ch;
  fp = fopen("one.txt", "w");
  printf("Enter data...");
  while( (ch = getchar()) != EOF) {
    putc(ch, fp);
  }
  fclose(fp);
  fp = fopen("one.txt", "r");
  while( (ch = getc(fp)! = EOF)
  printf("%c",ch);
  // closing the file pointer
  fclose(fp)
  return 0;
}
```

Q. 8. Write short notes on the following with the help of examples:- (4)

(a) Argument passing using pointers

Ans. Argument passing using pointers

When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value.

This is because, pointer stores the location of the memory, and not the value.

Program to swap two number using call by reference to show argument passing through pointers:

```
/* C Program to swap two numbers using pointers and function. */
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
  int num1 = 5, num2 = 10;
  // address of num1 and num2 is passed to the swap function
  swap( &num1, &num2);
  printf("Number1 = %d\n", num1);
  printf("Number2 = %d", num2);
  return 0;
}
void swap(int * n1, int * n2)
{
  // pointer n1 and n2 points to the address of num1 and num2 respectively
  int temp;
  temp = *n1;
```

```
    *n1 = *n2;
    *n2 = temp;
}
```

Output

Number1 = 10

Number2 = 5

Q. 8. (b) Array of pointer

Ans. Array of Pointers: (4)

In C, an array of pointer is an indexed set of variables in which the variables are pointers (a reference to a location in memory).

Pointers are an important tool in computer science for creating, using, and destroying all types of data structures. An array of pointers is useful for the same reason that all arrays are useful: it allows you to numerically index a large set of variables.

Below is an array of pointers in C that sets each pointer in one array to point to an integer in another and then print the values of the integers by dereferencing the pointers. In other words, this code prints the value in memory of where the pointers point.

```
#include <stdio.h>
const int ARRAY_SIZE = 5;
int main ()
{
  /* first, declare and set an array of five integers:   */
  int array_of_integers[] = {5, 10, 20, 40, 80};
  /* next, declare an array of five pointers-to-integers: */
  int i, *array_of_pointers[ARRAY_SIZE];
  for ( i = 0; i < ARRAY_SIZE; i++)
  {
    /* for indices 1 through 5, set a pointer to
      point to a corresponding integer:              */
    array_of_pointers[i] = &array_of_integers[i];
  }
  for ( i = 0; i < ARRAY_SIZE; i++)
  {
    /* print the values of the integers pointed to
      by the pointers:                               */
    printf("array_of_integers[%d] = %d\n", i, *array_of_pointers[i] );
  }
  return 0;
}
```

The output of the above program is:

array_of_integers[0] = 5

array_of_integers[1] = 10

array_of_integers[2] = 20

array_of_integers[3] = 40

array_of_integers[4] = 80

Q. 8. (c) Call by reference.

Ans. Refer Q. 5. (a) End Term May June 2018. (4.5)

FIRST TERM EXAMINATION [FEB. 2019]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING
[ETCS–108]

Time : 1½ hrs. **M.M. : 30**

Note : *Q. No. 1 which is compulsory. Attempt any two more questions from the rest.*

Q.1. Answer the following:

Q.1. (a) Differentiate between a keyword and identifier. List the criteria of declaring an identifier. **(2)**

Ans.

Identifier	Keyword
Identifiers are names that are given to various program elements, such as variable, function and arrays.	C take some reserved word called keyword, They have predefined meaning in C.
Identifiers consist of letters and digits.	Keyword consist only letter.
Identifier's first character must be a letter.	Keyword's all character is letter.
Indentifiers Upper and lowercase letter is use.	Keywords are all lowercase.
Upper and lowercase are not equivalent.	Upper and lowercase are also not equivalent.
Like: X, sum_5,_weather etc. But 4th is not identifier as first character must be a letter.	Like: auto, short long etc.

There are 32 Keywords in C languge.

Identifier naming rules: The name of a variable (or function, type, or other kind of item) is called an identifier. C++ gives you a lot of flexibility to name identifiers . However, there are a few rules that must be followed when naming identifiers:

(a) The identifier can not be a keyword. Keywords are reserved.

(b) The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. That means the name can not contain symbols (except the underscore) nor whitespace (spaces or tabs).

(c) The identifier must begin with a letter (lower or upper case) or an underscore. It can not start with a number.

(d) C++ is case sensitive, and thus distinguishes between lower and upper case letters. nvalue is different than nValue is different than NVALUE.

Q.1. (b) Write the steps involved from sample.c to sample.exe. **(2)**

Ans.

1. Step 1: Creating a Source Code

a. Click on the Start button

b. Select Run

c. Type cmd and press Enter

d. Type cd c:\TC\bin in the command prompt and press Enter

e. Type TC press Enter

f. Click on File -> New in C Editor window

g. Type the program

h. Save it as FileName.c (Use shortcut key F2 to save)

Step 2: Compile Source Code (Alt + F9): The compilation is the process of converting high-level language instructions into low-level language instructions. We use the shortcut key Alt + F9 to compile a C program in Turbo C.

Whenever we press Alt + F9, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into object code and stores it as a file with .obj extension. Then the object code is given to the Linker. The Linker combines both the object code and specified header file code and generates an Executable file with a .exe extension.

Step 3: Executing / Running Executable File (Ctrl + F9): After completing compilation successfully, an executable file is created with a .exe extension.We use a shortcut key Ctrl + F9 to run a C program. Whenever we press Ctrl + F9, the .exe file is submitted to the CPU. On receiving .exe file, CPU performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called User Screen.

Step 4: Check Result (Alt + F5): After running the program, the result is placed into User Screen. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key Alt + F5 to open the User Screen and check the result.

Q.1. (c) What would be appropriate data type to store the following. (2)

(i) Distance among the galaxies (ii) Factorial of a 2-digit number

(iii) An excleamation mark (!) (iv) Average age of student in your class.

Ans. (*i*) long double (*ii*) array of integers (*iii*) Char (*iv*) float

Q.1. (d) Define Precedence and associativity of operators with examples. (2)

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

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

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

```
#include <stdio.h>
main( ) {
 int a = 20;
 int b = 10;
 int c = 15;
 int d = 5;
 int e;
 e = (a + b) * c / d; // ( 30 * 15 ) / 5
 printf("Value of (a + b) * c / d is : %d\n", e );
 e = ((a + b) * c) / d; // (30 * 15 ) / 5
 printf("Value of ((a + b) * c) / d is : %d\n" , e );
 e = (a + b) * (c / d); // (30) * (15/5)
```



```
 printf("Value of (a + b) * (c / d) is : %d\n", e );
 e = a + (b * c) / d; // 20 + (150/5)
 printf("Value of a + (b * c) / d is : %d\n" , e );
 return 0;
}
```

When you compile and execute the above program, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

Operators Associativity: It is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left. For example: '*' and '/' have same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Q.1. (e) What is header file? List any three header files with their purpose. (2)

Ans .A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

We can use a header file by including it with the C preprocessing directive #include..

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

Stdio.h	input/output functions
Conio.h	console input/output
Math.h	mathematical functions

Q.2. (a) Describe various operators with the help of suitable examples. (5)

Ans. Refer Q.3. (a) First Term Examination 2018. (Page. 3-2018)

Q.2. (b) Write the output of the following statements. All are integer values. (5)

(i) i = 8; j = 7; j + = i * i; printf("%d",j);

(ii) i = j = k = 1; printf("%d",(j+2) %k/(i+1));

(iii) i = 9; j = 14; k = 6; ((i%4) * (5 + (j – 2)/(k+3))) = ?

(iv) a + 5 < 8 && 6! = 5; printf("%d",a);

(v) –10% –3 = ?

Ans. (i) 71 (ii) 0

(iii) 6 (iv) 1

(v) –1

Q.3. (a) What in the purpose of loops in programming. Differentiate between various loops with examples. (5)

Ans. Loop control statements in C are used to perform looping operations until the given condition is true. Control comes out of the loop statements once condition becomes false.

There are 3 types of loop control statements in C language. They are,

1. for 2. while 3. do-while

• Syntax for each C loop control statements are given in below table with description.

S.No.	Loop Name	Syntax	Description
1	for	for (exp1; exp2; expr3) { statements; }	Where, exp1 - variable initialization (Example: i=0, j=2, k=3)

			exp2 - condition checking (Example: i>5, j<3, k=3) exp3 - increment/decrement (Example: ++i, j-, ++k) where, condition might be a>5, i<10
2	while	while (condition) { statements; }	
3	do while	do { statements; } while (condition);	where, condition might be a>5, i<10

Example program (for loop) in C: In for loop control statement, loop is executed until condition becomes false.

```
#include <stdio.h>
int main()
{
int i;
for(i=0;i<10;i++)
 {
printf("%d ",i);
 }
}
```

Output:

0 1 2 3 4 5 6 7 8 9

Example program (while loop) in C: In while loop control statement, loop is executed until condition becomes false.

```
#include <stdio.h>
int main()
{
int i=3;
while(i<10)
 {
printf("%d\n",i);
i++;
 }
}
```

Output:

3 4 5 6 7 8 9

Example program (do while loop) in C: In do-while loop control statement, while loop is executed irrespective of the condition for first time. Then 2nd time onwards, loop is executed until condition becomes false.

```
#include <stdio.h>
int main()
{
int i=1;
do
 {
printf("Value of i is %d\n",i);
i++;
}while(i<=4 &&i>=2);
}
```


Output:

Value of i is 1
Value of i is 2
Value of i is 3
Value of i is 4

Q.3. (b) Write a program to find number and their sum between 100 to 1000 which are divisble by 7. **(5)**

Ans. WAP to find numbers and their sum between 100 to 1000 which are divisible by 7

```
#include <stdio.h>

void main()
{
  int i, sum=0;
  printf("Numbers between 100 and 1000, divisible by 7 : \n");
  for(i=101;i<1000;i++)
  {
    if(i%7==0)
    {
      printf("% 5d",i);
      sum+=i;
    }
  }
  printf("\n\nThe sum : %d \n",sum);
}
```

Q.4. (a) What is a switch statement ? List its disadvantages. Write a program using switch statement. **(5)**

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

Example:

```
#include <stdio.h>
int main () {
  /* local variable definition */
  char grade = 'B';
  switch(grade) {
    case 'A' :
      printf("Excellent!\n" );
      break;
    case 'B' :
    case 'C' :
      printf("Well done\n" );
      break;
    case 'D' :
      printf("You passed\n" );
      break;
    case 'F' :
      printf("Better try again\n" );
```

```
    break;
  default :
    printf("Invalid grade\n" );
 }
 printf("Your grade is %c\n", grade );
 return 0;
}
```

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

Well done
Your grade is B

Disadvantages of switch statements

a. Float constant cannot be used in the switch as well as in the case.
b. You can not use the variable expression in case.
c. You cannot use the same constant in two different cases.
d. We cannot use the relational expression in case.

Q.4. (b) Define break and continue statements with example.

Ans. Refer Q.4 (b) First Term Examination 2018. (3)

Q.4. (c) What is the purpsose of getchar() and putchar()? (2)

Ans. The C library function **int getchar(void)** gets a character (an unsigned char) from stdin. This is equivalent to **getc** with stdin as its argument.

Declaration for getchar() function:

int getchar(void)

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Example

The following example shows the usage of getchar() function.

```
#include<stdio.h>
int main ()
{
char c;
  printf("Enter character: ");
  c = getchar();
  printf("Character entered: ");
  putchar(c);
return(0);
}
```

Output

Enter character: a
Character entered: a

The putchar macro is used to write a single character on the standard output stream (i.e., display). A call to this macro takes the following form: putchar (c), where c is an integer expression representing the character being written. Although this macro expects an argument of type int, we usually pass a character to it. The argument value (c) is converted to unsigned char and written to the standard output stream.

END TERM EXAMINATION [MAY-JUNE 2019]
SECOND SEMESTER [B.TECH]
INTRODUCTION TO PROGRAMMING
[ETCS–108]

Time : 3 hrs. **M.M. : 75**

Note : *Attempt any five questions including Q. no. 1 which is compulsory.*

Q.1. Differentiate between the following (any five): **(5 × 5 = 25)**

Q.1. (a) Ternary and Bitwise Operator

Ans. The conditional operator/Ternary Operator is sometimes called a ternary operator because it involves three operands.

Syntax: The conditional operator contains a condition followed by two statements or values.

If the condition is true the first statement is executed, otherwise the second statement or value.

condition ? (value 1) : (value 2)

Example: younger = son < father ? 18 : 40;

In the above example, son's age is 18 whereas father's age is 40. But we need the younger age so we make use of conditional operator to extract least age.

Bitwise Operator: In C, the following 6 operators are bitwise operators:

(a) The & (bitwise AND) in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

(b) The | (bitwise OR) in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

(c) The ^ (bitwise XOR) in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

(d) The << (left shift) in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

(e) The >> (right shift) in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

(f) The ~ (bitwise NOT) in C or C++ takes one number and inverts all bits of it

Q.1. (b) External and Internal documentation.

Ans. Internal documentation is written in a program as comments. External documentation is written in a place where people who need to use the software can read about how to use the software. External documentation can be broken down into *library documentation*, which describes tools that a programmer can use, and user documentation, which is intended for users of an application.

The line between internal and library documentation is not clearcut because the trend is to write library documentation inside a program as comments, relying on software that extracts the documentation and puts it into a form suitable for people who only want to use the library. For example, the extensive Java library documentation is created by software called *javadoc* that reads Java programs, including comments, and writes documentation.

Q.1. (c) Primitive and Non Primitive data types.

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program.There are following storage classes which can be used in a C Program:

auto
register
static
extern

Auto Storage Class is the default storage class for all local variables.

```
{
int Month;
//or
auto int Month; //Both declarations are same
}
```

Auto can only be used within functions, i.e. local variables.

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

Example: register int month;

Static Storage Class is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

Static variables can be 'seen' within all functions in this source file. Static can also be defined within a function. If this is done, the variable is initialised at run time but is not reinitialized when the function is called. Thus, inside a function static variable retains its value during various calls.

External storage class: External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

Example:

```
main()
{
extern int x;
x=10;
printf("%d",x);
}
int x; // Global variable x;
```

Primitive and non-primitive data types: A data type is a classification of data, which can store a specific type of information. Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain. The term "data type" and "primitive data type" are often used interchangeably. Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types.

- For eg. Character (character , char);
- Integer (integer , int , short , long , byte) with a variety of precisions;
- Floating-point number (float , double , real , double precision);
- Fixed-point number (fixed) with a variety of precisions and a programmer-selected scale.

Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "lastname" and define it as a string data type. The variable will then store data as a string of characters.

- Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data. The data types that are derived from primary data types are known as **non-primitive data types.**
- The non-primitive datatypes are used to **store group of values.**
- The non-primitive data types are :
  - Arrays
  - String
  - Structures

Q.1. (d) Loader and Linker .

Ans. Refer Q4(d) First Term Examination 2018

Q.1. (e) Declaration and Definition of a variable.

Ans. Declaration of a variable is for informing to the compiler the following information: name of the variable, type of value it holds and the initial value if any it takes. i.e., declaration gives details about the properties of a variable. Whereas, **Definition** of a variable says where the variable gets stored. i.e., memory for the variable is allocated during the definition of the variable.

In C language definition and declaration for a variable takes place at the same time. i.e. there is no difference between declaration and definition. For example, consider the following declaration int a;

Here, the information such as the variable name: a, and data type: int, which is sent to the compiler which will be stored in the data structure known as symbol table. Along with this, a memory of size 2 bytes (depending upon the type of compiler) will be allocated.

Examples of Definition are:

```
int a;
int b = 0;
int myFunc (int a, int b){ return a + b; }
struct _tagExample example;
```

Q.1. (f) Auto, Register, static and Extern variables.

Ans. Refer Q.4 (c) Term Examination 2018.

Q.2. (a) Explain how strings can be stored using a multidimensional arrays? (5)

Ans. In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0] [0]	x[0] [1]	x[0] [2]	x[0] [3]
Row 1	x[1] [0]	x[1] [1]	x[1] [2]	x[1] [3]
Row 1	x[2] [0]	x[2] [1]	x[2] [2]	x[2] [3]

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, the array y can hold 24 elements.

Initializing a multidimensional array

Initialization of a 2d array

```
// Different ways to initialize two-dimensional array
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Initialization of a 3d array

Here's an example,

```
int test[2][3][4] = {
  {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
  {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Example 1: Two-dimensional array to store and print values

// C program to store temperature of two cities of a week and display it.

```
#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;
int main()
{
 int temperature[CITY][WEEK];
 // Using nested loop to store values in a 2d array
 for (int i = 0; i < CITY; ++i)
 {
  for (int j = 0; j < WEEK; ++j)
  {
   printf("City %d, Day %d: ", i + 1, j + 1);
   scanf("%d", &temperature[i][j]);
  }
 }
 printf("\nDisplaying values: \n\n");
 // Using nested loop to display values of a 2d array
 for (int i = 0; i < CITY; ++i)
 {
  for (int j = 0; j < WEEK; ++j)
  {
   printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
  }
 }
 return 0;
}
```

Output

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26
```

Displaying values:

```
City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26
```


Q.2. (b) Differentiate between break and continue. Write a C program to print the pattern. **(7.5)**

```
*
* * *
* * * * *
* * * * * * *
```

Ans.

Break	Continue
A break can appear in both switch and loop (for, while, do) statements.	A continue can appear only in loop (for, while, do) statements.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break is encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration. All iterations of the loop are executed even if continue is encountered. The continue statement is used to skip statements in the loop that appear after the continue.
The break statement can be used in both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.

When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.
A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.

Star Pattern Program:

```
#include <stdio.h>
int main()
{
  int n, c, k;
  printf("Enter number of rows\n");
  scanf("%d", &n);
  for (c = 1; c <= n; c++)
  {
  for(k = 1; k <= c; k++)
    printf("*");
  printf("\n");
  }
  return 0;
}
```

Q.3. (a) Explain actual and formal parameters with examples. (5)

Ans. Actual Parameters are the values that are passed to the function when it is invoked while Formal Parameters are the variables defined by the function that receives values when the function is called.

Actual parameters are values that are passed to a function when it is invoked.

Consider the program:

```
#include <stdio.h>
void addition (int x, int y) {
int addition;
addition = x+y;
printf("%d",addition);
}
void main () {
addition (2,3);
addition (4,5);
}
```

According to the above C program, there is a function named addition. In the main function, the value 2 and 3 are passed to the function addition. This value 2 and 3 are the actual parameters. Those values are passed to the method addition, and the sum of two numbers will display on the screen. Again, in the main program, new two integer values are passed to the addition method. Now the actual parameters are 4 and 5. The summation of 4 and 5 will display on the screen.

Formal Parameters: A function or a method follows a syntax similar to those given below:

```
<return type> <method name> (formal parameters) {
//set of statements to be executed
}
```

The method name is to identify the method. The return type specifies the type of the value the method will return. If the method does not return a value, the return type is void. If the function is returning an integer value, then the return type is an integer. The formal parameter list is enclosed in parenthesis. The list contains variable names and data types of all the necessary values for the method. Each formal parameter is separated by a comma. When the method is not accepting any input values, then the method should have an empty set of parentheses after the method name. e.g. addition () { }; The statements that should be executed are enclosed in curly braces.

Q.3. (b) Distinguish between an array of structures and an array within a structure. (5)

Ans. Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type, therefore an array can be of structure type.

Syntax for declaring structure array

```
struct struct-name
{
    datatype var1;
    datatype var2;
    - - - - - - - - - -
    - - - - - - - - - -
    datatype varN;
};
struct struct-name obj [ size ];
```

Example for declaring structure array

```
#include<stdio.h>
struct Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};
void main()
{
    int i;
    struct Employee Emp[ 3 ];      //Statement  1

    for(i=0;i<3;i++)
    {
     printf("\nEnter details of %d Employee",i+1);
```

```
        printf("\n\tEnter Employee Id :");
        scanf("%d",&Emp[i].Id);
        printf("\n\tEnter Employee Name :");
        scanf("%s",&Emp[i].Name);
        printf("\n\tEnter Employee Age :");
        scanf("%d",&Emp[i].Age);
        printf("\n\tEnter Employee Salary :");
        scanf("%ld",&Emp[i].Salary);
    }
    printf("\nDetails of Employees");
    for(i=0;i<3;i++)
    printf("\n%d\t%s\t%d\t%ld",Emp[i].Id,Emp[i].Name,Emp[i].Age,Emp[i].Salary);
}
```

Output :

```
Enter details of 1 Employee
    Enter Employee Id : 101
    Enter Employee Name : Suresh
    Enter Employee Age : 29
    Enter Employee Salary : 45000
Enter details of 2 Employee
    Enter Employee Id : 102
    Enter Employee Name : Mukesh
    Enter Employee Age : 31
    Enter Employee Salary : 51000
Enter details of 3 Employee
    Enter Employee Id : 103
    Enter Employee Name : Ramesh
    Enter Employee Age : 28
    Enter Employee Salary : 47000
Details of Employees
    101   Suresh   29   45000
    102   Mukesh   31   51000
    103   Ramesh   28   47000
```

In the above example, we are getting and displaying the data of 3 employee using array of object. Statement 1 is creating an array of Employee Emp to store the records of 3 employees.

Array within Structure: Structure is collection of different data type. Like normal data type, It can also store an array as well.

Syntax for array within structure

```
struct struct-name
{
    datatype var1;              // normal variable
    datatype array [size];      // array variable
    ----------
    ----------
    datatype varN;
};
struct struct-name obj;
```


Example for array within structure

```
struct Student
{
    int Roll;
    char Name[25];
    int Marks[3];          //Statement 1 : array of marks
    int Total;
    float Avg;
};
void main()
{
    int i;
    struct Student S;
    printf("\n\nEnter Student Roll :");
    scanf("%d",&S.Roll);
    printf("\n\nEnter Student Name :");
    scanf("%s",&S.Name);
    S.Total = 0;
    for(i=0;i<3;i++)
    {
        printf("\n\nEnter Marks %d :",i+1);
        scanf("%d",&S.Marks[i]);

        S.Total = S.Total + S.Marks[i];
    }
    S.Avg = S.Total / 3;
    printf("\nRoll : %d",S.Roll);
    printf("\nName : %s",S.Name);
    printf("\nTotal : %d",S.Total);
    printf("\nAverage : %f",S.Avg);
}
```

Output :

```
Enter Student Roll : 10
Enter Student Name : Kumar
Enter Marks 1 : 78
Enter Marks 2 : 89
Enter Marks 3 : 56
Roll : 10
Name : Kumar
Total : 223
Average : 74.00000
```

In the above example, we have created an array Marks[ ] inside structure representing 3 marks of a single student. Marks[ ] is now a member of structure student and to access Marks[ ] we have used dot operator(.) along with object S.

Q.3. (c) What is dangling pointer ? What precautions should be taken to avoid it.? **(2.5)**

Ans. Dangling pointers in computer programming are pointers that point to a memory location that has been deleted (or freed).

Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

The system may reallocate the previously freed memory, unpredictable behavior may result as the memory may now contain completely different data.

Avoiding dangling pointer errors: We can avoid the dangling pointer errors by initialize pointer to NULL, after de-allocating memory, so that pointer will be no longer dangling. Assigning NULL value means pointer is not pointing to any memory location.

Q.4. (a) Explain argument passing using pointer. **(5)**

Ans. Refer Q.8. (a) End Term Examination 2018

Q.4. (b) Explain preprocessor. directives with examples. **(5)**

Ans. The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if

Preprocessors Examples:

#define MAX_ARRAYJLENGTH 20

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

Q.4. (c) What is self referential structure? **(3)**

Ans. Self referential structure is a structure that refers to itself to it contains a pointer which is of the type of structure itself.

For example, single link list structure. Apart from data, it contains a pointer which is used to point to the next element in the link list.

```
struct node {
        int data;
        struct node* next;
    };
```

Here "*next" is part of the struct node and is a pointer of the type "struct node" itself which makes this a self-referential structure.

A double link list has two pointers of the type "struct node", one points to next element and other points to previous element (well except for very first and very last elements, very first node will have "*prev" element set to null ie no more previous elements because it is the very first element. Very last node will have "*next" set to null because there are no more elements in the list as it is the very last element).

Q.4. (d) Define token with example? **(2.5)**

Ans. Tokens are the smallest elements of a program, which are meaningful to the compiler.

The following are the types of tokens: Keywords, Identifiers, Constant, Strings, Operators, etc.

Keywords: Keywords are predefined, reserved words in C and each of which is associated with specific features. These words help us to use the functionality of C language. They have special meaning to the compilers.

There are total 32 keywords in C: auto, break, char, continue etc.

Identifiers: Each program element in C programming is known as an identifier. They are used for naming of variables, functions, array etc. These are user-defined names which consist of alphabets, number, underscore '_'. Identifier's name should not be same or same as keywords. Keywords are not used as identifiers.

Strings: A string is an array of characters ended with a null character(\0). This null character indicates that string has ended. Strings are always enclosed with double quotes(" ").

Q.5. (a) Write a C program to compare contents of two files. **(6)**

Ans.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
   FILE *fp1  ;
   FILE *fp2  ;
   int cnt1 = 0;
   int cnt2 = 0;
   int flg  = 0;
   if( argc < 3 )
   {
      printf("Insufficient Arguments!!!\n");
      printf("Please use \"program-name file-name1 file-name2\" format.\n");
      return -1;
   }
   fp1 = fopen(argv[1],"r");
   if( fp1 == NULL )
   {
```

```
    printf("\n%s File can not be opened : \n",argv[1]);
    return -1;
  }
  // move file pointer to end and get total number of bytes
  fseek(fp1,0,SEEK_END);
  cnt1 = ftell(fp1);
  fp2 = fopen(argv[2],"r");
  if( fp2 == NULL )
  {
    printf("\n%s File can not be opened : \n",argv[2]);
    return -1;
  }
    // move file pointer to end and get total number of bytes
  fseek(fp2,0,SEEK_END);
  cnt2 = ftell(fp2);

  fseek(fp1,0,SEEK_SET);
  fseek(fp2,0,SEEK_SET);
  // check for the total number of bytes
  if( cnt1 != cnt2 ){
    printf("\nFile contents are not same\n");
  }
  else
  {
    while( ! feof(fp1) )
    {
        if( fgetc(fp1) != fgetc(fp2) )
        {
          flg = 1;
          break;
        }
    }
    if( flg )  printf("\nFile contents are not same.\n");
    else       printf("\nFile contents are same.\n");
  }
  fclose(fp1);
  fclose(fp2);
  return 0;
```



```
}
```

Output

Terminal command: ./compPrg file1.txt file2.txt

File contents are not same.

Q.5. (b) Differentiate between text mode and binary mode in file handling. (3)

Ans. At the time of execution, every program comes in main memory to execute. Main memory is volatile and the data would be lost once the program is terminated. If we need the same data again, we have to store the data in a file on the disk. A file is sequential stream of bytes ending with an end-of-file marker.

Types of file supported by C:

Text Files

Binary Files

Text file is human readable because everything is stored in terms of text. In binary file everything is written in terms of 0 and 1, therefore binary file is not human readable.

A newline(\n) character is converted into the carriage return-linefeed combination before being written to the disk. In binary file, these conversions will not take place.

In text file, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. There is no such special character present in the binary mode files to mark the end of file.

In text file, the text and characters are stored one character per byte. For example, the integer value 1245 will occupy 2 bytes in memory but it will occupy 5 bytes in text file. In binary file, the integer value 1245 will occupy 2 bytes in memory as well as in file.

Q.5. (c) Differentiate between append and write mode. (3.5)

Ans. A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure. In C language, we use a structure pointer of file type to declare a file. FILE *fp; C provides a number of functions that helps to perform basic file operations.

Difference is as follows: Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already. The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

Q.6. (a) Explain storage classes available in C. (6.5)

Ans. A storage class defines the scope (visibility) and life time of variables and functions within a C Program.

There are four storage classes which can be used in a C Program

Auto

register

static

extern

Auto Storage Class is the default storage class for all local variables.

```
{
int Month;
//or
auto int Month; //Both declarations are same
}
```

Auto can only be used within functions, i.e. local variables.

Register Storage Class: Register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

Example: register int month;

Static Storage Class is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
{
printf("%d\n", Road);
}
```

Static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

Static can also be defined within a function. If this is done, the variable is initalised at run time but is not reinitalized when the function is called. Thus, inside a function static variable retains its value during vairous calls.

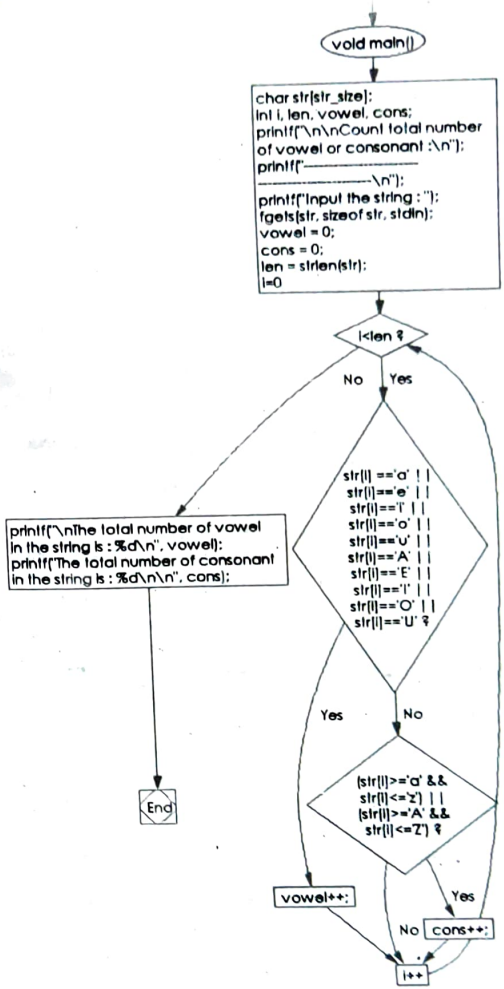
External storage class: External variable can be accessed by any function: They are also known as global variables. Variables declared outside every function are external variables.

Example:

```
main()
{
extern int x;
x=10;
printf("%d",x);
}
int x; // Global variable x;
```

Q.6. (b) Draw a flow chart to count the number of vowels in a string of characters. **(6)**

Ans.



Q.7. (a) Distinguish between user defined function and build in functions. (4.5)

Ans. A function is a block of code that performs a specific task.There are several different types of functions in C. The built-in C functions, like *printf( )* and *scanf( )*. These functions are part of the C programming language and user-defined functions.

Built-in Functions: The system provided these functions and are stored in the library. Therefore it is also called Library Functions. e.g. scanf(), printf(), strcpy, strlwr, strcmp, strlen, strcat etc. To use these functions, you just need to include the appropriate C header files.

C has many built-in functions that you can use in your programs. Some built-in functions are:

main( )	gets( )	strlen( )	strcat( )	isdigit( )
printf( )	puts( )	strcmp( )	strstr( )	isupper( )
scanf( )	strcpy( )	stricmp( )	isalpha( )	islower( )

User-Defined Functions: This is a function which the programmer creates and uses in a C program. A function is a block of code that performs a specific task.C allows you to define functions according to your need. These functions are known as user-defined functions. For example:Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

createCircle() function

color() function

If you have a special set of instructions that aren't in a built-in function, you can create a user-defined function. Here are the steps:

Give function a name that isn't already used in C (by built-in functions, types of variables, keywords, etc.)

create a function header, which contains three things:

(a) the type of variable (int, char, double, etc.) that the function will produce (return)

(b) the name of the function, which can be one or more words (but put underscores _ or Capital Letters connecting these words, because no spaces are allowed)

(c) the parameters of the function, which are the names and types of variables inside your function

Q.7. (b) Write a Program that dynamically allocates an integer, initializes the integer with a value, increment it and then print the incremented value. (4)

Ans.
```
#include<stdio.h>
#include<stdlib.h>
Void main()
{
 int *p;
p=(int*)malloc(1*sizeof(int));
*p=2;
*p++;
Printf("%d",*p);

getch();

}
```


Q.7. (c) Differentiate between sequential and random access files. (4)

Ans. Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.

On the other hand, Random Access to a file means that the computer system can read or write information anywhere in the data file. This type of operation is also called "Direct Access" because the computer system knows where the data is stored (using Indexing) and hence goes "directly" and reads the data.

Sequential access has advantages when you access information in the same order all the time. Also is faster than random access.

Q.8. (a) What is type casting? Explain with example. (6)

Ans. Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows

(type_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation-

```
#include<stdio.h>
main(){
int sum =17, count =5;
double mean;
mean=(double) sum / count;
printf("Value of mean : %f\n", mean );
}
```

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

Value of mean : 3.400000

Here the cast operator has precedence over division, so the value of **sum** is first converted to type double and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Q.8. (b) Explain nested looping with example. (3.5)

Ans. Create code that embeds one loop inside another loop. This is called a nested loop. Any loop type can be nested within another type; the most common is the nesting for loops. Nested for loops are used to cycle through matrix/tabular data and multi-dimensional arrays.

Syntax

The syntax for a nested for loop statement in C is as follows–

```
for ( init; condition; increment ) {
  for ( init; condition; increment ) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a nested while loop statement in C programming language is as follows-

```
while(condition) {

  while(condition) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a nested do...while loop statement in C programming language is as follows-

```
do {
  statement(s);
  do {
    statement(s);
  }while( condition );

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Q.8. (c) Write a C program to search an element in an array. (3)

Ans.

```
#include <stdio.h>
void main()
{
  int a[10], i, item;
  printf("\nEnter SEVEN elements of an array:\n");
  for (i=0; i<=6; i++)
    scanf("%d", &a[i]);
  printf("\nEnter item to search: ");
  scanf("%d", &item);
  for (i=0; i<=9; i++)
    if (item == a[i])
    {
      printf("\nItem found at location %d", i+1);
      break;
    }
  if (i > 9)
    printf("\nItem does not exist.");
  getch();
}
```

OUTPUT:

Enter SEVEN elements of an array: 12 25 45 89 54 68 88

Enter item to search: 89

Item found at location 4