

Diploma of Associate Engineer
CIT Technology 1st Year

Approved by TEVTA Punjab

Text Book of
**INTRODUCTION TO
COMPUTER PROGRAMMING**

CIT-113

Text book of INTRODUCTION TO COMPUTER PROGRAMMING CIT-113

The logo of TEVTA (Technical Education & Vocational Training Authority) Punjab, featuring a circular emblem with a green border. Inside the border, the acronym 'TEVTA' is written in white, with 'GOVERNMENT OF THE PUNJAB' written below it. The central part of the emblem contains four quadrants with icons representing different fields of study.

A screenshot of a computer screen displaying a block of programming code in a dark-themed code editor. The code appears to be in JavaScript or a similar language, showing various functions and conditional statements.

Academics Wing
Technical Education & Vocational Training Authority
PUNJAB

Preface

The text book has been written to cover the syllabus of Introduction to Computer Programming, 1st year D.A.E (CIT) according to the new scheme of studies. Hopefully this book will center the needs of all those students who are preparing themselves in the subject of Introduction to Computer Programming. For D.A.E Examinations of different technical boards throughout Pakistan.

The aim of bringing out this book is to enable the students to have sound knowledge of the subject. Every endeavour has been made to present the subject matter in the most concise, compact lucid & simple manner to help the subject without any difficulty. Frequent use of illustrative figures has been made for clarity.

Questions for self test have also been included at the end of each chapter which will serve as a quick learning tool for students.

I am grateful to the reviewers whose valuable recommendations have made the book user friendly.

Constructive criticisms and suggestions for the improvements in future are welcome.

AUTHORS

Committee Names & Designation

- | | |
|--|-----------------|
| 1. Muhammad Mohsin
GCT, Bahawalpur | Convener |
| 2. Mrs.Nusrat Parveen
GCT (W), Lahore | Member |
| 3. Muhammad Zakriya
GCT, Kamiliya | Member |

Contents

	Page.No
Chapter 1: Introduction with the Computers and Programming Languages	07
Chapter 2: Programming Cycle	30
Chapter 3: The C Integrated Development Environment (IDE)	40
Chapter 4: C Building Blocks	59
Chapter 5: Conditional Control Construct: Decisions	87
Chapter 6: Iterative Control Construct: Loops	115
Chapter 7: Functions	137
Chapter 8: Arrays and Strings	155
Chapter 9: Pointers	180
Chapter 10: Structures and Unions	224
Chapter 11: Files	266
Chapter 12: Larger Programs	297

COURSE OUTLINE

1. Introduction with the Computers and Programming Languages

- 1.1. Discuss computer development
- 1.2. Describe computer generations
- 1.3. Describe electronic data processing
- 1.4. Explain the terms: hardware , software, and peopleware
- 1.5. Explain advantages of EDP
- 1.6. Discuss programming languages
- 1.7. Explain categories of programming languages
- 1.8. Describe machine language
- 1.9. Describe symbolic or assembly language
- 1.10. Describe high-level languages
- 1.11. Explain features of programming languages

2. Programming Cycle

- 2.1. Describe the programming cycle
- 2.2. Explain importance of readability and documentation
- 2.3. Describe Flowcharts
- 2.4. Cite advantages and disadvantages of flowcharting
- 2.5. Draw flowchart of procedures

3. The C Integrated Development Environment (IDE)

- 3.1. Demonstrate setting up the IDE
- 3.2. Explain files used in c program developer
- 3.3. Explain the use of IDE
- 3.4. Explain the structure of c programs

4. C Building Blocks

- 4.1. Define variable
- 4.2. Explain Input/Output
- 4.3. Enumerate operator symbols in C
- 4.4. Explain the use these operators

5. Conditional Control Construct: Decisions

- 5.1. Demonstrate use of the if Statement
- 5.2. Demonstrate use of the if-else Statement
- 5.3. Demonstrate use of the else-if Statement
- 5.4. Demonstrate use of the switch Statement
- 5.5. Write C functions using conditional statements

6. Iterative Control Construct: Loops

- 6.1. Demonstrate use of the for Loop
- 6.2. Demonstrate use of the while Loop
- 6.3. Demonstrate use of the do while Loop
- 6.4. Write C functions using iterations

7. Functions

- 7.1. Explain Functions
- 7.2. Explain Simple Functions
- 7.3. Explain Value-Returning Functions
- 7.4. Perform Parameter Passing
- 7.5. Demonstrate use of Multiple Functions
- 7.6. Describe External Variable
- 7.7. Explain Preprocessor Directives

8. Arrays and Strings

- 8.1. Describe Array Data Types
- 8.2. Demonstrate use of Single and Two-dimensional Arrays
- 8.3. Explain Strings

9. Pointers

- 9.1. Explain Pointers
- 9.2. Describe how to Return Data from Functions
- 9.3. Explain Pointers and Arrays
- 9.4. Explain Pointers and Strings
- 9.5. Demonstrate use of Double Indirection: Pointers to Pointers

10. Structures and Unions

- 10.1. Demonstrate use of Structures
- 10.2. Demonstrate use of Unions
- 10.3. Demonstrate use of Unions of Structures

11. Files

- 11.1. Explain Types of Disk I/O
- 11.2. Explain Standard Input/Output
- 11.3. Explain Binary and Text Mode Files
- 11.4. Program Record Input/Output
- 11.5. Explain Random Access Files
- 11.6. Explain Error Handling in File I/O
- 11.7. Explain Redirection

12. Larger Programs

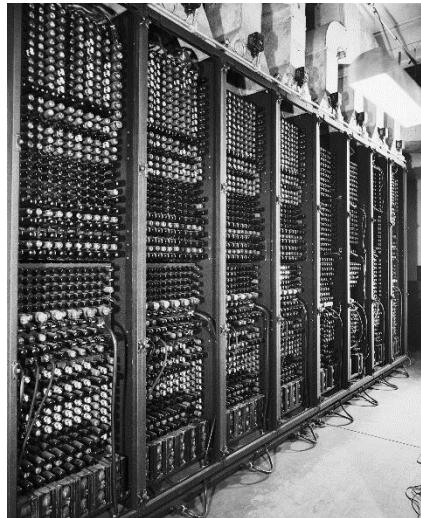
- 12.1. Make Stand-alone Executables
- 12.2. Perform Separate Compilation
- 12.3. Perform Conditional Compilations
- 12.4. Explain Memory Models

INTRODUCTION WITH THE COMPUTERS AND PROGRAMMING LANGUAGES

01

1.1 Computer Development

- The progression in hardware representation of a bit of data:
 - Vacuum Tubes (1950s) - one bit on the size of a thumb;
 - Transistors (1950s and 1960s) - one bit on the size of a Fingernail;
 - Integrated Circuits (1960s and 70s) - thousands of bits on the size of a hand
 - Silicon computer chips (1970s and on) - millions of bits on the size of a finger nail.
 - The progression of the ease of use of computers:
 - Almost impossible to use except by very patient geniuses (1950s);
 - Programmable by highly trained people only (1960s and 1970s);
 - Useable by just about anyone (1980s and on).
1. **The Very First Digital Computer:** The first substantial computer was the ENIAC machine (1946 – 1955) by John W. Mauchly and John. Presper Eckert at the University of Pennsylvania. ENIAC (Electrical Numerical Integrator and Calculator) used,
 - ✓ A word of 10 decimal digits instead of binary ones like previous automated calculators/computers.
 - ✓ First machine to use more than 2,000 vacuum tubes, using nearly 18,000 vacuum tubes.
 - ✓ Took up over 167 square meters (1800 square feet) of floor space.



2. **Progression of Hardware:** In the 1950's two devices would be invented that would improve the computer field and set in motion the beginning of the computer revolution. The first of these two devices was the transistor. Invented in 1947 by William Shockley, John Bardeen, and Walter Brattain of Bell Labs, the transistor replaced vacuum tubes in computers, radios, and other electronics. Vacuum tubes were highly inefficient, required a great deal of space, and needed to be replaced often.

Transistors, however, had their problems too. The main problem was that transistors, like other electronic components, needed to be soldered together. As a result, the more complex the circuits became, the more complicated and numerous the connections between the individual transistors and the likelihood of faulty wiring increased.

In 1958, this problem too was solved by Jack St. Clair Kilby of Texas Instruments. He manufactured the first integrated circuit or chip. A chip is really a collection of tiny transistors which are connected together when the transistor is manufactured. Thus, the need for soldering together large numbers of transistors was practically finished.

3. Mainframes To PCs: The 1960s saw large mainframe computers become much more common in large industries and with the US military and space program. IBM became the unquestioned market leader in selling these large, expensive and very hard to use machines.

- ✓ A explosion of personal computers occurred in the early 1970s, starting with Steve Jobs and Steve Wozniak exhibiting the first Apple II at the First West Coast Computer Faire in San Francisco. The Apple II boasted built-in BASIC programming language, color graphics, and a 4100 character memory. Programs and data could be stored on an everyday audio-cassette recorder.
- ✓ Also introduced in 1977 was the TRS-80. This was a home computer manufactured by Tandy Radio Shack. In its second incarnation, the TRS-80 Model II, came complete with a 64,000 character memory and a disk drive to store programs and data on. At this time, only Apple and TRS had machines with disk drives. With the introduction of the disk drive, personal computer applications took off as a floppy disk was a most convenient publishing medium for distribution of software.
- ✓ IBM, which up to this time had been producing mainframes and minicomputers for medium to large-sized businesses, decided that it had to get into the act and started working on the Acorn, which would later be called the IBM PC. The PC was the first computer designed for the home market which would feature modular design so that pieces could easily be added to the architecture. Most of the components, surprisingly, came from outside of IBM, since building it with IBM parts would have cost too much for the home computer market. When it was introduced, the PC came with a 16,000 character memory, keyboard from an IBM electric typewriter, and a connection for tape cassette player.

- ✓ Apple released the first generation Macintosh, which was the first computer to come with a graphical user interface(GUI) and a mouse. The GUI made the machine much more attractive to home computer users because it was easy to use. Sales of the Macintosh soared like nothing ever seen before.
- ✓ IBM was hot on Apple's tail and released the 286-AT, which with applications like Lotus 1-2-3, a spreadsheet, and Microsoft Word, quickly became the favourite of business concerns.
- ✓ That brings us up to about ten years ago. Now people have their own personal graphics workstations and powerful home computers. The average computer a person might have in their home is more powerful by several orders of magnitude than a machine like ENIAC. The computer revolution has been the fastest growing technology in man's history.

1.2 Computer Generations

The history of the computer goes back several decades however and there are five definable generations of computers.

- ✓ **1940 – 1956: First Generation – Vacuum Tubes**

These early computers used vacuum tubes as circuitry and magnetic drums for memory. As a result they were enormous, literally taking up entire rooms and costing a fortune to run. These were inefficient materials which generated a lot of heat, sucked huge electricity and subsequently generated a lot of heat which caused ongoing breakdowns.

These first generation computers relied on ‘machine language’ (which is the most basic programming language that can be understood by computers). These computers were limited to solving one problem at a time. Input was based on punched cards and paper tape. Output came out on print-outs. The two notable machines of this era were the UNIVAC and ENIAC machines – the UNIVAC is the first every commercial computer which was purchased in 1951 by a business –



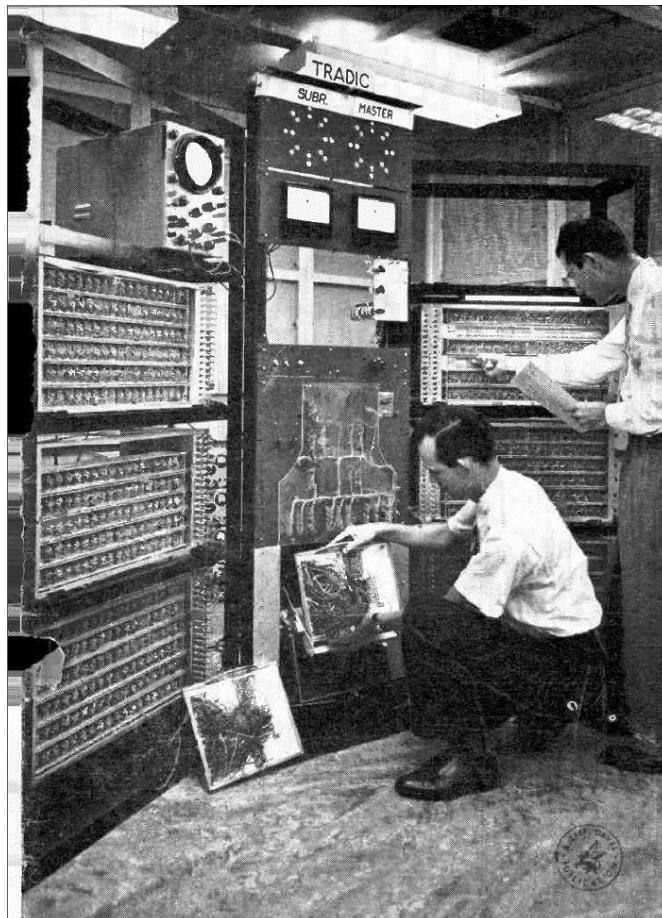
the US Census Bureau.

✓ **1956 – 1963: Second Generation – Transistors**

The replacement of vacuum tubes by transistors saw the advent of the second generation of computing. Although first invented in 1947, transistors weren't used significantly in computers until the end of the 1950s. They were a big improvement over the vacuum tube, despite still subjecting computers to damaging levels of heat. However they were hugely superior to the vacuum tubes, making computers smaller,

faster, cheaper and less heavy on electricity use. They still relied on punched card for input/printouts.

The language evolved from cryptic binary language to symbolic ('assembly') languages. This meant programmer could create instructions in words. About the same time high level programming languages were being developed (early versions of COBOL and FORTRAN). Transistor-driven machines were the first computers to store instructions into their memories – moving from magnetic drum to magnetic core 'technology'. The early versions of these machines were developed for the atomic energy industry.



✓ **1964 – 1971: Third Generation – Integrated Circuits**

By this phase, transistors were now being miniaturized and put on silicon chips (called semiconductors). This led to a massive increase in speed and efficiency of these machines. These were the first computers where users interacted using keyboards and monitors which interfaced with an operating system, a significant leap up from the punch cards and printouts. This enabled these machines to run several applications at once using a central program which functioned to monitor memory. As a result of these advances which again made machines cheaper and smaller, a new mass market of users emerged during the '60s.



✓ **1972 – 2010: Fourth Generation – Microprocessors**

This revolution can be summed in one word: Intel. The chip-maker developed the Intel 4004 chip in 1971, which positioned all computer components (CPU, memory, input/output controls) onto a single chip. What filled a room in the 1940s now fit in the palm of the hand. The Intel chip housed thousands of integrated circuits. The year 1981 saw the first ever computer (IBM) specifically designed for home use and 1984 saw the Macintosh introduced by Apple. Microprocessors even moved beyond the realm of computers and into an increasing number of everyday products. The increased power of these small computers meant they could be linked, creating networks. Which ultimately led to

the development, birth and rapid evolution of the Internet. Other major advances during this period have been the Graphical user interface (GUI), the mouse and more recently the astounding advances in laptop capability and hand-held devices.

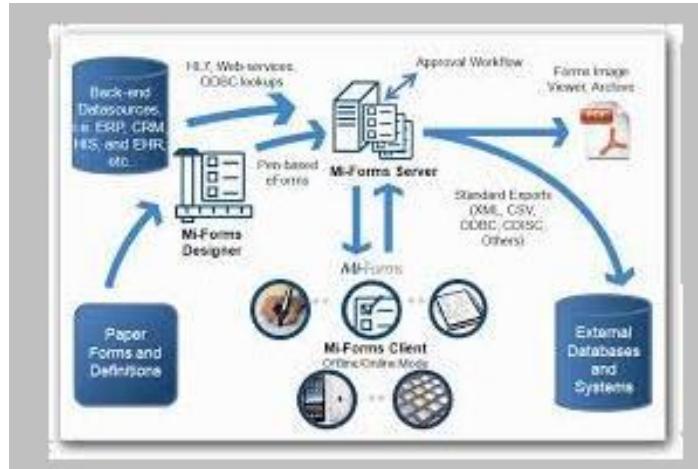
✓ **2010- : Fifth Generation – Artificial Intelligence**

Computer devices with artificial intelligence are still in development, but some of these technologies are beginning to emerge and be used such as voice recognition. AI is a reality made possible by using parallel processing and superconductors. Leaning to the future, computers will be radically transformed again by quantum computation, molecular and nano technology.

The essence of fifth generation will be using these technologies to ultimately create machines which can process and respond to natural language, and have capability to learn and organize themselves.

1.3 Electronic Data Processing (EDP)

Electronic data processing is a frequently used term for automatic information processing. It uses the computers to manipulate, record, classification and to summarize data. A computer is the best example of an electronic data processing machine. Electronic data processing is an accurate and rapid method.



Methods of Electronic Data Processing

- Time-sharing
- Real-time processing
- Online processing
- Multiprocessing
- Multitasking
- Interactive processing
- Batch processing
- Distributed processing

- ✓ **Time- Sharing:** In this processing method, many nodes connected to a CPU accessed central computer. A multi-user processing system controls the time allocation to each user. Each user can allocate the time slice in a sequence of the Central Processing Unit. The user should complete the task during the assigned time slice. If the user cannot finish the task, then the user can complete the task during another allocated time slice.

- ✓ **Real-Time Processing:** Providing accurate and up-to-date information is the primary aim of real-time processing. It is possible when the computer processes the incoming data. It will give the immediate response to what may happen. It would affect the upcoming events. Making a reservation for train and airline seats are the best example for real-time processing. If the seats are reserved, then the reservation system updates the reservation file. The real-time processing is almost an immediate process to get the output of the information. This method saves the maximum time for getting output.
- ✓ **Online Processing:** In this processing method, the data is processed instantly it is received. A communication link helps to connect the computer to the data input unit directly. The data input may include a network terminal or online input device. Online processing is mostly used for information research and recording.
- ✓ **Multi-Processing:** Multiprocessing is processing of more than one task that uses the different processors at the same time of the same computer. It is possible in network servers and mainframes. In this process, a computer may consist of more than one independent CPU. There is a possibility to make coordination in a multiprocessing system. In this process, the different processors share the same memory. The processor gets the information from a different part of one program or various programs.
- ✓ **Multitasking:** It is an essential feature of data processing. Working with different processors at the same time is called multitasking. In this process, the various tasks share the same processing resource. The operating systems in the multitasking process are time-sharing systems.

✓ **Interactive Processing:** This method includes three types of functions. The following are the types of function

- Peak detection
- Integration
- Quantitation

It is a simple way to work with the computer. This method of the process can compete for each other.

- ✓ **Batch Processing:** Batch processing is a method of the process the organized data into divided groups. In this method, the processing data can be divided as a group over a required time period. The batch processing method allows the computer to perform different priorities for an interaction. This method is very unique and useful to process.
- ✓ **Distributed Processing:** This method is usually used for remote workstations, since the remote workstations are connected to a big workstation. The customers get the better services from this process. In this process, the firms can distribute the use of geographical computers. The best example for this distributed processing method is ATMs. ATMs are connected to the banking system.

Elements of Electronic Data Processing

Hardware, Software, procedure, personnel is the basic elements of electronic data processing. In the hardware section, scanners, barcode scanners, cash registers, personal computers, medical device, servers, video and audio equipment are the elements of electronic data processing. In the software section, accounting software, data entry, scheduling software, analytics, and software are the elements of **electronic data processing**. In the procedure section, sorting, analysis, reporting, conversion, data collection, aggregation be the elements of EDP.

In personnel, the programmer uses the electronic data processing to create the components and spreadsheets. The data entry specialists use to scan the barcodes.

Stages of Electronic Data Processing

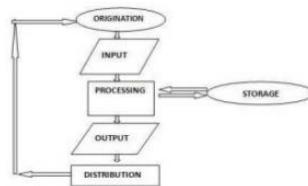
A collection is the first stage of **electronic data processing**. It is a very crucial part. This process ensures that accurate data gathering. Census, sample survey, and administrative by-product are some types of data collection. Preparation is the second stage of electronic data processing. Preparation is used to analyze the data processing.

Input is the third stage in **electronic data processing**. Data entry is done by the use of a scanner, keyboard, and digitizer. The fourth stage is processing. It has various methods. The last stage is storage. Every computer has the use to store the file.

Electronic Data Processing

Electronic Data Processing (EDP) can refer to the use of automated methods to process commercial data. Typically, this uses relatively simple, repetitive activities to process large volumes of similar information.

For example: stock updates applied to an inventory, banking transactions applied to account and customer master files, booking and ticketing transactions to an airline's reservation system, billing for utility services.



1.4 Hardware , Software, and Peopleware

- ✓ *Hardware:* Computer hardware is the collection of physical parts of a computer system. This includes the computer case, monitor, keyboard, and mouse etc. It also includes all the parts inside the system block, such as the hard disk drive, motherboard, video card, and many others. Computer hardware is what you can physically touch.
- ✓ *Software:* Computer software, or simply software, is a collection of data or computer instructions that tells the computer how to work. It is a program that enables a computer to perform a specific task.
- ✓ **Peopleware:** Refers to the role people play in technology and the development of hardware or software. It can include various aspects of the process such as human interaction, programming, productivity, teamwork, and project management.

1.5 Advantages of Electronic Data Processing

- The system of electronic data processing is once created then the cost of the managing data will be reduced. Documents can be protected as an extreme data sensitive. Because of the documents should be treated as a primary asset. When all the information is collected by the papers are the challenging one.
- The management of document is costly. So, electronic data processing reduces the cost of the paperwork. The electronic data processing provides the documentation controls. With the help of electronic data processing, you can easily automate the PDF publishing process.
- In electronic data processing, there is a facility to search a document in the system. It will reduce the time loss. The electronic data

processing has the benefit to improve the internal and external collaboration. The electronic data processing helps to improve the better submissions. The electronic data processing also fast up the complete structure to make the generation of documents.

- The famous software product such as Ms. Office is using the electronic data processing concept. The EDP has the facility to reduce the duplication of effort and repeated entries. The EDP has the capability to make the decisions. An electronic data processing has the ability to store the enormous amounts of data.

Disadvantages of Electronic Data Processing

- When the computer hackers make the strike on the computer, then the processing of data will make the insecurity. Then the data will be the loss. The fault in a equipment will harm all the equipment in the office. The security of the computer would be the big problem. In a coding process, a computer not recognizes the same individuals.
- When a small number of digit codes are compared with a large number code then, it occupies the computer storage less. The alphabetic codes can be descriptive.

Examples of EDP: It is used in a telecom company to format bills and to calculate the usage-based charges. In schools, they use EDP to maintain student records. In supermarkets, used for recording whereas hospitals use it to monitor the progress of patients.

Further, the **electronic data processing** is used for hotel reservations. It can be used in learning institutions. The EDP is also used in banks to monitor the transactions. In the departments such as police, cybercrime, and chemical the electronic data processing is used to note the entries. It enables larger

organizations to collect the information and process the data. The electronic data processing can also be used as video and audio equipment. It can be used as a barcode scanner.

1.6. Programming languages

Computer Program: is a set of instructions written in a sequence by a programmer to complete a specific task. Programmer can use a variety of programming languages to write a program.

Programming Language: A *programming language* is a vocabulary and set of grammatical rules for instructing a *computer* or computing device to perform specific tasks. The term *programming language* usually refers to high-level *languages*, such as BASIC, C, C++, COBOL, Java, FORTRAN and Pascal.

1.7 Categories of Programming Languages

There are three main categories of programming languages,

- ✓ Low Level Languages
- ✓ Intermediate Level Languages
- ✓ High Level Languages

Low level languages: are used to write programs that relate to the specific architecture and hardware of a particular type of computer. These are directly understandable instructions by the machine for which these are written, without any translator.

Intermediate level languages: is an abstract *programming language* used by a translator as an in-between step when translating a *computer* program into machine code. Assembly language is the example of Intermediate language.

These languages are machine specific as low level languages but need translator to translate their instructions into machine code like high level languages.

High level languages: are written in a form that is close to our human language, enabling the programmer to just focus on the problem being solved. These are not understandable instructions for any computerized machine without any translator.

1.8 Machine Language

It is a computer programming language consisting of binary or hexadecimal instructions which a computer can respond/understand directly without any translation.

Sometimes machine language is referred to as machine code or object code. Machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding. The instructions written in this language are machine specific, so the program written for one machine cannot be run on other machine.

1.9 Symbolic or Assembly Language

An assembly language implements a symbolic representation of the machine code needed to program a given CPU architecture. Assembly language is also known as assembly code or symbolic language. It is an intermediate level language. An assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations that are implemented directly on the physical processor. They have the same structures and set of commands as machine

language, but allow a programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages. Assembler is used to translate assembly symbols into machine codes. Like machine language it is also machine specific.

1.10 High-level Languages

High-level programming languages mean that languages of writing computer instructions in a way that is easily understandable and close to human language. High-level languages are created by developers so that programmers don't need to know highly difficult low level/machine language. Programmers can easily learn high-level languages as compared to low level languages. However these are not understandable directly by the machine and have to translate into machine codes for execution through a translator. These languages are not machine specific, so the program written on one machine can be run on another machine. Programs written in these languages can be short and easy to understand.

1.11 Features of Programming Languages

A programming language is an artificial language used to create programs that express precise algorithms to make a computer perform computations. Programming languages allow the manipulation of data structures and the flow of execution of a program. There are several different kinds of programming languages, which differ in many aspects, the most important of them being the computations they are capable of, also known as

the **expressive power** of a programming language. Each programming language provides a basic set of elements, which describes data and the processes and transformations which can be applied to them, also called **primitives** of that language.

A very important element of programming languages is their syntax. Most programming languages are textual and their syntax includes words, numbers, and punctuations. However, there are other programming languages that make use of a graphical approach, where programs are created by a visual representation. Following are the main characteristics of programming languages,

- ✓ **Syntax and Structure:** Programming languages for commands can overlap just like when using words in spoken languages. To produce text to screen in C or VB, you are to use printf or print commands, similar to using imprimer and imprimir when we want to print in French and Spanish.
- ✓ **Functionality of Languages:** All these languages can make the same functionality, similar to how all spoken languages can reflect the same phrases, objects, and emotions.
- ✓ **Written In English:** As opposed to spoken languages (except English), nearly all languages are written in English. This is true whether one is programming JavaScript, Ruby, HTML or C++ etc, they all use English syntax codes and keywords even if the programmers who create them are French or Chinese.
- ✓ **One Programmer.** As opposed to spoken languages, programming languages can be made by only one programmer or a single creator. Languages that can be created by only one creator

include JavaScript (Brendan Eich), Ruby (Yukihiro Matsumoto), and Python (Guido van Rossum).

END CHAPTER 01

Multiple Choice Questions with Answers

1. UNIVAC is

- a) Universal Automatic Computer
- b) Universal Array Computer
- c) Unique Automatic Computer
- d) Unvalued Automatic Computer

2. VGA is

- a) Video Graphics Array
- b) Visual Graphics Array
- c) Volatile Graphics Array
- d) Video Graphics Adapter

3. BCD is

- a. Binary coded Decimal
- b. Bit Coded Decimal
- c. Binary Coded Digit
- d. Bit Coded Digit

4. Chief Component of first generation Computer was

- a. Transistors
- b. Vacuum Tubes
- c. Integrated Circuit
- d. None of the above.

5. Second Generation computers was developed during

- a. 1949 to 1955

- b. 1956 to 1965
- c. 1965 to 1970
- d. 1970 to 1990

6. The Computer size was very large in

- a). First Generation
- b) Second Generation
- c) third Generation
- d) Fourth Generation

7. Which Generation is still under development

- a) Fourth Generation
- b) Fifth Generation
- c) Sixth Generation
- d) seventh generation

8. The language that computer can understand and execute is called

- a) Machine Language
- b) Application Software
- c) System Program
- d) All of the above

9. Who designed the first electronic computer-ENIAC?

- a) Von Neumann
- B) Joseph M Jacquard
- c) J.P Eckert and J.W . Archly

d) All of the above.

10. Modern Computer are very reliable but they are not

- a) Fast
- b) Powerful
- c) Infallible
- d) Cheap

11. A Computer program that converts an entire program into machine at one time is called a/an

- a) Interpreter
- b) CPU
- c) Simulator
- d) Compiler

12. A computer program that translates one instruction at a time into machine language is called a/an

- a) Interpreter
- b) CPU
- c) Simulator
- d) Compiler

13. Abacus is known to be the first_____ calculating device

- a) Electrical
- b) Mechanical
- c) physical
- d) all of the above

14. Who invented the microprocessor.

- a) Mercian E Huff
- b) Herman H Goldstein
- c) Joseph Jacquard
- d) All of the above

15. Who developed a mechanical device in the 17th Century what could add, subtract, multiple, divide and find square roots?

- a) Napier
- b) Babbage
- c) Pascal
- d) Leibniz

16. Analytical Engine was invented in

- a) 1833
- b) 1843
- c) 1853
- d) None of the above

17. Computer is an electronic device for perform

- a) Arithmetic operation
- b) Logical Operation
- c) Both a and b
- d) None of the above

18. In the year 1642, a French scientist invented an adding machine is called

- a) Napier Bones
- b) Abacus
- c) Pascal calculator
- d) Leibniz calculator

19. There are _____ categories of programming languages.

- a) Two
- b) Three
- c) Four
- d) one

20. The first commercial computer, the Lyons Electronic office(LEO 1), was developed in 1951 by_____

- a) Mercian E Huff
- b) Joseph Jacquard
- c) Joe Lyons
- d) None of the above

Answers

1. a	2. a	3. a	4. b	5. b
6. a	7. b	8. a	9. c	10. c
11. d	12. a	13. b	14. a	15. d
16. a	17. a & b	18. c	19. b	20. c

Questions with Short Answers

Q 1. What is computer program?

Ans. The set of instructions given to the computer to solve a specific problem is called computer program. Computer can solve problems with the help of computer program. Computer programs are written in programming languages.

Q 2. What is programming language?

Ans. Programming language is used to communicate with computer. All computer programs are written in programming languages. Every programming language has a set of alphabets and rules. The instructions of computer program are written by using the alphabets and rules defined by the programming language.

Q 3. List different types of programming language.

Ans. There are two types of programming languages.

Low level languages

High level languages

Q 4. What is high level language?

Ans. A programming language that is close to human language is called high level language. The instructions written in high level language look like English language sentences. High level languages are easy to learn and understand.

Q 5. What is low level language?

Ans. A language that is close to the language of computer. Computer itself uses this language is called a low level language. There are two types of low level languages.

Machine language

Assembly language

Q 6. What is machine language?

Ans. Machine language is also called binary language. There are only two alphabets of machine language those are zero (0) and one (1). Computer can understand only machine language. The programs written in other programming languages are first translated to machine language, and then used on computer. It is called the native language of computer.

Q 7. What is assembly language?

Ans. It is a programming language in which machine language instructions are replaced by English like words. These words are known as mnemonics. It is pronounced as Ne-Monic. An assembly language used English like words. It is easy to write a program in assembly language. It is mostly used for

system software.

Q. 8 List name of some high level language?

Ans. High level languages

C

C++

C#

COBOL

BASIC

FORTRAN

PASCAL

JAVA

Q.9 What is EDP?

Electronic data processing is a frequently used term for automatic information processing. It uses the computers to manipulate, record, classification and to summarize data. A computer is the best example of an electronic data processing machine. Electronic data processing is an accurate and rapid method.

Q.10. Who is programmer?

A Person who designs, writes, and tests computer program is called Programmer.

Long Questions

1. Define Computer Development?
2. Describe Electronic Data Processing in detail?
3. Describe Features of Programming Languages in detail?
4. Describe Categories of Programming Languages in detail?
5. Describe the Advantages of EDP?

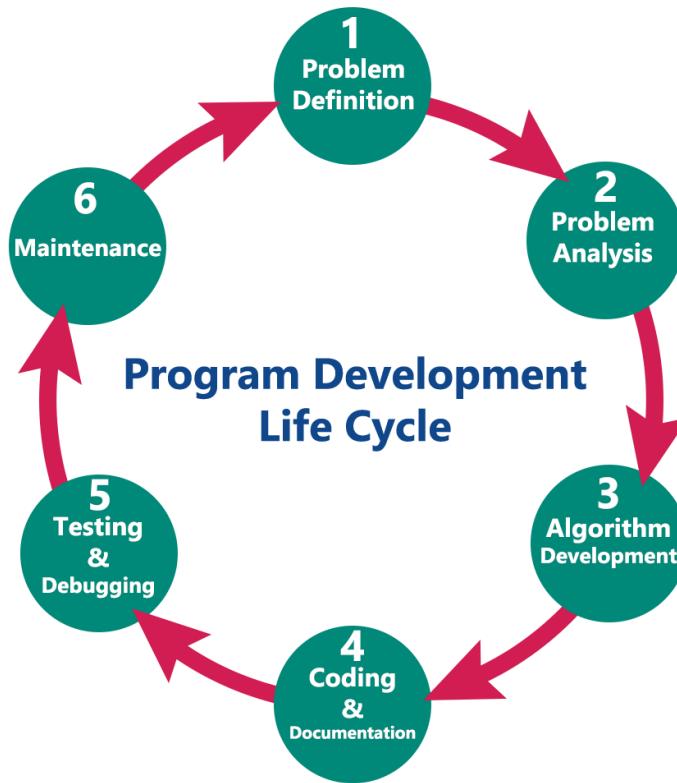
PROGRAMMING CYCLE

02

2.1 Programming Cycle

To develop a program using any programming language, sequence of steps should have to follow. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language. Generally, the program development life cycle contains 6 phases, they are as follows,

- 1) Problem Definition
- 2) Problem Analysis
- 3) Algorithm Development
- 4) Coding & Documentation
- 5) Testing & Debugging
- 6) Maintenance



✓ Problem Definition

In this phase, the problem statement is defined and decides the boundaries of the problem. This phase is needed to understand the problem statement, what requirements are and what should be the output of the problem solution. These are defined in this first phase of the program development life cycle.

✓ Problem Analysis

In this phase, programmer determines the requirements like variables, functions, etc. to solve the problem. That means programmer gather the required resources to solve the problem defined in the problem definition phase. The bounds of the solution are also determined.

✓ Algorithm Development

During this phase, programmer develops a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means to write the solution in step by step statements.

✓ Coding & Documentation

This phase uses a programming language to write or implement the actual programming instructions for the steps defined in the previous phase. In this phase, programmer constructs the actual program. That means programmer writes the program to solve the given problem using programming languages like C, C++, Java, etc.

✓ Testing & Debugging

During this phase, programmer checks whether the code written in the previous step is solving the specified problem or not. It means to test the program whether it is solving the problem for various input data values or not. Programmer also tests whether it is providing the desired output or not.

✓ Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated to make the enhancements. That means in this phase, the solution (program) is used by

the end-user. If the user encounters any problem or wants any enhancement, then programmer need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

2.2 Importance of Readability and Documentation

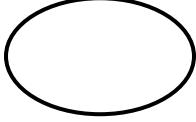
- ✓ **Readability:** Source code is designed for us. It may end up being processed by a machine, but it evolves in our hands and we need to understand what the code does and where changes need to be made. We may understand our code now, but what about six months or a year from now? Readable code helps us to remind and understand that what we wrote and why we wrote it. If we are working in a team to develop some code then readable source code can ensure that everyone can understand the code written by everyone else. Formatting also make impact on readability. The formatting or appearance of code determines how quickly and easily the reader can understand what it does. The careful selection of names for variables, functions etc. are very important to understand the source code. Since the source code tells the reader what the code does and comments improve readability. It allows us to provide the reader with additional information. The reader should be able to understand a single function or method from its code and its comments, and should not have to look elsewhere in the code for clarification.
- ✓ **Documentation:** Documentation is written in a place where people who need to use the software can read about how to use the software. 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. For programmer reliable documentation is always must. The presence of documentation helps keep track of all aspects of an application and it

improves on the quality of a software product. Its main focuses are development, maintenance and knowledge transfer to other developers

2.3 Flowcharts

Flowcharts are used in designing and documenting simple processes or programs. A flowchart is simply a graphical representation of steps. It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes. Typically, a flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows. It was originated from computer science as a tool for representing algorithms and programming logic but had extended to use in all other kinds of processes. Nowadays, flowcharts play an extremely important role in displaying information and assisting reasoning.

Flow Chart Symbols: The American National Standards Institute (ANSI) set standards for flowcharts and their symbols in the 1960s. The International Organization for Standardization (ISO) adopted the ANSI symbols in 1970. Generally, flowcharts flow from top to bottom and left to right. Following table shows the most common symbols used in programming.

SYMBOL	NAME	FUNCTION
	Terminator	The terminator symbol represents the starting or ending point of the system.



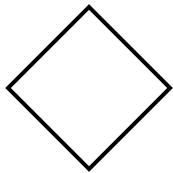
Process

A box indicates some particular operation.



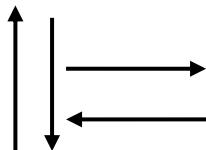
Data

It represents information regarding Inputs or outputs



Decision

A diamond represents a decision or branching point.



Arrows

Lines represent flow of the sequence and direction of a process.

2.4 Advantages and Disadvantages of flowcharting

ADVANTAGES

Communication

Flowcharts are better way of communicating the logic of a system to all concerned or involved.

Effective analysis

With the help of flowchart, problem can be analyzed in more effective way therefore reducing cost and wastage of time.

Proper Documentation	Program flowcharts serve as a good program documentation, which is needed for various purposes, making things more efficient.
Efficient Coding	The flowcharts act as a guide during the systems analysis and program development phase.
Proper Debugging	The flowchart helps in debugging process.
Efficient Program Maintenance	The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

DISADVANTAGES

Complex Logic	Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex. This will become a pain for the user, resulting in a waste of time and money trying to correct the problem.
Alterations and Modifications	If alterations are required the flowchart may require re-drawing completely. This will usually waste valuable time.

END OF CHAPTER 02

Multiple Choice Question with Answers

- 1. A flow chart is a _____ representation of a process.**
a) Graphical b) Instrumental c) Linear d) None of the Above
- 2. The flow chart symbols are linked together with _____ showing the process flow direction.**
a) Lines b) Arrows c) Rectangular d) None of the above
- 3. Flowcharts symbols are also known as _____**
a) Flow direction b) Both c and d c) Business process Map Symbol
d) Flowcharts shape.
- 4. The symbol in figure 1 is known as**
a) Decision b) Preparation delay c) Process d) Alternate process
- 5. The Symbol in figure 2 is known as**
a) Decision b) Data I/O c) Terminator d) Magnetic disk
- 6. The symbol in figure 3 is known as**
a) Magnetic disk b) Stored data c) Direct Storage access d) Flow line
- 7. The symbol in figure 4 is known as**
a) Terminator b) Stored data c) Decision d) flow line
- 8. The symbol in figure 5 is known as**
a) Decision b) Terminator c) Internal Storage d) None of these
- 9. The symbol in figure 6 is known as**
a) Flow path b) Flow direction c) Flow destination d) Flow line
- 10. The Symbol in figure 7 is known as**
a) Automatic input b) Manual input c) Stored input d) page-off connector.
- 11. The Symbol in figure 8 is known as**
a) Connector b) page-off connector c) Page-in-connector d)
Terminator
- 12. The Symbol in figure 9 is known as**
a) card b) Punched type c) Storage data d) None of the above
- 13. The Symbol in figure 10 is known as**
a) Punched type b) Card c) Data Storage d) All of the above
- 14. The Symbol in figure 11 is known as:**
a) Internal storage b) Magnetic Disk c) Both a and b d) stored data
- 15. The Symbol in figure 12 is known as**
a) Multi document b) Document c) Terminator d) None
- 16. The most universally recognizable symbol for a data storage**
location, this flow chart shape depicts a database
a) stored data b) Magnetic disk c) Internal Storage d)None

17. A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer is called

- a) Flow chart
- b) Algorithm
- c) Both a and b
- d) None

18. The basic steps involved in a program development cycle are

- a) Three
- b) seven
- c) six
- d) two

19. The process of isolating and removing errors(bugs) form a program is called

- a) Debugging
- b) Testing
- c) Both a and b
- d) None

20. The series of steps involved in the process of program development collectively is known as;

- a) Programming routine
- b) programming schedule
- c) programming timing
- d) programming cycle.

Answers

1. a	2. b	3. b	4. c	5. b
6. a	7. c	8. b	9. d	10. b
11. a	12. a	13. a	14. d	15. b
16. b	17. b	18. c	19. a	20. d

SHORT QUESTIONS ANSWERS

Q. 1 Define Programming Cycle?.

To develop a program using any programming language, sequence of steps should have to follow. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language.

Q. 2 What is Flow Chart?

Flowcharts are used in designing and documenting simple processes or programs. A flowchart is simply a graphical representation of steps. It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes.

Q.3 What is Readability?

Readable code helps us to remind and understand that what we wrote and why we wrote it. If we are working in a team to develop some code then readable source code can ensure that everyone can understand the code written by everyone else. Formatting also make impact on readability.

Q.4 What is Documentation?

Documentation is written in a place where people who need to use the software can read about how to use the software. 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.

Q.5 What is mean by Testing & Debugging?

During this phase, programmer checks whether the code written in the previous step is solving the specified problem or not. It means to test the program whether it is solving the problem for various input data values or not. Programmer also tests whether it is providing the desired output or not.

Q.6 How many steps are there in program development cycle?

There are six steps in program development cycle.

1. Analyses/Define the problem
2. Design a solution
3. Code the program
4. Test and debug the program
5. Documentation
6. Implementation and maintenance

Q.7 Explain any three advantages of flowchart?

Communication: Flowcharts are better way of communicating the logic of a system to all concerned.

Effective analysis: with the help of flowchart, problem can be analyzed in more effective way.

Proper documentation: Program flowcharts serve as a good program documentation, which is needed for various purposes.

Q.8 Explain any three disadvantages of flowchart?

Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.

Alterations and Modifications: If alterations are required the flowchart may require-re-drawing completely.

Reproduction: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

Q.9 Define the working magnetic disk symbol in flowchart?

The most universally recognizable symbol for a data storage location, this flowchart shape depicts a database.

Q.10 Why we use the word cycle in programming cycle?

The word “Cycle” is used because in case we cannot achieve the results on testing the problem. We may have to go back to one or all of these steps again to find the errors and correct them.

Long Questions

Q.1 What is Programming cycle?

Q.2 What is Algorithm?

Q.3 Explain Common Flowcharts symbols

Q.4 What is the importance of Readability and Documentation

Q.5 What is mean by Debugging and Testing.

THE C INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

03

Computer understands only binary instructions. Since instructions written in high level languages are close to human language, therefore these should have to translate in binary instructions before execution. Different software applications are used as Translator to translate high level instructions into binary codes. These translators could be of following three major types or mixed.

Compiler: A compiler is a language translator that converts high level programs into machine understandable machine codes. In this process, the compiler converts the whole program to machine code at a time. If there are any syntactic or semantic errors, the compiler will indicate them. It checks the whole program and displays all errors. It is not possible to execute the program without fixing those errors.

Interpreter: An interpreter is also a language translator that converts high level programs into machine codes. Unlike compilers, interpreters convert the source code to machine code line by line. As it checks line by line, the scanning time is lower. But the overall execution time is higher. Interpreter displays an error at a time. The programmer should fix that error to interpret the next line. Programming languages such as Python, Ruby, PHP, Perl are some examples of interpreter-based languages.

Assembler: In addition to high level languages and machine language, there is another language called the assembly language. Assembly language is

in between the high level languages and machine language. It is closer to machine language than high level languages. It is also called intermediate level language. This language is not easily readable and understandable by the programmer like a high level programming language. The assembler works as the translator in converting the assembly language program to machine code.

COMPILER VS INTERPRETER VS ASSEMBLER

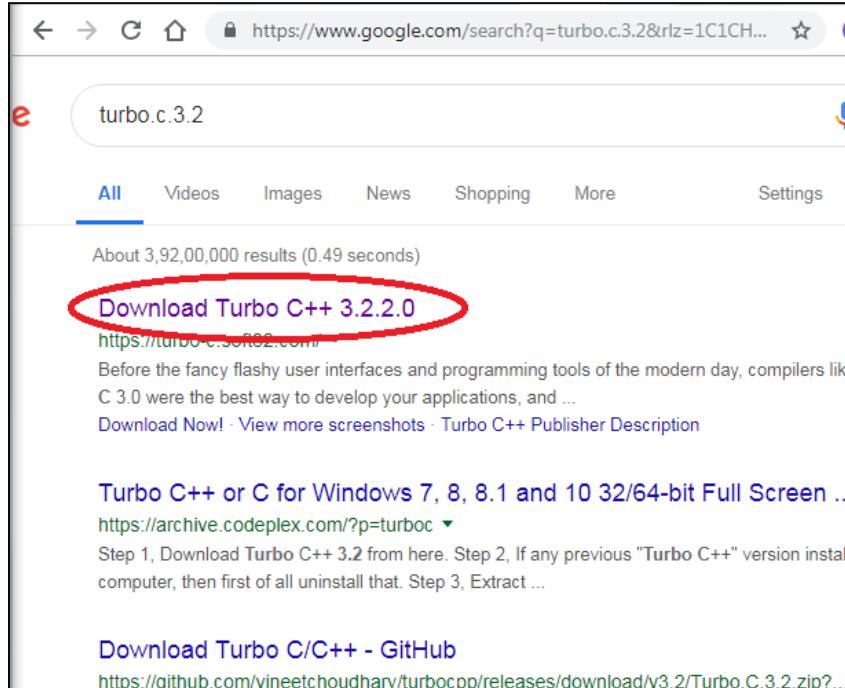
Software that converts programs written in a high level language into machine language	Software that translates a high level language program into machine language	Software that converts programs written in assembly language into machine language
Converts the whole high level language program to machine language at a time	Converts the high level language program to machine language line by line	Converts assembly language program to machine language
Used by C, C++	Used by Ruby, Perl, Python, PHP	Used by assembly language Visit www.pediaa.com

3.1 Setting up the IDE

Different C compilers are available in market. The following steps show how one can download, install and use Turbo C++ compiler.

✓ Step-1: Download Turbo C++ software

Firstly, download Turbo C++ from the link: Turbo.C.3.2. using Google. When you will open this link following page will appear. Open the first link to download it.



turbo.c.3.2

All Videos Images News Shopping More Settings

About 3,92,00,000 results (0.49 seconds)

Download Turbo C++ 3.2.2.0

<https://turbo-c.soft32.com/>

Before the fancy flashy user interfaces and programming tools of the modern day, compilers like C 3.0 were the best way to develop your applications, and ...

[Download Now!](#) · [View more screenshots](#) · [Turbo C++ Publisher Description](#)

Turbo C++ or C for Windows 7, 8, 8.1 and 10 32/64-bit Full Screen ..

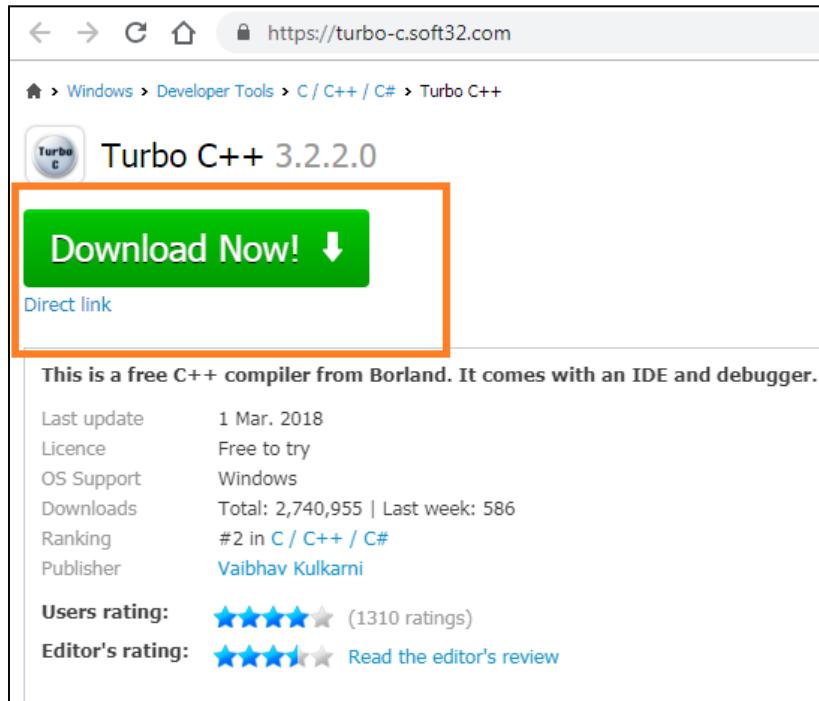
<https://archive.codeplex.com/?p=turboc>

Step 1, Download Turbo C++ 3.2 from here. Step 2, If any previous "Turbo C++" version installed on your computer, then first of all uninstall that. Step 3, Extract ...

Download Turbo C/C++ - GitHub

<https://github.com/vineetchoudhary/turbocpp/releases/download/v3.2/Turbo.C.3.2.zip?>

When you will open the “open link”, the option to download turbo C++ will appear. Click and download it.



<https://turbo-c.soft32.com>

Windows > Developer Tools > C / C++ / C# > Turbo C++

Turbo C++ 3.2.2.0

Download Now! ↓

[Direct link](#)

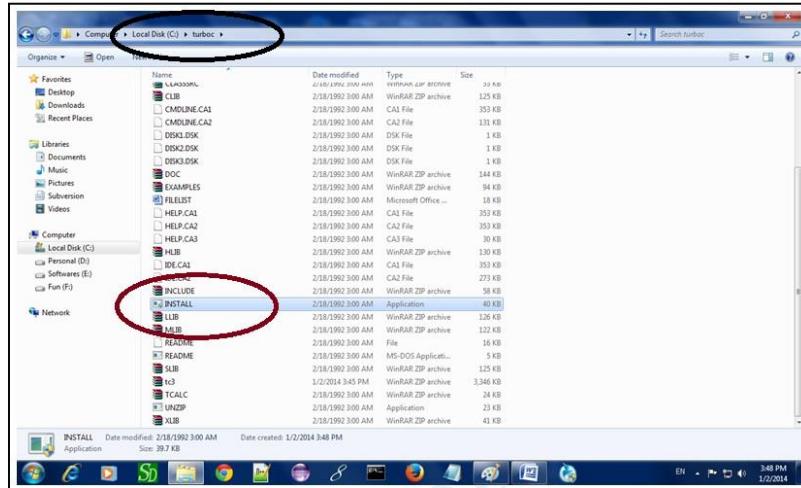
This is a free C++ compiler from Borland. It comes with an IDE and debugger.

Last update: 1 Mar. 2018
Licence: Free to try
OS Support: Windows
Downloads: Total: 2,740,955 | Last week: 586
Ranking: #2 in C / C++ / C#
Publisher: Vaibhav Kulkarni

Users rating: ★★★★☆ (1310 ratings)
Editor's rating: ★★★★☆ [Read the editor's review](#)

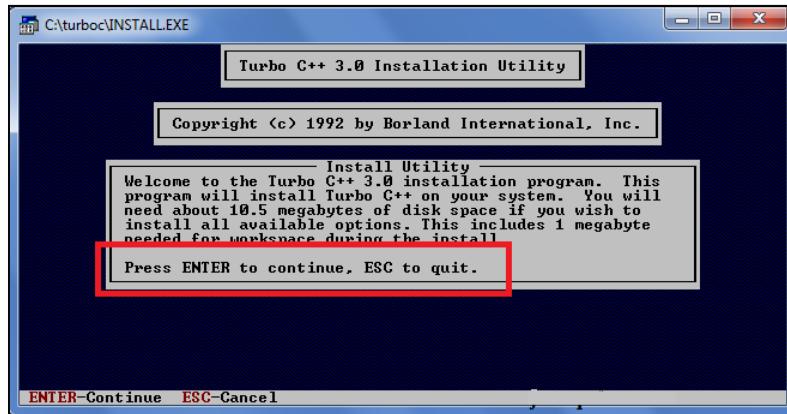
✓ Step-2: Create turbo c directory in c drive and extract tc3.zip

Now, you must create turbo c directory inside c:\ drive and extract the zip file in this directory.



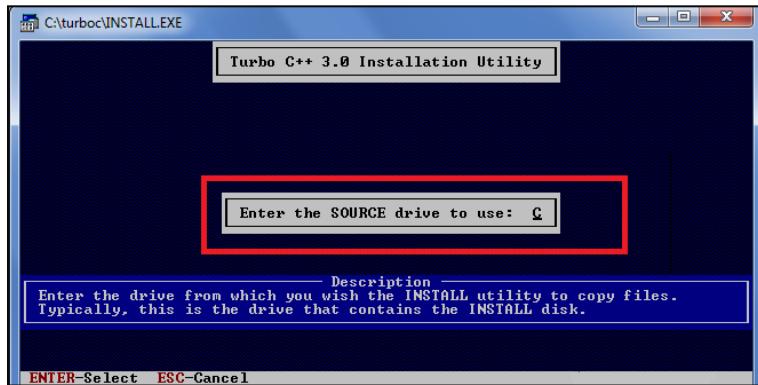
✓ Step-3: Permission to install C

Now a window will appear asking for permission to install or not, press enter to install C.



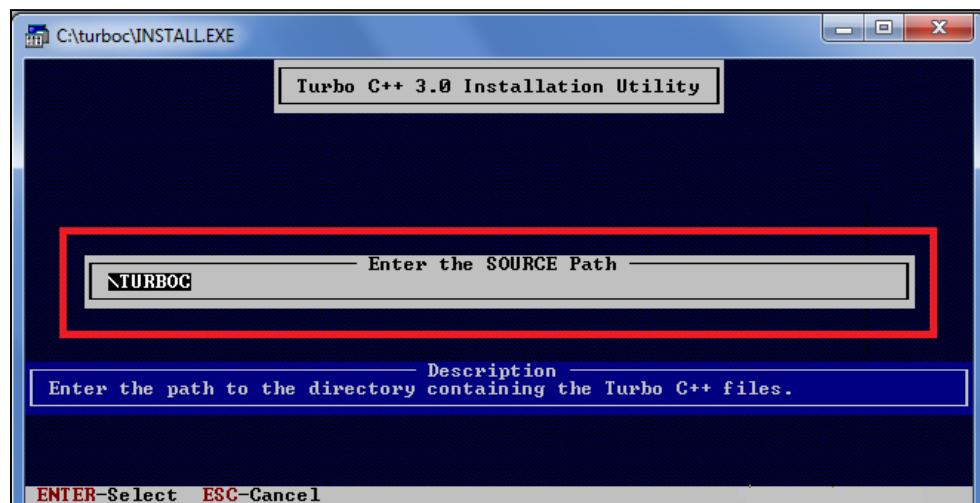
✓ Step-4: Change drive to C

After pressing enter a window will appear. Change the drive to C



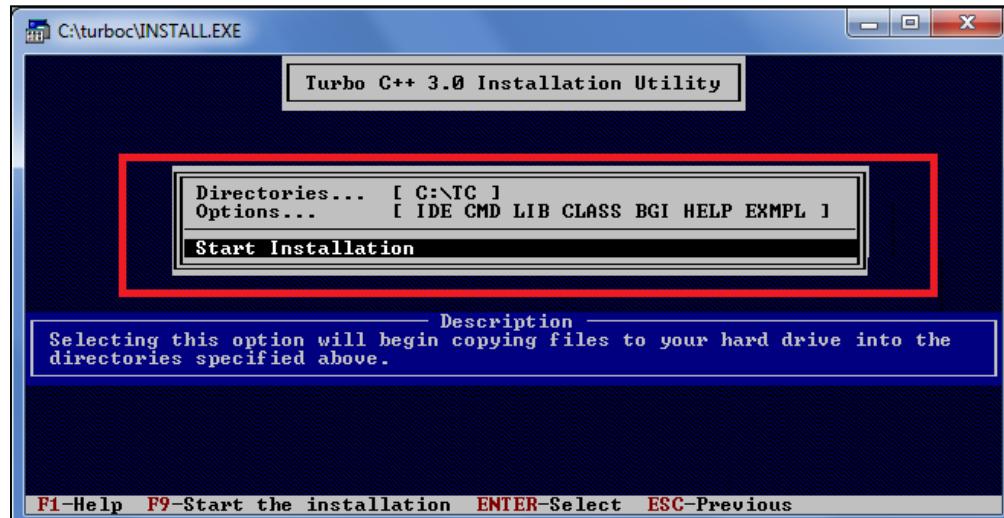
✓ Step-5: Press enter

It will look the directory for the required files.



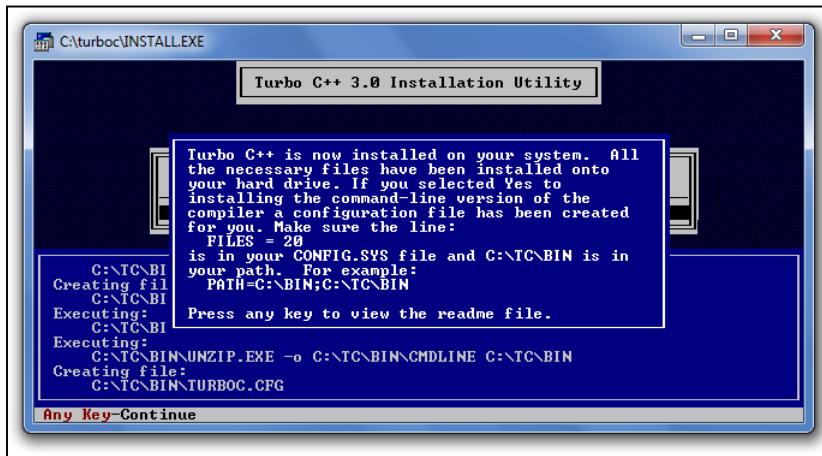
✓ Step-6: Start installation

Select start installation option by the down arrow key and then press enter.



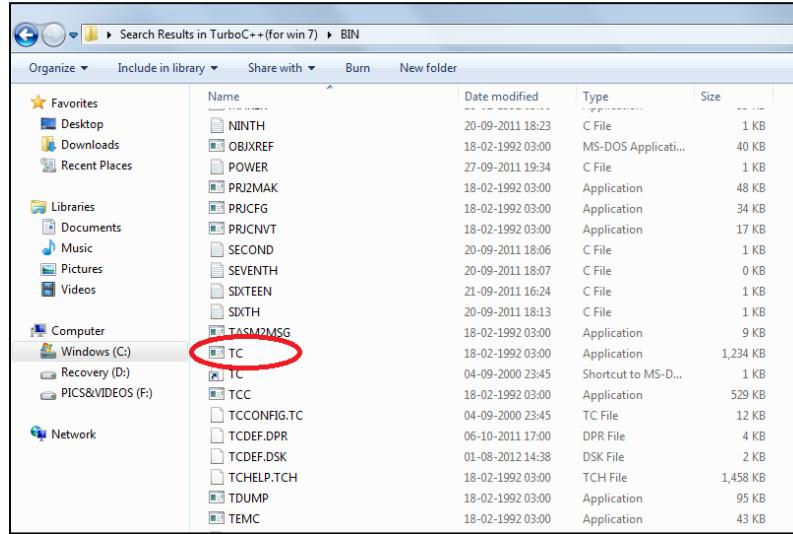
✓ Step-7: C is installed

C is properly installed in your system. So press enter to read a document or close the window.



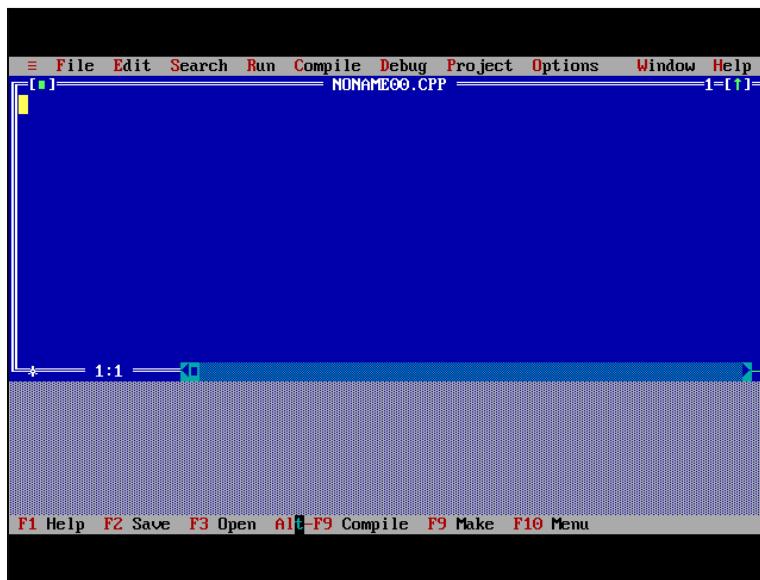
✓ Step-8: Click on TC application in BIN folder

Now select or double click on TC application to start programming.



✓ Step 10: A blue window will appear

Start coding in C and execute your programs.

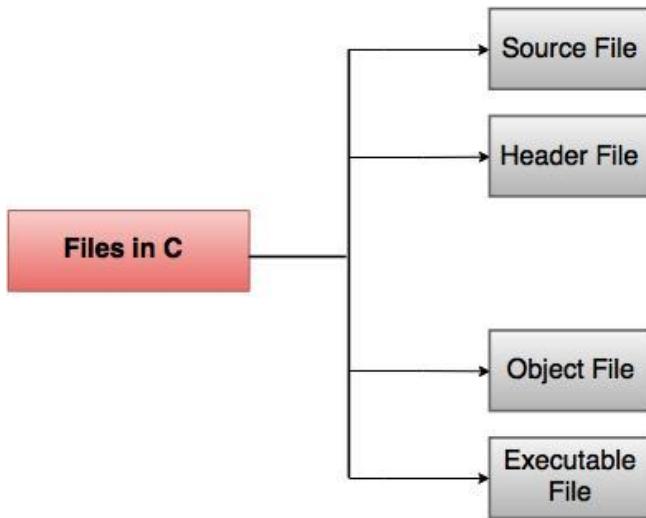


3.2 Files used in C Program Developer

A ‘C’ program uses following four types of files,

1. Source File
2. Header File

3. Object File
4. Executable File



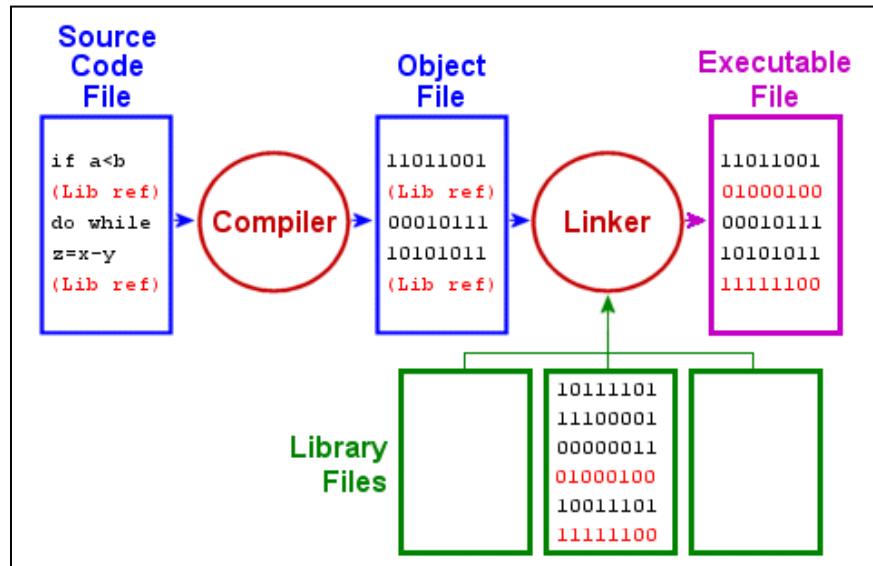
Source File / Source Code: Programming statements and instructions that are written by the developer. Source code is what a programmer writes, but it is not directly executable by the computer. It must be converted into machine language by compilers, assemblers or interpreters.

Header File: A header file is a file (.h extension) 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 come with compiler.

Object File: An Object file (.obj extension) is the compiled file itself. It is created when a C program (.c file) is compiled using a compiler. An executable file (.exe) is formed by linking the Object files. Object file contains

low level instructions which can be understood by the CPU. That is why it is also called machine code.

Executable File: A file in a format that the computer can directly execute. Unlike source files, executable files cannot be read by humans. To transform a C source file into an executable file, programmer needs to pass it through a C compiler.



3.3 Use of IDE (Integrated Development Environment)

IDE is a tool that provides user interface with compilers to create, compile and execute C programs. For example Turbo C++, Borland C++ and DevC++ etc. These provide Integrated Development Environment with compiler for both C and C++ programming language.

Advantages of Using an IDE

Using an IDE could save a lot of effort in writing a program. Some advantages include:

Increases Efficiency: The entire purpose of an IDE is to make program development faster and easier. Its tools and features are support for the programmer to organize resources, prevent mistakes, and provide shortcuts.

Collaboration: A group of programmers can easily work together within an IDE, simply by working in the same development environment.

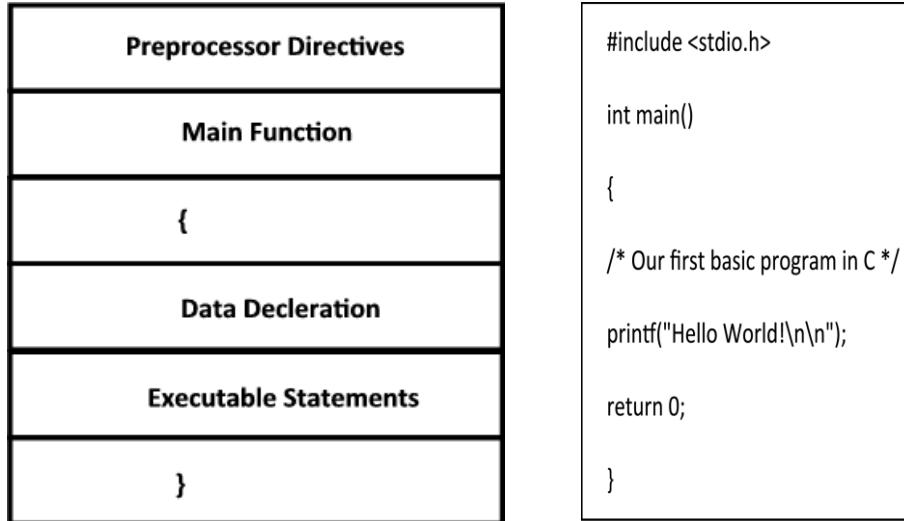
Project management: Many IDEs have documentation tools that either automate the entry of developer comments, or may actually force developers to write comments in different areas. Project management with team of developers is much easier using IDE tools and features.

Disadvantages to IDEs

- ✓ May be too complex for beginning programmers
- ✓ Each IDE will have a unique learning curve requiring time to learn
- ✓ Cannot automatically fix errors, still need knowledge to code efficiently

3.4 Structure of C Program

The basic structure of C program consists of following sections,



Basic Structure of C Program

A Simple C Program

- ✓ **#include<stdio.h>:** This command is a preprocessor directive in C that includes all standard input-output files before compiling any C program so as to make use of all those functions in our C program.

- ✓ **int main():** This is the line from where the *execution of the program starts*. The main() function starts the execution of any C program.

- ✓ **{ (Opening bracket):** This indicates the *beginning of any function* in the program. Here in this example it indicates the beginning of the main function.

- ✓ **/* some comments */:** Whatever is inside `/*-----*/` are *not compiled and executed*. They are only written for user understanding or for making the program interactive by inserting a comment line. These are known as multiline comments.

- ✓ **printf("Hello World"):** The printf() command is included in the C stdio.h library, which helps to *display the message on the output screen*.

- ✓ **getch():** This command helps to *hold the screen*.

- ✓ **return 0:** This command terminates the C program and returns a null value, that is, 0.

- ✓ **} (Closing brackets):** This indicates the *end of the function*. Here in this example it indicates the end of the main function.

Preprocessor Directives

The preprocessor will process directives that are inserted into the C source code. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself.

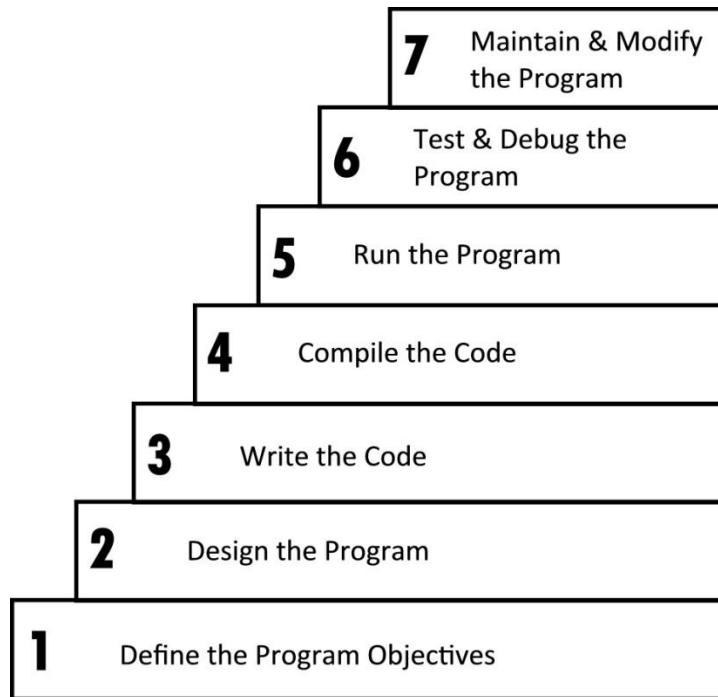
Main Function

main() function is the entry point of any C program.

	<p>It is the point at which execution of program is started. When a C program is executed, the execution control goes directly to the main() function. Every C program have a main() function.</p>
Data Declaration	<p>Data declaration determines the name and data type of a variable. Programmers declare variables by writing the name of the variable into code, along with any data type indicators and other required syntax.</p>
Executable C Statements	<p>These are the lines of code written for the processor. Compiler translates these lines into binary executable codes.</p>

3.5 Stages to develop a Program

Generally following seven steps are involved to develop a program,



✓ **Step 1: Define the Program Objectives**

The programmer should start with a clear idea of what he wants the program to do. Think in terms of the information program needs, the feats of calculation and manipulation the program needs to do. And finally the information the program should report back to the programmer or user. At this level of planning, a programmer should be thinking in general terms, not in terms of some specific computer language.

✓ **Step 2: Design the Program**

After the programmer has a conceptual picture of what his program have to do, he should decide how the program will go about it. What should the user interface be like? How should the program be organized? Who will the target user be? How much time he have to complete the program?

He also need to decide how to represent the data in the program.

✓ **Step 3: Write the Code**

Now that the programmer has a clear design for his program, he can begin to implement it by writing the code. That is, to translate his program design into the programming language. Here is where he really has to put his knowledge of programming languages to work. As part of this step, he should document his work. The simplest way is to use comments facility to incorporate explanations into his source code.

✓ **Step 4: Compile**

The next step is to compile the source code. Again, the details depend on programming environment (IDE). In case of C language, compiler is a program whose job is to convert source code into executable code. Executable code is code in machine language. C compilers also include code from C libraries into the final program. The end result is an executable file containing code that the computer understands and that you can run. The compiler also checks that the program is valid C code. If the compiler finds errors, it reports them to programmer and doesn't produce an executable file.

✓ **Step 5: Run the Program**

Integrated development environments (IDEs), such as those provided for Windows allow programmer to edit and execute C programs from within the IDE by selecting choices from a menu or by pressing special keys. The resulting program also can be run directly from the operating system by clicking or double-clicking the filename or icon.

✓ **Step 6: Test and Debug the Program**

The fact that program runs is a good sign, but it's possible that it could run incorrectly. Consequently, programmer should check to see that his program does what it is supposed to do. Debugging is the process of finding and fixing program errors.

✓ **Step 7: Maintain and Modify the Program**

A programmer might think of a better way to do something in the program in future. He could add new feature. He might adapt the program so that it runs on a different computer system. All these tasks are greatly simplified if he documents the program clearly and if he follows sound design practices.

END OF CHAPTER 03

Multiple Choice Questions with Answer

1. C Language has been developed by?

- a) Ken Thomson b) Dennis Ritchie c) Peter Norton d) Martin Richards

2. C can be used on?

- a) only MS-Dos b) only Linux c) only Windows d) All of the these.

3. C programs are converted into the machine language with the help of

- a) An editor b) Compiler c) An operating system d) None of these

4. The real constant in C can be expressed which of the following forms?

- a) Fractional form only b) Exponential form only c) ASCII form only d) Both

A &B

5. A C variable cannot start with

- a) An alphabet
- b) A number
- c) A special symbol other than underscore
- d) Both B &C

6. We declare a function with _____ if it does not have any return type

- a) Long
- b) double
- c) Void
- d) int

7. Arguments of a functions are separated with

- a) comma(,)
- b) Semicolon(;)
- c) colon (:)
- d) None of these

8. Strings are character arrays. The last index of it contains the null-terminated character.

- a) \n
- b) \t
- c) \o
- d) \I

9. Preprocessor does not do which of the following

- a) macro
- b) Conditional compilation
- c) type checking
- d) including load file

10) Which is the difference between the two declaration?

#include<stdio.h> & #include “stdio.h”

- a) No Difference
- b) The 2nd declaration will not compile
- c) First case Compiler looks in all default location and in 2nd case only in the working directory.
- d) Depends on the Compiler

11) C is a _____ Computer Programming Language

- a) Multi-Purpose
- b) General Purpose
- c) Both a and b
- d) None of these.

12) C was developed between

- a) 1955-1962
- b) 1969-1978
- c) 1969-1973
- d) 1964-1969

13) The type of comments are

- a) Single line comments
- b) Block Comments
- c) Both a and b
- d) None of these.

14) A _____ is not a part of the actual program

- a) Directive b) Preprocessor c) Both a & b d) None of these.

15. Single line comment start with

- a) /* b) */ c. () d. //

16. Block comment start with

- a) /* b) */ c. () d. //

17. Comments are useful in program_____

- a) Execution b) Compilation c) Maintenance d) None of these.

18. The Compiler translates the preprocessor modified source code into

- _____
a) Header file b) Object Code c) Machine Code d) Both b and c.

19. The Process of isolating and removing errors(bugs) form a program is called

- a) Debugging b) Testing c) Both a and b d) None of these.

20. To Compile a program in a command line mode we write

- a) bl b) al c) cl d) dl

Answers

1. b	2. d	3. b	4. d	5. d
6. c	7. a	8. c	9. c	10. c
11. b	12. c	13. c	14. a	15. d
16. b	17. c	18. d	19. a	20. c

SHORT QUESTIONS ANSWERS

Q 1. What is Turbo C++?

Ans. Turbo C++ is an Integrated development environment (IDE) for creating C and C++ programs. Borland international has developed it. It is also called TC editor. It is used to create, edit and save programs. It also has powerful debugging features. These help us in finding and removing errors from a program. We can easily compile program. Linking a program is also very easy. It is also used to execute a program.

Q 2. What are necessary step to prepare a C program?

Ans. Step to prepare a C program

Creating & Editing

Saving

Compiling

Linking

Loading

Running

Q 3. What are header files?

Ans. Header files are part of C compiler. C language provides many built-in programs. Every program has a unique name. These programs are called built-in functions or library functions. Every library function can perform a specific task. We can use these library functions in our C language program. These functions are divided into groups according to their functionality. A group of same type of functions are stored in a same file. This file is called header file.

Q 4. What is C statement?

Ans. Every instruction written in C language program is called a C statement. Every statement ends with a semicolon ";". Semicolon is called statement terminator.

Q 5. What are syntax errors?

Ans. The rule for writing a program in a specific programming language is called syntax of the language. We must follow the syntax of a language. Syntax error occurs when the statements of program are not according to syntax. Compiler detects syntax errors. If there is a syntax error in program. It cannot be compiled successfully. Compiler tells about the location and type of syntax error. Syntax errors can be removed easily.

Q 6. What are logical errors?

Ans. The error that is due to wrong algorithm is called logical error. These errors occur due to the wrong logic of program. Compiler cannot detect these errors. A program having logical errors gives wrong results on execution. These errors are difficult to find, as compiler cannot detect these errors. The programmer should examine the whole program to find logical errors.

Q 7. What are runtime errors?

Ans. These errors occur during the execution of program are called runtime errors. When runtime error occur the execution of program stops and computer shows an error message. These errors occur when program wants to perform such task that computer cannot perform.

Q 8. What is ANSI C?

Ans. C language is very powerful and flexible language. Wide range of application programs are written in C language. American National Standard Institute (ANSI) made standard version of C language in late 1980s. This standard version of C is also called ANSI C. New version of C has many new features that were not available in older versions.

Q 9. List any four advantages of C language?

Ans. Advantages of C language

- Easy to learn
- Easy to Remove Errors
- Machine Independence
- Standard Syntax
- Shorter Programs

Q 10. What is meant by machine independence?

Ans. A low level language program can run only on the type of computers for which it is written. So low level languages are machine dependent. A program written in high level language is machine independent. It can run on all types of computers.

Long Question

1. Write the steps to developing a program in C?
2. Describe the structure of C Program in details?
3. Describe the Debugging feature in detail?
4. Describe Integrated Development Environment in detail?
5. What is the Difference between Compiler and Interpreter?

C BUILDING BLOCKS

04

4.1 Data Types & variables

In computer programming data declaration determines the name and data type of a variable. Programmers declare variables by writing the name of the variable into code, along with any data type indicators and other required syntax.

4.1.1 Data Types

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. C provides various types of data-types which allow the programmer to select the appropriate type for the variable to set its value. The type of a variable determines how much space it occupies in memory. C Data Types are used to:

- ✓ Identify the type of a variable when it declared.
- ✓ Identify the type of the return value of a function.
- ✓ Identify the type of a parameter expected by a function.

ANSI C provides three types of data types:

- ✓ **Primary** (Built-in) Data Types:

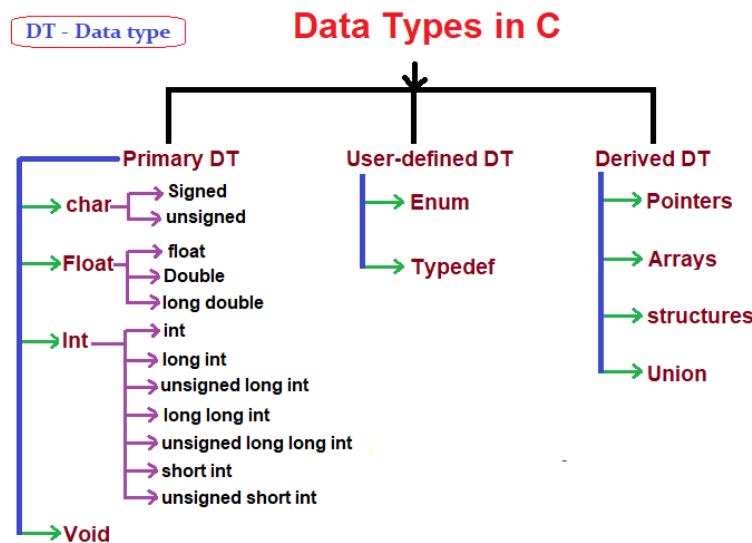
void, int, char, double and float.

- ✓ **Derived** Data Types:

Array, References, and Pointers.

✓ **User Defined Data Types:**

Structure, Union, and Enumeration.



Sr. No	Data Type	Description
--------	-----------	-------------

Primary(Built-in) Data Types

1	void	It holds no value and is generally used for specifying the type of function or what it returns. If the function has a void type, it means that the function will not return any value.
---	------	--

2	int	Used to denote an integer type.
3	char	Used to denote a character type.
4	float, double	Used to denote a floating point type.
5	int*, float*, char*	Used to denote a pointer type.

Derived Data Types

1	Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
2	References	Function pointers allow referencing functions with a particular signature.
3	Pointers	These are powerful C features which are used to access the memory and deal with their addresses.

User Defined Data Types

1	Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
2	Union	These allow storing various data types in the

same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time.

- 3 Enum Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

The following table provides the details of standard data types with their storage sizes and value ranges:

Integer Types

Sr. No	Type	Storage size	Value range
1	char	1 byte	-128 to 127 or 0 to 255
2	unsigned char	1 byte	0 to 255
3	signed char	1 byte	-128 to 127
4	int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
5	unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295

6	short	2 bytes	-32,768 to 32,767
7	unsigned short	2 bytes	0 to 65,535
8	long	8 bytes	-9223372036854775808 to 9223372036854775807
9	unsigned long	8 bytes	0 to 18446744073709551615

Floating Point Types

Sr. No	Type	Storage size	Value range	Precision
1	float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
2	double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
3	long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

4.1.2 Variables:

Variables are the names given by programmer to computer memory locations (Usually RAM) which are used to store values in a computer program. The value stored in a variable can be changed during program execution. A variable is only a name given to a memory location, all the operations done on

the variable effects that memory location. In C, all the variables must be declared before use. Each variable in C has a specific,

- ✓ Type, which determines the size and layout of the memory location for that variable.
- ✓ The range of values that can be stored within that memory.
- ✓ And the set of operations that can be applied to the variable.

Variables Declaration & Initialization in C:

All variables in C must be declared before using them in program. A declaration specifies a type and list of one or more variables of that type. Here is the syntax,

Type Variable_List;

Here in this syntax,

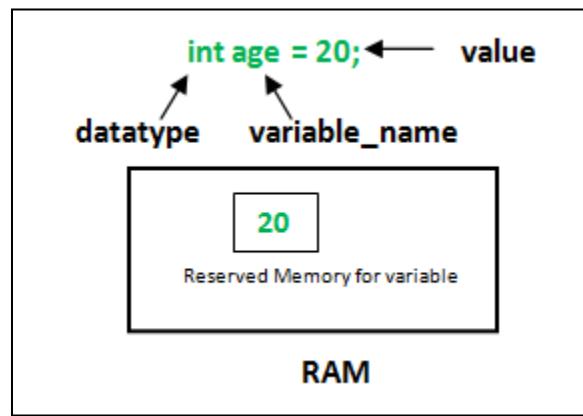
- ✓ Type must be a valid C data type such as int, float, char etc.
- ✓ Variable_List may consist of one or more identifier names separated by commas.

Some valid variable declarations are,

```
int    i, j, k;  
char   c, ch;  
float  f, salary;  
double d, b;
```

A variable declaration does not allocate any memory space for the variable but a variable definition allocates required memory space for that variable. A variable declaration with an initial value as shown below will become variable definition automatically,

```
int      i = 100;
```



In above figure,

- ✓ **Data type:** Type of data that can be stored in this variable.
- ✓ **variable_name:** Name given to the variable.
- ✓ **value:** It is the initial value stored in the variable.

To assign a value to a variable at the time of declaration is known as initialization of variable. Assignment operator '=' is used to assign value to a variable. The syntax to initialize a variable is as,

Type Variable_Name = Initial_Value

For example: *int d = 3, f = 5; /* initializing d and f */*

4.2 Input & Output in C

In a programming language,

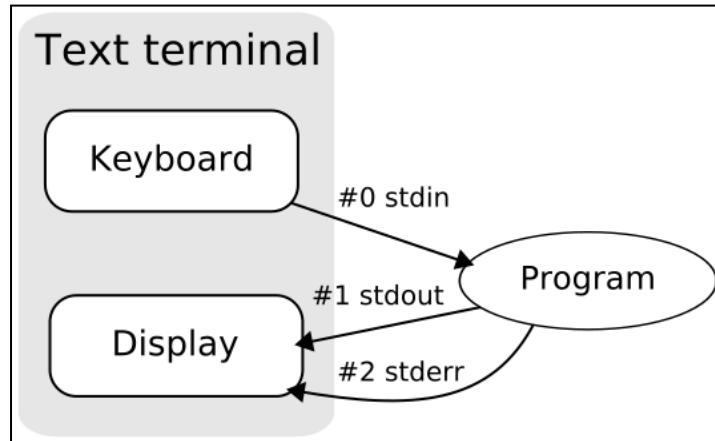
- ✓ Input means to feed some data into a program. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.
- ✓ Output means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

4.2.1 The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	Stdin	Keyboard
Standard output	Stdout	Screen

Standard error	Stderr	Your screen
----------------	--------	-------------



The file pointers are the means to access the file for reading and writing purpose. Following functions of C are used to read values from the keyboard and to print the result on the screen.

- **The `getchar()` and `putchar()` Functions**
- ✓ The `int getchar(void)` function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. This method can be used in the loop in case to read more than one character from the screen.
- ✓ The `int putchar(int c)` function puts the passed character on the screen and returns the same character. This function puts only single

character at a time. This method can be used in the loop in case to display more than one character on the screen. Example,

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

    printf( "Enter a value :");
    c = getchar( );

    printf( "\nYou entered: ");
    putchar( c );

    return 0;
}
```

When the above code is compiled and executed, it waits for user to input some text. When user enters a text and presses enter, then the program proceeds and reads only a single character and displays it as follows,

Enter a value : this is test

You entered: t

- o **The gets() and puts() Functions**

A string can be read using the %s conversion specification in the scanf function. However, it has a limitation that the strings entered cannot contain spaces and tabs. To overcome this problem, the C standard library provides the gets() function. It allows reading a line of characters (including spaces and tabs) until the newline character is entered (Enter key is pressed). A call to this function takes the following form,

gets(s);

Where *s* is an array of char (a character string). The function reads characters entered from the keyboard until newline is entered and stores them in the argument string *s*. The C standard library provides another function named puts to print a string on the display. A typical call to this function takes the following form,

puts(s);

Where *s* is an array of char (a character string). This string is printed on the display followed by a newline character.

```
#include <stdio.h>
int main( )
{
    char str[100];
    printf( "Enter a value :");
```

```
gets( str );  
  
printf( "\nYou entered: " );  
puts( str );  
  
return 0;  
}
```

When the above code is compiled and executed, it waits for user to input some text. When user enter a text and press enter, then the program proceeds and reads the complete line till end and displays it as follows,

Enter a value : this is test

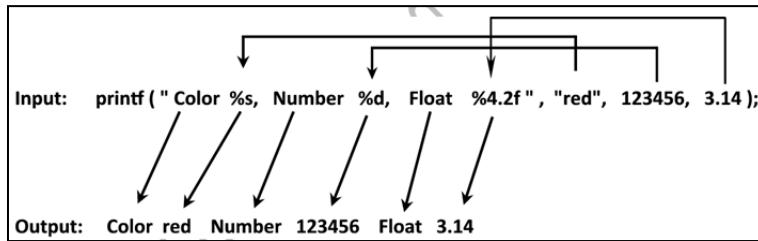
You entered: this is test

- o **The scanf() and printf() Functions**

The printf() Function: This function is used to print any text as well as value of the variables on the standard output device/Console (monitor screen). The printf() is very basic library function in c language that is declared in stdio.h header file. The printf() function prints all types of data values to the console. It requires format conversion symbol or format string and variable names to print the data. Following is the Syntax to write this function,

```
printf("message text");  
printf("message text+ format-string", variable-list);
```

First printf() style print the simple text on the monitor, while second printf() prints the message with values of the variable list.



```

#include <stdio.h>
int main()
{
    char ch = 'A';
    char str[20] = "gctbwp.edu.pk";
    float flt = 10.234;
    int no = 150;
    double dbl = 20.123456;

    printf("Character is %c \n", ch);
    printf("String is %s \n" , str);
    printf("Float value is %f \n", flt);
    printf("Integer value is %d\n" , no);
    printf("Double value is %lf \n", dbl);
    printf("Octal value is %o \n", no);
    printf("Hexadecimal value is %x \n", no);
}
    
```

```
    return 0;  
}
```

Here in this program printf() function is used with %c format specifier to display the value of an character variable. Similarly %s is used to display string, %f for float variable, %d for integer variable, %lf for double and %x for hexadecimal variable. To generate a new line “\n” is used in printf() statement.

Character is A

String is gctbwp.edu.pk

Float value is 10.234000

Integer value is 150

Double value is 20.123456

Octal value is 226

Hexadecimal value is 96

The scanf() Function: This function is used to read formatted data (character, string or numeric data) from keyboard. The scanf() is very basic library function in c language that is declared in stdio.h header file. Following is the Syntax to write this function,

```
scanf( "formatSpecifier" , address of variable );
```

Here at ‘*address of variable*’ field, an ‘&’ symbol is used with the name of variable, that is infect pointer to the value stored in memory at that variable location.

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

```
int main()
{
    int a, b, c;

    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);

    c = a + b;

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

    return 0;
}
```

Output of the above program is as,

```
Enter the first value: 3
Enter the second value: 4
3 + 4 = 7
```

Using only one scanf() function, more than one variables can be feed, for example if a is integer and b is float then a and b variables can be feed using only one scanf statement,

```
Scanf(“ %d %f”, &a, &b);
```

4.3 Operator & their Use in C

An operator is a symbol that operates on a value or a variable. Operator tells the compiler to perform specific mathematical or logical operation. C language provides following types of operators,

- o Arithmetical Operators
- o Increment and Decrement Operators
- o Assignment Operators
- o Relational Operators
- o Logical Operators
- o Bitwise Operators
- o Misc Operators

1. Arithmetic Operators

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

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division

Operator	Meaning of Operator
%	remainder after division (modulo division)

```

#include <stdio.h>
int main()
{
    int a = 9, b = 4, c;

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

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

    c = a*b;
    printf("a*b = %d \n",c);

    c = a/b;
    printf("a/b = %d \n",c);

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

    return 0;
}

```

```
}
```

Output:

a+b = 13

a-b = 5

a*b = 36

a/b = 2

Remainder when a divided by b=1

The operators +, - and * computes addition, subtraction, and multiplication respectively. In normal calculation, $9/4 = 2.25$. However, the output is 2 in the program. It is because both the variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25. The modulus operator % computes the remainder. When a=9 is divided by b=4, the remainder is 1. The % operator can only be used with integers.

2. Increment and Decrement Operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1. Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

```
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
```

```

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

return 0;
}

```

Output:

```

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

```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`.

3. Assignment Operators

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

Operator	Example	Same as
<code>=</code>	<code>a = b</code>	<code>a = b</code>
<code>+=</code>	<code>a += b</code>	<code>a = a+b</code>

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

```
#include <stdio.h>

int main()
{
    int a = 5, c;

    c = a; /* c is 5 */
    printf("c = %d\n", c);

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

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

    c *= a; /* c is 25 */
    printf("c = %d\n", c);

    c /= a; /* c is 5 */
}
```

```
printf("c = %d\n", c);

c %= a; /* c = 0 */
printf("c = %d\n", c);

return 0;
}
```

Output:

c = 5

c = 10

c = 5

c = 25

c = 5

c = 0

4. Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1 and if the relation is false, it returns value 0. Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1

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

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

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

    return 0;
}
```

}

Output:

5 == 5 is 1

5 == 10 is 0

5 > 5 is 0

5 > 10 is 0

5 < 5 is 0

5 < 10 is 1

5 != 5 is 0

5 != 10 is 1

5 >= 5 is 1

5 >= 10 is 0

5 <= 5 is 1

5 <= 10 is 1

5. Logical Operators

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

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) &&

Operator	Meaning	Example
		(d>5)) equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;
    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);
    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);
    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);
    result = !(a != b);
    printf("!(a == b) is %d \n", result);
    result = !(a == b);
```

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

return 0;
}
```

Output:

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

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

6. Bitwise Operators

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

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

7. Misc Operators (The Sizeof Operator)

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

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

```
float b;  
double c;  
char d;  
printf("Size of int=%lu bytes\n",sizeof(a));  
printf("Size of float=%lu bytes\n",sizeof(b));  
printf("Size of double=%lu bytes\n",sizeof(c));  
printf("Size of char=%lu byte\n",sizeof(d));  
  
return 0;  
}
```

Output:

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

END OF CHAPTER 04

Multiple Choice Questions with Answers

1. Character set of a C Language contains

- a) Alphabets
- b) Digits
- c) Special Symbol
- d) All of these

2. A variable name can have

- a) Any Special symbol
- b) Blank Space
- c) Comma(,)
- d) underscore

3. In C language, one of the following is not a valid data type.

- a) Long
- b) Float
- c) Double
- d) Char

4. The format string %if used for ?

- a) float b) double c) unsigned int d) long double

5. A variable of type unsigned int can have a value in the range of

- a) -32768 to + 32768 b) 0 to 32767 c) 0 to 65535 d) -32767 to + 32767

6. Which type of data is not a primary data type

- a) int b) array c) Float d) char

7. Which of the format string is not valid

- a) %ld b) %lf c) %lu d) %lc

8. Which is the valid string data

- a) 'A' b. A c. "A" d. None of these.

9. How much memory is required to store a value of type double

- a) 4 bytes b) 6 bytes c) 8 bytes d) 10 bytes.

10. The modifier which is used to declare a variable as constant

- a) Short b) Signed c) Unsigned d) Const.

11. Variable is a:

- a) Location in memory b) Location in CPU registers c) Both d) None

12. int can store:

- a) Real numbers b) Character c) String d) None of these

13. The arithmetic Operator ‘%’ can be used with

- a. int b. float c. double d. void

14. ‘%d’ is the conversion letter for:

- a) char b) int c) float d) double

15. Binary operator's needs.

- a) one Operand b) Two Operand c) Three Operand d) None of these.

16. Which is the symbol for AND operator

- a) || b) && c) \$\$ d) None of these

17. >> operator is used for

- a) Right shift b) Left Shift c) Both d) None of these

18. C program starts executing from:

- a) main() b. header file c) both d) None of these

19) Which is the Correct variable name:

- a) for b) goto c) character d) if

20. '\n' used for

- a) alert b) New line c) form feed d) Backspace.

Answers

1.d	2. d	3. a	4. d	5. c
6.b	7. d	8.c	9.c	10.d
11.c	12.d	13.a	14.b	15.b
16.b	17.a	18.a	19.c	20.b

Short Questions with Answers**Q.1 Define Data Type?**

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types.

Q.2 Define Variable?

Variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory ; and the set of operations that can be applied to the variable.

Q.3 Define LValue?

An expression that is an LValue may appear as either the left-hand or right-hand side of an assignment.

Q.4 Define RValue?

An expression that is an RValue may appear on the right but not left-hand side of an assignment.

Q.5 What do you mean by input in C language?

When we are saying input that means to feed some data into program. This can be given in the form of file or from command line.

Q.6 What do you mean by output in C Language?

When we are saying output that means to display some data into program. This can be given in the form of file or from command line.

Q.7 Explain scanf()

The int scanf(const char *format, ---) function reads according to format provided.

Q. 8 Explain printf()

The int printf(const char *format,---) function writes output to the standard output stream stdout and produces output according to a format provided.

Q.9 What is an Operator?

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations

Q.10 How can we initialize a Variable?

Variables are initialized(assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

Variable_name = Value;

Conditional Control Construct

Decisions

05

- 5.1. The if Statement
- 5.2. The if-else Statement
- 5.3. The else-if Statement
- 5.4. The switch Statement
- 5.5. The Conditional Operator

We all need to alter our actions in the face of changing circumstances. If the weather is fine, then will go for a stroll. If the highway is busy I would take a diversion. If the pitch takes spin, we would win the match. If she says no, I would look elsewhere. If you like this book, I would write the next edition. You can notice that all these decisions depend on some condition being met. C language too must be able to perform different sets of actions depending on the circumstances.

Control structure:

The statements of a computer programming are executed one after the other in the order in which they are written. This is known as sequential flow. It is the default flow of execution of statements in a program.

The order of execution of statements in a program can be changed. This is done with the help of control structure.

Control structure is statements that are used to control flow of execution in a program. A control structure consists of a group of statements. This group makes one logical unit in a program. It has one entry point and one exit point.

Control structure is used to implement the program logic.

Basic control structure:

Program instructions can be organized into three kinds of control structures to control execution flow. These are sequence, selection and repetition.

Following is a brief description of these structures.

- **Sequence structure:** In this structure, statements are executed in the order in which they occur in a program. This mode of execution of statements is called sequential flow. It is the default structure of programming languages.

- **Selection structure:** In selection structure, program statements are divided into two or more logical groups or block of statements. The selection structure chooses which group of statements is to execute. In C language, there are two basic selection structure. These are:

- **If-else**
- **Switch**

- **Repetition structure:** Repetition structure is also called **iteration structure** or **loop structure**. This control structure repeats a statement or group of statements in a program. There are three basic loop structures in C language. These are,

- **For loop**
- **While loop**
- **Do while loop**

Types of selection structure:

There are two types of selection structures in C language.

1. **If-else structure:** it is used to execute or skip a statement or set of statements according to condition.
2. **Switch Statement:** it is used when there are many choices are available and only one should be executed.

5.1 The if Statement:

If statement is used for branching when a single condition is to be checked. The condition enclosed in if statement decides the sequence of execution of instruction. If the condition is true statement or set of Statements inside if statement are executed, otherwise they are skipped. In C programming language, any non-zero value is considered as true and zero or null is considered false. If is a keyword in C language is statement is a decision-making statement. It is the simplest form of selection constructs. It is used to execute or skip a statement or set of statements by checking a condition. As a

general rule, we express a condition using C's 'relational' operators. The relational operators allow us to compare two values to see whether they are equal to each other, unequal, or whether one is greater than the other. Here's how they look and how they are evaluated in C.

This expression**is true if** $x == y$

x is equal to y

 $x != y$

x is not equal to y

 $x < y$

x is less than y

 $x > y$

x is greater than y

 $x <= y$

x is less than or equal to y

 $x >= y$

x is greater than or equal to y

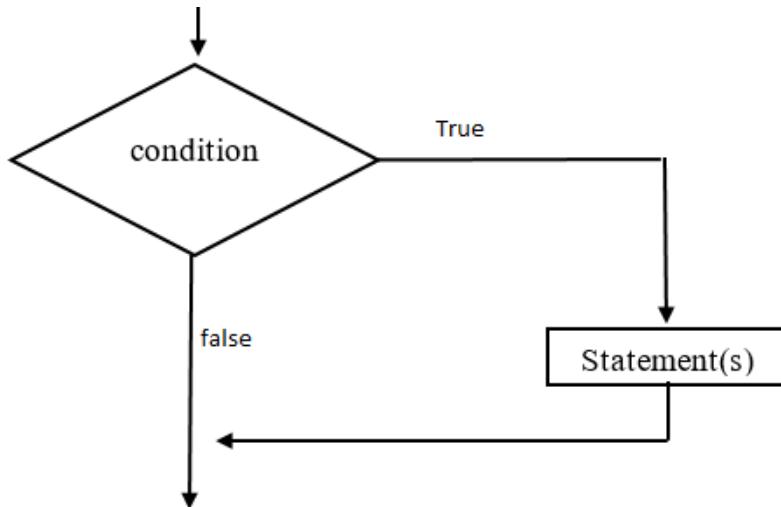
Syntax of if statement

if(this condition is true)

Execute this statement;

The above syntax is used for single statement. A set of statements can also be made conditional. In this case these Statement are written in curly brackets {}.

If statement flow chart



Write a program that inputs marks and displays “congratulations you have passed” if the marks are 40 or more.

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

void main( )
{
    Int marks;
    printf ("Enter student marks:");
    scanf ("%d", &marks);
    if (marks >=40)
        printf ("Congratulations you have passed! ");
    getch();
}
```

working of program:

On execution of this program, if you type students marks greater than 40, you get a message on the screen through **printf()** **Congratulations you have passed!**. If you type some other number the program doesn't do anything. The following flowchart would help you understand the flow of control in the program.

Output:
Enter student marks: 89
Congratulations you have passed!

Write a program that inputs three numbers and displays maximum number.

```
#include<conio.h>
#include<stdio.h>
void main()
{
    int a,b,c,max;
    clrscr();
    printf("Enter first number");
    scanf("%d",&a);
    printf("Enter second number");
    scanf("%d",&b);
    printf("Enter third number");
    scanf("%d",&c);
    max=a;
    if(b>max)
```

```
max=b;  
if(c>max)  
    max=c;  
printf("The maximum number is %d", max);  
getch();  
}
```

Output

```
Enter first number 10  
Enter second number 20  
Enter third number 30  
The maximum number is 30
```

Limitations of if statement:

If statement is a simple selection structure the if statement(s) are executed if the condition is true if the condition is false nothing happens in other words we may say it is not the effective one.

5.2 If else double selection statement

The if single-selection statement performs an indicated action only when the condition is true otherwise the action is skipped. The if else double selection statement allows you to specify an action to perform when the condition is true and a different action to perform when the condition is false.

If else Syntax:

```
if (condition)  
{  
    statement(s);
```

```

If condition is true
}
else
{
    statement(s);
}

```

If condition is false

```
}
```

Example:

```
if(student marks greater than or equal to 40)
```

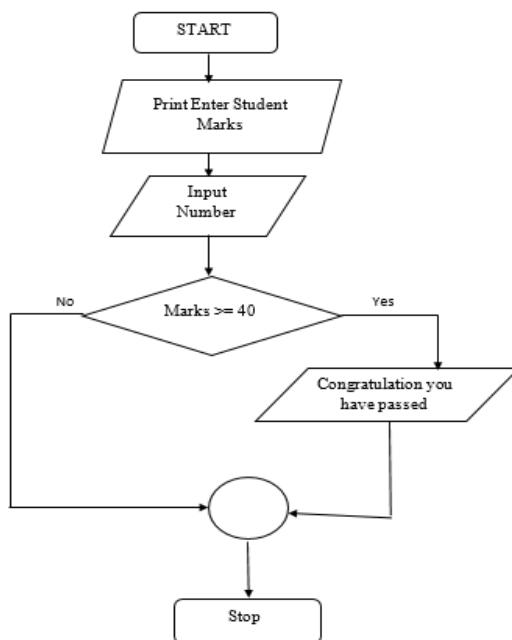
```
    Print"passed"
```

Else

```
    Print"failed"
```

- both blocks of statements can never be executed.
- Both blocks of statements can never be skipped.

Flow chart of IF-ELSE condition



Two or more statements are written in curly brackets { }. The syntax for compound statements of if else condition is as follow.

If-else compound statement syntax

```
if (condition)
{
    statement 1;

    statement 2;
```

.

Statement N;

```
}
```

else

```
{
    statement 1;
```

Statement 2;

.

Statement N;

```
}
```

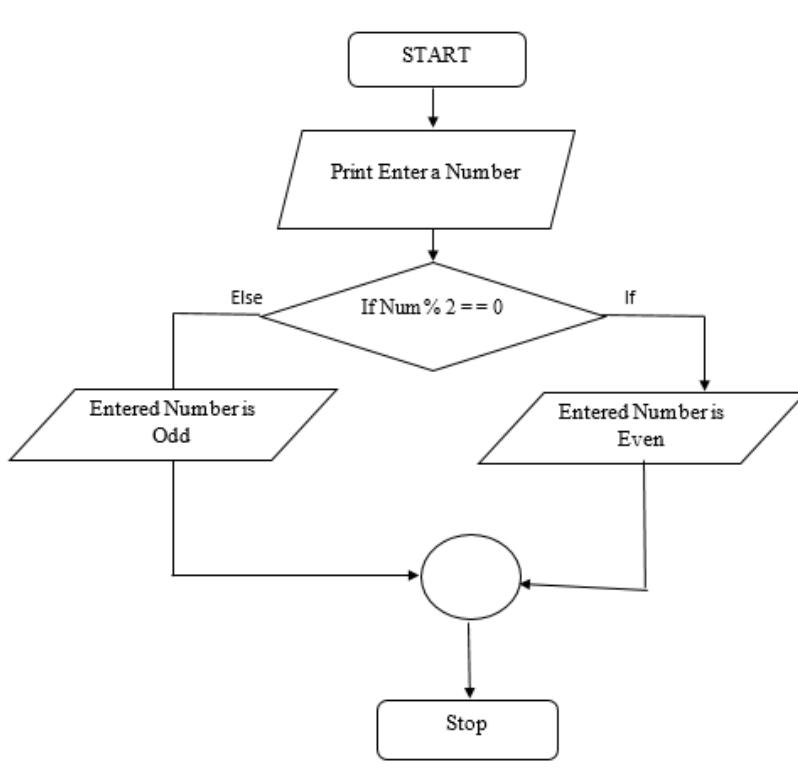
Write a program that inputs a number and finds whether it is even or odd using if-else

structure.

```
#include<conio.h>
#include<stdio.h>
void main()
int number;
clrscr();
printf("Enter a number");
scanf("%d",&number);
if(n%2==0)
    printf("%d Entered number is even");
else
    printf("%d Entered number is odd");
getch();
}
```

Output: 1
Enter a number 5
Entered number is odd
Output 2:
Enter a number 4
Entered number is even

Flow chart of even and odd program.



Working of even odd program:

If the input number remainder equal to zero than the given condition is true and if block is executed and print “Entered number is Even” and else block is ignored. If the input number remainder is not equal to zero than the given condition is false and if block is ignored and else block is executed and print “Entered number is Odd”

Write a program that input a number and display square root of the number if the number is positive, otherwise, the program displays a message “invalid input”.

```
#include<conio.h>
#include<stdio.h>
#include<math.h>
void main()
{
    int a;
    double sqroot=0.0;
    clrscr();
    printf("Enter any number:");
    scanf("%d",&a);
    if(a>=0)
    {
        sqroot=sqrt(a);
        printf("The square root of %d is=%lf",a,sqroot);
    }
    else
        printf("Invalid Input");
    getch();
}
```

Output: 1
Enter any number: 9
The square root of 9 is = 3.000000
Output: 2
Enter any number: -9
Invalid input

Working of square root program:

In this program if the entered number is greater than or equal to zero, then its square root can be calculated. Thus, when a value greater than or equal to zero is input, the condition becomes true and the statements following the if are executed and the statement following the else is ignored. If the entered number is less than zero, then the given condition becomes false and the statements following the if are ignored and the statement following the else is executed.

Write a program that inputs salary and grade. It adds 50% bonus if the grade is greater than 15. It adds 25% bonus if the grade is 15 or less then display the total salary.

```
#include<conio.h>
#include<stdio.h>
void main()
{
float salary,bonus;
int grade;
clrscr();
printf("Enter your salary= ");
scanf("%f",&salary);
printf("Enter your grade= ");
scanf("%d",&grade);
if(grade>15)
    bonus=salary*50.0/100.0;
```

```
else  
bonus=salary*25.0/100.0;  
salary=salary+bonus;  
printf("Your total salary is Rs f",salary);  
getch();  
}
```

Output 1

Enter your salary=20000

Enter your grade=17

Your total salary is Rs= 30000

Output 2

Enter your salary=10000

Enter your grade= 14

Your total salary is Rs= 12500

Working of program.

User input two values employ salary and grade if the grade is greater than 15 then if condition is true and if condition is executed and else condition is ignore, and 50% bonus is add to salary. If the employ grade is less then 15 then condition is false else statement is execute and if condition is ignore.

5.4 Else-if statement

If-else-if statement can be used to choose one block of statements from many blocks of statements. Is it used when there

are many options and only one block of statements should be executed on the basis of a condition.

Syntax of else-if statement:

```
if(condition 1)
{
    Statement Block 1;
}

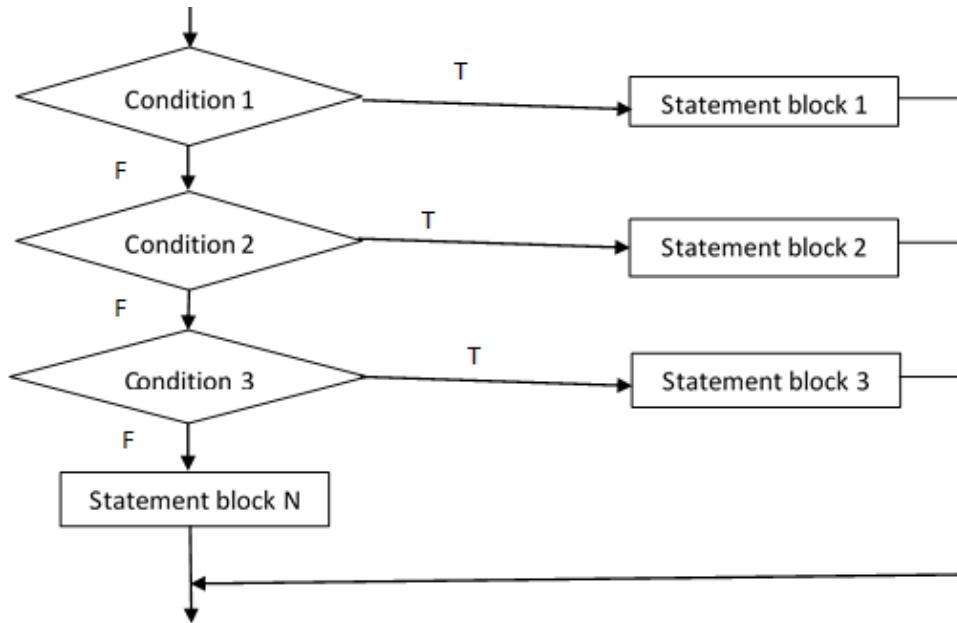
Else If(condition 2)
{
    Statement Block 2;
}

Else If(condition 3)
{
    Statement Block 3;
}

.
.

Else
{
    Statement Block N;
}
```

Flow chart of else-if:



Important Points:

- The test conditions in this structure are evaluated in the given sequence until a true condition is found.
- When a true condition is found, the statements associated with the condition are executed. And the remaining conditions are skipped.
- If a condition is false, its associated statements are not executed and the next condition is tested.
- If all conditions are false, then the statement following the last else statement is execute.

Working of if-else-if:

The test conditions in if-else statement with multiple alternatives are executed in a sequence until a true condition is reached. If a

condition is **true**, the block of statement is executed. The remaining blocks are skipped. If a condition is false, the block of statements following the condition is skipped. The statement after the last else are executed if all conditions are false.

Program:

Write a program that inputs a number from user and determines whether it is positive, negative or zero.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    clrscr();
    printf("Enter a Number:");
    scanf("%d",&n);
    if(n>0)
        printf("The number is positive");
    else if(n<0)
        printf("The number is negative");
    else
        printf("The number is zero");
    getch();
}
```

Output:

Enter a Number 9

The number is positive

Program:

Write a program that inputs salary if the salary is 20000 or more, it deducts 7% salary. If the salary is 10000 or more but less than 20000, it deducts 1000 from salary. If salary is less than 10000, it deducts nothing.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int salary;
    float net;
    clrscr();
    printf("Enter salary=");
    scanf("%d",&salary);
    if(salary>=20000)
        net=salary-(salary*7.0/100);
    else if(salary>=10000)
        net=salary-1000;
    else
        net=salary;
    printf("Your net salary is= %.2f", net);
    getch();
}
```

Output:

Enter salary=15000

Your net salary is=14000

Program:

Write a program that inputs marks of student and display his grade according to the following criteria.

Marks	grade
>=90	A+
80-89	A
70-79	B
60-69	C
50-59	D
< 60	Fail

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int grade;
printf("Enter student marks=");
scanf("%d",&grade);
if(grade>=90)
printf("Your grade is A+");
else if(grade>=80)
printf("Your grade is A");
else if(grade>=70)
printf("Your grade is B");
else if(grade>=60)
printf("Your grade is C");
else if(grade>=50)
printf("Your grade is D");
```

```
else
printf("You are fail!");
getch();
}
```

Output

Enter student marks=91

Your grade is A+

5.7 Decisions using switch:

In real life we are often faced with situations where we are required to make a choice between a number of alternatives rather than only one or two. For example, which school to join or which hotel to visit or still harder which girl to marry (you almost always end up making a wrong decision is a different matter altogether!). Serious C programming is same; the choice we are asked to make is more complicated than merely selecting between two alternatives. C provides a special control statement that allows us to handle such cases effectively; rather than using a series of **if** statements.

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch case-default**, since these three keywords go together to make up the control statement. They most often appear as follows:

Syntax of switch statement:

```
switch (integer expression)
{
    case constant 1:
        do this;
    case constant 2:
        do this;
    case constant 3:
        do this;
    default:
```

```
    do this;  
}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. The “do this” lines in the above form of **switch** represent any valid C statement. What happens when we run a program containing a **switch**? First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case** and **default** statements as well. If no match is found with any of the **case** statements, only the statements following the **default** are executed. A few examples will show how this control structure works.

Consider the following program:

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
clrscr();  
int i=2;  
switch(i)  
{  
case 1:  
    printf("I am in case 1 \n");  
case 2:  
    printf ("I am in case 2 \n");  
case 3:  
    printf("I am in case 3 \n");  
default:  
    printf("I am in default \n");  
}  
getch();  
}
```

The output of this program would be:

I am in case 2

I am in case 3

I am in default

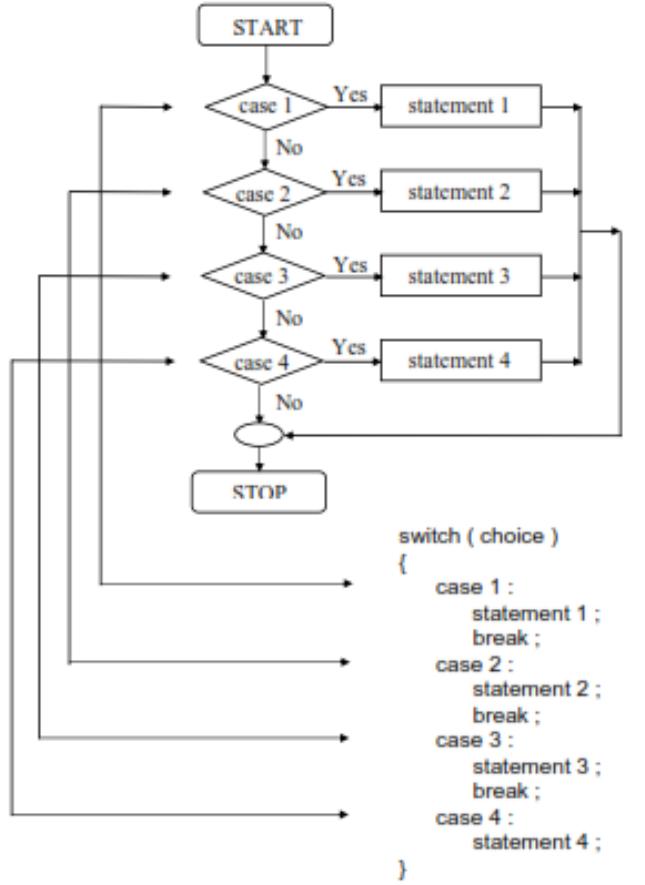
The output is definitely not what we expected! We didn't expect the second and third line in the above output. The program prints case 2 and 3 and the default case. Well, yes. We said the **switch** executes the case where a match is found and all the subsequent **cases** and the **default** as well. If you want that only case 2 should get executed, it is up to you to get out of the **switch** then and there by using a **break** statement. The following example shows how this is done. Note that there is no need for a **break** statement after the **default**, since the control comes out of the **switch** anyway.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=2;
    clrscr();
    switch (i)
    {
        case 1:
            printf("I am in case 1 \n");
            break;
        case 2:
            printf("I am in case 2 \n");
            break;
        case 3:
            printf("I am in case 3 \n");
            break;
        default:
            printf("I am in default \n");
    }
    getch();
}
```

The output of this program would be:

I am in case 2

The operation of **switch** is shown below in the form of a flowchart for a better understanding.



Program:

Write a program that inputs a number of weekdays and display the name of the day. For example if the user enter 5 it displays “Friday” and so on.

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

```
#include<conio.h>

void main()
{
    int day;
    clrscr();
    printf("Enter number of week day:");
    scanf("%d", &day);
    switch(day)
    {
        case 1:
            printf("Friday");
            break;
        case 2:
            printf("Saturday");
            break;
        case 3:
            printf("Sunday");
            break;
        case 4:
            printf("Monday");
            break;
        case 5:
```

```
printf("Tuesday");  
break;  
case 6:  
printf("Wednesday");  
break;  
case 7:  
printf("Thursday");  
break;  
default:  
printf("Invalid Number");  
}  
getch();  
}
```

Output:
Enter number of week day: 1
Friday

Program:

Write a program that inputs a character from user and checks whether it is a vowel or consonant.

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

```
{  
char c;  
clrscr();  
printf("Enter an alphabet= ");  
scanf("%c",&c);  
switch(c)  
{  
case 'a':  
case 'A':  
    printf("You entered vowel \n");  
    break;  
case 'e':  
case 'E':  
    printf("You entered vowel \n");  
    break;  
case 'i':  
case 'I':  
    printf("You entered vowel \n");  
    break;  
case 'o':  
case 'O':  
    printf("You entered vowel \n");  
}
```

```

break;

case 'u':

case 'U':

printf("You entered vowel \n") ;

break;

default:

printf("you entered a consonant");

}

getch();

}

```

Output:

Enter an alphabet= a

You entered vowel

Sr. #	Switch	Nested if
1.	It is easy to use when there are multiple choices.	It is difficult to use when there are multiple choices.
2.	It uses single expression for multiple choices.	It uses multiple expressions for multiple choices.
3.	It can't check a range of value.	It can check a range of value.
4.	It checks only constant values. Use can't use variable with case statement.	It checks only variables also. Use can use a constant or variable in relational expression

END OF CHAPTER 05

Multiple Choice and Answer

1) The three programming structures are

- a. Sequence, decision and repetition
- b. process, decision and alternation
- c. Sequence, definition and process
- d. relation, comparison and process

**2) which programming structure execute program statements
in order?**

- a. relation b. Decision c. Sequence d. repetition

3) Another term for computer making a decision is:

- a. Sequential b. Selection c. Repetition d. Iteration

4) which programming structure make a comparison?

- a. Relation b. Decision c. Sequence d. repetition

5) An expression that uses a relational operator is known as:

- a. operational b. Sequential c. Serial d. Relational

6) Relational operators allow you to_____ numbers.

- a. Compare b. Add c. multiply d. Divide

**7) The operators to compare a operands and decide if the relation
is true or false are:**

- a. Arithmetic operators b. logical operators
- c. syntax operators d. Relational operator

**8) Which of following statements is the simplest form of a decision
structure?**

- a. Select....case b. if statement c. try..catch..finally d. Nested if

9) _____ is used to specify two different choices with “if” statement:

- a. switch statement b. else statement
- c. if statement d. if-else

10) In if statement, false is represented by:

- a. 0
- b. 1
- c. 2
- d. 3

11) _____ refers to group of statements enclosed in opening and closing braces.

- a. control structure
- b. compound statement
- c. Sequence structure
- d. instruction

12) the conditional portion of the if statement contain.

- a. Any Valid constant
- b. Any expression that can be evaluated to a Boolean value
- c. Any valid variable
- d. Any valid constant or variable

13) What does a compound condition use to join two condition?

- a. Relational operator
- b. Logical operator
- c. Relational results
- d. Logical results

14) Which of the following are valid case statements in a switch?

- a. case 1:
- b. case x<4
- c. case 'ab':
- d. case 1.5:

15) The case block ends with:

- a. End select
- b. End case
- c. Break;
- d. case else

16) Another term for a conditional operator is:

- a. Ternary
- b. binary
- c. byte
- d. Iteration

17) Which operator in C is called a ternary operator?

- a. if
- b. ++
- c. ?
- d. ()

18) Which of the following is used for making two way decision:

- a. if-else
- b. if
- c. Nested-if
- d. switch

19) Switch statement is an alternative of:

- a. if
- b. if-else
- c. Nested-if
- d. nested if-else

20) if x=10 and y=5, what will the output of the following

expression?

$x>y ? x*y : x+y;$

- a. 5 b. 10 c. 15 d. 50

Answers:

1) a	2) c	3) b	4) b	5) d	6) a
7) d	8) b	9) d	10) a	11) b	12) b
13) b	14) a	15) c	16) a	17) c	18) a
19) d	20) d				

Short Question & Answers

Q1. Define Control structure.

A statement used to control the flow of execution in a program or function is called Control structure. The control structures in C are used to combine individual Instruction into a single logical unit. The logical unit has one entry point and one exit point.

Q2. Write three selection statements and three repetition statements.

Selection statements are if, if-else and switch. The repetition statements are while, do...while and for.

Q3. Describe sequence structure.

In sequence structure, the control flow from one statement to other in a logical sequence.

Q4. Describe a repetition structure.

A repetition structure executes a statement or set of statements repeatedly. It is also known as iteration structures are loop. The repetition structure includes

for loop, while loop and do-while loop.

Q5. Explain “if” statement.

If is a keyword in C language. If statement is a decision-making statement. It is the simplest form of selection constructs. It is used to execute or skip a statement or set of statement by checking a condition.

Q6. Define compound statement.

A set of statement written in curly brackets {} after if statement is called compound statement.

Q7. what is the use of “if-else” statement?

If-else statement is another type of if statement. It execute one block of statement(s)

when the condition is true and the other when it is false. In any situation, one block is executed and the other is skipped.

Q8. write the syntax of “if-else-if” statement.

```
If (condition)
    Statement;
Else
    Statement;
```

Q9. what is the use of “if-else-if” statements?

If-else-if statement can be used to choose one block of statements from many blocks

of statements. It is used when there are many option and only one block of statements

should be executed on the basic of a condition.

Q10. Differentiate between if and if-else statement.

If statement execute a statement or set of statement if given condition is True.

It does not perform anything if the condition is false. However, if-else

statement execute one statement or set of statements if the condition is true.

It also executes another statement or set of statements if the condition is false.

Long Questions:

1. Explain basic control structure?
2. Explain if statement with example?
3. What is switch statement? Explain working of switch statement?
4. Explain working of else-if statement?
5. Define compound statement and write syntax of if else Compound statement?

Iterative Control Construct:

Loops

06

- 6.1. The for Loop**
- 6.2. The while Loop**
- 6.3. The do while Loop**

A statement or set of statements that is executed repeatedly is known as loop. The structure that repeats a statements is known as iterative or looping construct.

These programs were of limited nature, because when executed, they always performed the same series of actions, in the same way, exactly once. Almost always, if something is worth doing, it's worth doing more than once. You can probably think of several examples of this from real life, such as eating a good dinner or going for a movie. Programming is the same; we frequently need to perform an action over and over, often with variations in the details each time. The mechanism, which meets this need, is the 'loop'. Loops are basically used for two purposes:

- To execute a statement or number of statements for a specified number of times. For example, a user may display his name on screen for 10 times.
- To use a sequence of values for example, a user may display a set of natural numbers from 1 to 100.

C programming language provides the following types of loop to handle looping requirements.

Loop Type	Description
<u>while loop</u>	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
<u>for loop</u>	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
<u>do-while loop</u>	Like a while statement, except that it tests the condition at the end of the loop body

While loop:

While loop is the simplest loop of C language. This loop executes one or more statements while the given condition remains true. It is useful where the number of iterations is not known in advance.

Syntax:

The syntax of a **while** loop in C programming language is:

```
while(condition)
{
    statement(s);
}
```

Condition: the condition is given as a relational expression. It controls the iteration of loop. The statement is executed only if the given condition is true. If the condition is false, the statement is never executed.

Statement: statement is the instruction that is executed when the condition is true. Two or more statements are specified in braces {}. It is called the body of the loop.

Syntax for compound statements:

While(condition)

{

Statement 1;

Statement 2;

.

.

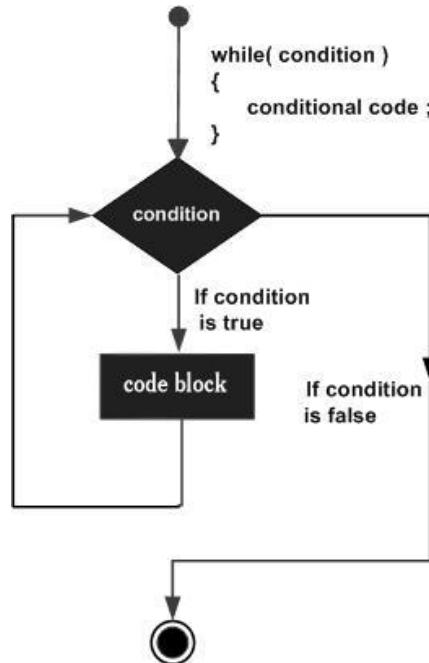
Statement N;

}

Working of while loop:

First of all the condition is evaluated if it is true the control enters the body of the loop and executes all statements in the body. After executing the statements, it again moves to the start of the loop and evaluates the condition again. This process continues as long as the condition remains true. When the condition becomes false, the loop is terminated. While loop terminates only when the condition becomes false. If the condition remains true the loop never ends. A loop that has no end point is known as an infinite loop.

Flow chart of while loop:



Program:

The following example displays “I love Pakistan” for 10 times using while loop:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    n=1;
    clrscr();
    while(n<=10)
    {
        printf("I love Pakistan");
        n++;
    }
    getch();
}
```

Output

I love Pakistan
I love Pakistan

Explanation of above program:

The above program uses a variable n to control the iteration of the loop. It is known as counter variable or loop control variable. It is used to control the number of iterations. The variable n is initialized to 1. It is always initialized outside the body of the loop. When the condition is evaluated for the first time, the value of n is less than 10 so the condition is true. The control enters the body of the loop. The body of the loop contains two statements. The first statement prints “I Love Pakistan” on the screen. The second statement increments the value of n by 1 and makes it 2. The increment or decrement statement is always written inside the body of the loop. The control moves back to the condition after executing both statements. This process continues for 10 times. When the value of n becomes 11 the condition becomes false and loop terminates.

Program:

Write a program that displays counting from 1 to 10 using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
n=1;
clrscr();
while(n<=10)
{
    printf("%d \n",n);
    n++;
}
getch();
}
```

Output:

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

Program:

Write a program that take inputs from the user and display a table of that number using while loop:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,c;
c=1;
clrscr();
printf("Enter a number:");
scanf("%d",&n);
while(c<=10)
{
    printf("%d * %d=%d\n",n,c,n*c);
c=c+1;
}
getch();
}
```

Output:

Enter a number: 4

4 * 1=4

4 * 2=8

4 * 3=12

.

.

4 * 10=40

Program:

Write a program that inputs a number from the user and display the factorial of the number using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,c,f;
c=1;
f=1;
clrscr();
printf("Enter a number:");
scanf("%d",&n);
while(c<=n)
{
    f=f*c;
    c=c+1;
}
printf("Factorial of %d is %d",n,f);
getch();
}
```

Output:

Enter a number:3

Factorial of 3 is 6

The Infinite Loop:A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>
#include <conio.h>
void main ()
{
    for(;;)
    {
        printf("This loop will run forever.\n");
    }
    getch();
}
```

Output:

This loop will run forever.

This loop will run forever.

This loop will run forever.

.

.

.

- When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.
- **NOTE:** You can terminate an infinite loop by pressing Ctrl + C keys.

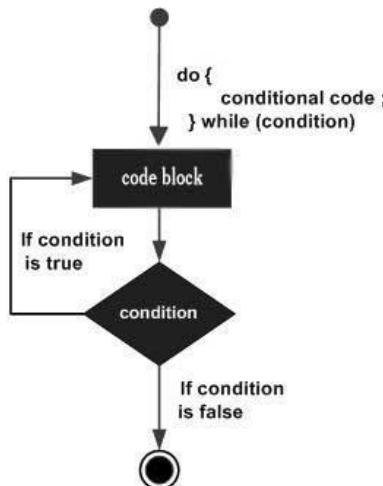
Do while loop:

The do while loop is an iterative control in C language. This loop executes one or more statements while the given condition is true. In this loop, the condition comes after the body of the loop. The Do while loop is important in a situation where a statement must be executed at least once.

Syntax of Do while loop:

```
do
{
    Statement 1;
    Statement 2;
    .
    .
    Statement N;
}
while(condition);
```

Flow chart of do while loop



Do: It is the keyword that indicate the beginning of the loop.

Condition: The condition is given as a relational expression. The loop continues only if the given condition is true. If the condition is false the loop

is terminated.

Statement: Statement is the instruction that is executed when condition is true. Two or more statements are written in braces { }. It is called the body of the loop. The important point about do-while loop is that the condition ends with a semicolon. An error occurs if the semicolon is not used at the end of the condition.

Working of do-while loop:

First of all the body of the loop is executed. After executing the statements in loop body, the condition is evaluated. If it is true the control again enters the body of the loop and executes all statements in the body again. This process continues as long as the condition remains true. The loop terminates when the condition becomes false. This loop is executed at least once even if the condition is false in the beginning.

Program:

The following example displays “I love Pakistan” for five times using do while loop.

```
#include<conio.h>
#include<stdio.h>
void main()
{
    int n=1;
    clrscr();
    do
    {
```

```
printf("I Love Pakistan\n");  
n++;  
}  
while(n<=5);  
getch();  
}
```

Output:

```
I Love Pakistan  
I Love Pakistan  
I Love Pakistan  
I Love Pakistan  
I Love Pakistan
```

Program:

Write a program that display back-counting from 8 to 1 using do-while loop.

```
#include<conio.h>  
  
#include<stdio.h>  
  
void main()  
{  
int n=8;  
clrscr();  
do
```

```
{  
printf("%d \n",n);  
n=n-1;  
}  
while(n>=1);  
getch();  
}
```

Output:

```
8  
7  
6  
5  
4  
3  
2  
1
```

Program:

Write a program that displays first give numbers with their cubes using do-while loop.

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

```
int n=1;
clrscr();
do
{
printf("%d \t %d\n",n,n*n*n);
n=n+1;
}
while(n<=5);
getch();
}
```

Output:

1	1
2	8
3	27
4	64
5	125

Different between while and do while loop:

While loop

In while loop condition comes before the body of the loop.

If condition if false in the beginning while loop is never executed.

The semicolon not used after the condition.

Do-while loop

In while loop condition comes after the body of the loop.

Do while loop is executed at least one even if condition is false at the beginning.

The semicolon used after the condition.

For loop:

For loop executes one or more statements for a specified number of times. This loop is also called counter-controlled loop. It is the most flexible loop. The for loop allows us to specify three things about a loop in a single line,

Syntax:

The syntax of a **for** loop in C programming language is:

for(initialization; condition; increment/decrement)

```
{  
    Statement 1;  
    Statement 2;  
    .  
    .  
    Statement N;  
}
```

Initialization: It specifies the starting value of counter variable. One or many variables can be initialized in this part. To initialize many variables, each variable separated by comma.

Condition: The condition is given as a relational expression. The statement is executed only if the given condition is true. If the condition is false, the statement never execute.

Increment/

decrement: This part of loop specifies the change in counter variable after each execution of the loop. To change many variables, each variable must be separated by comma.

Statement: Statement is the instruction that is executed when the condition is true. If two or more statements are used, these are given in braces {}. It is called the body of the program.

The for loop contains three expressions separated by semicolon. The initialization and increment / decrement expressions are optional.

A for loop may be written as follows:

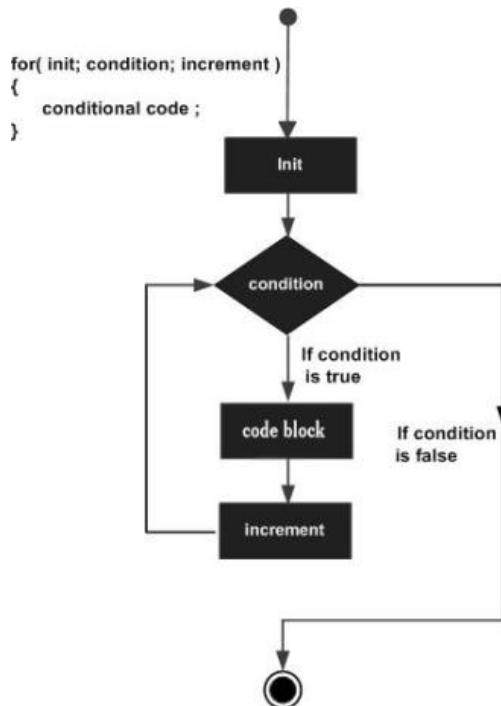
```
For( ; condition ; )
```

The above statement contains one condition expression. The initialization can be specified before the loop. The increment/decrement can be specified in the body of the loop. The condition in for loop is mandatory. It can't be omitted.

Working of “For loop”:

The number of iterations depends on the initialization, condition and increment/decrement parts. The initialization part is executed only once when the control enters the loop. After initialization, the given condition is evaluated. If it is true the control enters the body of the loop and executes all statements in it. Then the increment/decrement part is executed that changes the value of counter variable. The control again moves to condition part. This process continues while the given condition remains true. The loop is terminated when the condition is false.

Flow chart of for loop:



Program:

The following example displays counting from 1 to 5 using for loop.

```
#include <stdio.h>
#include <conio.h>
void main ()
{
    int n;
    clrscr();
    for(n=1; n<=5; n++)
        printf("%d\n",n);
    getch();
}
```

Output:
1
2
3
4
5

Working of above program:

In the above program the variable n is initialized with 1. The termination condition is $n \leq 5$. It means that the loop will continue while the value of counter is less than or equal to 5. The loop will terminate when the value of counter is greater than 5. The third part contains $n++$. It indicates that the value of n will be incremented by 1 after each execution of the loop

Program:

Write a program to display alphabets from A to Z using for loop.

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

```
void main ()  
{  
    char c;  
    clrscr();  
    for(c='A'; c<='Z'; c++)  
        printf("%c ", c);  
    getch();  
}
```

Output:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Program:

Write a program that inputs table number and length of table and then display the table using for loop.

```
#include <stdio.h>  
  
#include <conio.h>  
  
void main ()  
{  
    int tab, length,c;  
    clrscr();  
  
    printf("Enter number for table:");  
    scanf("%d",&tab);
```

Output:

Enter number for table: 4

Enter length of table:10

4 * 1 = 4

4 * 2 = 8

4 * 3 = 12

.

.

.

4 * 10 = 40

```
printf("Enter length of table:");

scanf("%d",&length);

for(c=1; c<=length; c++)
    printf("%d * %d = %d\n ",tab,c,tab*c);

getch();
}
```

Working of above program:

The above program uses a variable tab for table instead of a constant number. It gets a number from the user as a table. It also inputs a number in variable length up to which the table is displayed. It executes the loop for length times by using the condition $c \leq length$.

END OF CHAPTER 06

Exercise:

Multiple Choice Questions:

- 1. The continue statement cannot be used with**
A. for
B. while
C. do while
D. switch
- 2. Which keyword can be used for coming out of recursion?**
A. return
B. break
C. exit
D. both A and B
- 3. Goto can be used to jump from main to within a function?**

- A. TRUE
- B. FALSE
- C. May Be
- D. Can't Say

4. Switch statement accepts.

- A. int
- B. char
- C. long
- D. All of the above

5. Which loop is guaranteed to execute at least one time.

- A. for
- B. while
- C. do while
- D. None of the above

6. A labeled statement consist of an identifier followed by

- A. ;
- B. :
- C. ,
- D. =

7. do-while loop terminates when conditional expression returns?

- A. One
- B. Zero
- C. Non - zero
- D. None of the above

8. c = (n) ? a : b; can be rewritten as exp1 ? exp2 : exp3;

- A. if(n){c = a;}else{c = b;}
- B. if(!n){c = a;}else{c = b;}
- C. if(n){c = b;}else{c = a;}
- D. None of the above

9. Which of the following statement about for loop is true ?

- A. Index value is retained outside the loop
- B. Index value can be changed from within the loop
- C. Goto can be used to jump, out of the loop
- D. All of these

10. Using goto inside for loop is equivalent to using

- A. Continue
- B. Break
- C. Return
- D. None of the above

11. What is the final value of x when the code int x; for(x=0; x<10; x++) {} is run?

- A. 10
- B. 9
- C. 0
- D. 1

12. When does the code block following while(x<100) execute?

- A. When x is less than one hundred
- B. When x is greater than one hundred
- C. When x is equal to one hundred
- D. While it wishes

13. Which is not a loop structure?

- A. For
- B. Do while
- C. While
- D. Repeat Until

14. How many times is a do while loop guaranteed to loop?

- A. 0
- B. Infinitely
- C. 1
- D. Variable

15. In _____, the bodies of the two loops are merged together to form a single loop provided that they do not make any references to each other.

- A. Loop unrolling
- B. Strength reduction
- C. Loop concatenation
- D. Loop jamming

16. What is the output of this program?

```
void main()
{
    if(!printf(""))
        printf("hello");
    else
        printf("world");
}
```

- A. hello
- B. world
- C. Compilation Error
- D. None of the above

17. The three programming structures are:

- A. Sequence, decision and repetition
- B. Process, decision and alternation.
- C. Sequence, definition and process
- D. Relation, comparison and process

18. Which programming structure executes program statements in order.

- A. Relation.
- B. Decision.
- C. Sequence.
- D. Repetition.

19. If used to specify two different choices with if-else statement.

- A. Switch statement.
- B. Else statement.
- C. If statement.
- D. If -else

20. The operation to compare operands and decide if the relation is true or false are:

- A. Arithmetic operation
- B. Logical operator
- C. Syntax operator
- D. Relational operators

Answers

1.d	2.a	3.b	4.d	5.c
6.c	7.b	8.a	9.d	10.d
11.a	12.a	13.d	14.c	15.d
16.a	17.a	18.c	19.d	20.d

Short Questions & Answers

Q.1. Define loop?

A statement or number of statements that are executed repeatedly is known as loop.

Q2. Write two uses or advantages of loop?

Loops are used to execute a statement or number of statements for a specified number of times. Loops are used to access a sequence of values.

Q.3. which program control statements are used to control iterations?

The program control statements used to control iterations are while loop, do-while

loop and for loop.

Q.4. which part of the loop contains the statements to be repeated?

The loop body is the part of the loops that contains the statements to be repeated.

Q.5. which three steps must be done using loop control variable?

The three steps that must be done using the loop control variable are initialization,

test and increment/decrement.

Q.6. Define “while” loop.

“while” loop is the simplest loop of C language. It executes one or more statements ,

while the given condition remains true. It is useful where the number of iterations is

not known in advance.

Q.7. Define Do-While loop?

The do-while is an iterative control in C language. It executes one or more statement

while the given condition is true. In this Loop, the condition comes after the body of

the loop. The loop is important in a situation where a statement must be executed at

least once.

Q.8. Describe the syntax of while loop with example.

The Syntax of while loop is as follows:

while(condition)

 Statement(s);

Example

```
int count =0;  
while (count < 10)  
{  
    Printf( “ C programming”)  
    Count++;  
}
```

Q.9.Describe the syntax of do-while loop?

The Syntax of while loop is as follows

```
do
{
    Statement 1;
    Statement 2;
}
while(condition);
```

Q.10.Describe the Syntax of for loop?

The syntax of this loop as follows:

```
for(initialization; condition; increment decrement)
{
    Statement 1;
    Statement 2;
}
```

Long questions

1. Explain working of while loop?
2. Explain do-while loop?
3. Explain working of for loop?
4. Write a program that display product of all odd numbers from 1 to 10
Using for loop?
5. Write a program that inputs an integer and display its table in
descending Order using for loop?

Functions

07

- 7.1. Introduction**
- 7.2. Simple Functions and Value-Returning Functions**
- 7.3. Parameter Passing**
- 7.4. Using Multiple Functions and External Variable**
- 7.5. Preprocessor Directives**
- 7.6. Recursion**

Knowingly or unknowingly we rely on so many persons for so many things. Man is an intelligent species, but still cannot perform all of life's tasks all alone. He has to rely on others. You may call a mechanic to fix up your bike, hire a gardener to mow your lawn, or rely on a store to supply your groceries every month. A computer program (except for the simplest one) finds itself in a similar situation. It cannot handle all the tasks by itself. Instead, it requests other program like entities called 'functions' in C to get its tasks done. In this chapter we will study these functions. We will look at a variety of features of these functions, starting with the simplest one and then working towards those that demonstrate the power of C. Suppose you have a task that is always performed exactly in the same way you monthly servicing of your motorbike. When you want it to be done, you go to the service station and say, "It's time, do it now". You don't need to give instructions, because the mechanic knows his job. You don't need to be told when the job is done. You assume the bike would be serviced in the usual.

What is a Function:

A function is a named block of code that performs some action. The statements written in a function are executed when it is called by its name. each function has a unique name. functions are the building blocks of C programs. They encapsulate piece of code to perform specified operations. the

functions are used to accomplish the similar kinds of tasks again and again without writing the same code again. They are used to perform the tasks that are repeated many times. The functions provide a structured programming approach. It is a modular way of writing programs. The whole program logic is divided into a number of smaller modules or functions. The main function calls these functions when they are needed to execute.

Importance of functions:

A program may need to repeat the same piece of code at various places. It may be required to perform certain tasks repeatedly. The program may become very large if functions are not used. The piece of code that is executed repeatedly is stored in a separate function. The real reason of using functions is to divide a program into different parts. These parts of a program can be managed easily.

Advantages of functions:

Some important benefits or advantages of using functions are as follows:

Easier to code:

A lengthy program can be divided into small functions. It is easier to write small functions instead of writing a long program. A function is written to solve a particular problem. A programmer can focus the attention on a specific problem. Functions make programming easier.

Easier to modify:

Each function has a unique name and is written as an independent block. If there is any error in the program, the change is made to the particular function in which the error exists. A small function is easier to modify than a large program.

Easier to maintain and debug:

Functions are easier to maintain than long programs. Each function contains independent code. A change in the function does not affect other parts of the program. In case of an error, the infected function is debugged only. The user does not need to examine the whole program.

Reusability:

The code written in function can be reused as and when required. A function but can be executed many times. A function is written to solve a particular problem. Whenever that problem has to be solved, the function can be executed. Suppose a function line displays a line on the screen. It can be executed in different parts of the program to display the line again and again.

Less programming time:

A program may consist of many functions. These functions are written as independent programs. Different programmers can work on different functions at the same time. It takes far less time to complete the program.

Types of function:

There are two types of function in C programming:

- Standard library functions
- User-defined functions

User-defined functions:

A type of function written by the programmer is known as user-defined functions. User-defined function has a unique name. a program may contain many user-defined functions. These functions are written according to the exact need of the user.

Standard library functions:

A type of function that is available as a part of language is known as built-in function or standard library function. These functions are ready-made programs. These functions are stored in different header files. Built-in functions make programming faster and easier.

For example,

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the <stdio.h> header file.

Hence, to use the printf() function, we need to include the <stdio.h> header file

using `#include<stdio.h>`. The `sqrt()` function calculates the square root of a number. The function is defined in the `math.h` header file to use the `sqrt()` function we need to include the `<math.h>` header file using `#include<math.h>`.

Function declaration:

Specifying the name, return-type, arguments, etc. of a function is called function declaration. It is also called function prototype.

A function is declared in the program in which it is to be used. The function declaration provides the following information to the compiler.

- The name of function.
- The data type returned by the function.
- The number and types of arguments or parameters used in the function.

Semicolon is used to the end of function declaration to indicate the end of the function declaration. The function declaration is similar to a variable declaration. The rules for naming functions are same as of naming variables.

Syntax of function declaration:

`return_type function_name(argument list)`

return_type: Return type can be of any data type such as `int`, `double`, `char`, `void`, `short` etc. specifies the data type returned by the function. For example, `int` is used if the function returns integer type data. Similarly, `float` is used if the function is to return floating-point data. It is also possible that a function does not return any data. If the function does not return any value, the keyword `void` is used. It indicates the type of value that will be returned by function.

function_name: It indicates the name of the function. It can be anything; however, it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing its name.

argument list: Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer

argument. If the function needs no parameters, the keyword void is used or the parentheses are left blank.

Example:

```
Void display(void);
```

This function declaration provides the following information to the compiler.

- Name of the function is “display”.
- Return data type is void it does not return any value.
- List of parameters is void no parameters or arguments is required.

```
int sum(int x, int y);
```

This function declaration provides the following information to the compiler.

- Name of the function is “sum”.
- Return data type is integer.
- Two parameters, both of integer type are required.

The x and y are not being declared as variable. These are only identifiers and are used only to indicate that values are to be provided to the compiler. Their use is optional. The function declaration can also be without these identifiers. If these identifiers are not used, only data types of the arguments are specified. For example, the above function declaration can also be specified as.

```
int sum(int, int);
```

Example:

```
float temp(void);
```

This function declaration provides the following information to the compiler.

- Name of the function is “temp”.
- Return data type is float.
- List of parameters is void, no parameters or arguments are required.

Example:

```
void print(int, float, char);
```

This function declaration provides the following information to the compiler.

- Name of the function is “print”.
- Return data type is void, it does not return any value.
- Three parameters first of integer, second of float type and third of character type are required.

Function definition:

The actual code of the function is called function definition. It consists of a set of instructions that are written to perform a specific task. Function definition is always outside the main() function. It can be written before or after the main() function. It can also be written in a separate file. If a function is written in a separate file, it is included in the program using #include directive. The function definition consists of the following parts.

- Function header
- Function body

Function header:

It is the first line of the function definition. It is also called declarator. It is the same as the function declaration but it is not terminated by semicolon (;).

Function body:

The set of statements enclosed in braces after the declarator are called body of the function. The statements are written in the body of the function to perform a task.

The general format of the function definition is:

```
return_type function_name ([arguments]) // function header
{
    set of statements      // body pf function
}
```

Return type: specifies the data type returned by the function. The void type specifies that no data is returned by the function.

Function_name: specifies the name of function. It must be the same name that is used in the function declaration.

Arguments: specify the list of parameters or arguments separated by commas (,). The variable and their types that are to be provided to the function are given as parameters. The data type of parameters must be same and in same order as mentioned in the function declaration. The parameters receive the data values that are passed to the function. If the function needs no data values or parameters then word “void” is written in braces are left blank(). The variable used in parameters list are treated as local variables. They cannot be declared in the function body.

Example:

Following is an example of a simple function the function takes two integer values as arguments and find out the maximum of the two.

```
int max(int a, int b)
{
    If(a > b)
        printf("a is greater");
    else
        printf("b is greater");
}
```

Function calls:

The statement that activates a function is known as function call. A function is called with its name. Function name is followed by necessary parameters in parentheses. If there are many parameters, these are separated by commas. If there is no parameter, empty parentheses are used. When a function is called, the following steps take place.

- The control moves to the function that is called.
- All statements in the function body are executed.
- The control returns back to the calling function.

When the control returns back to the calling function, the remaining statements in the calling function are executed.

Program:

Write a program that displays a message “Programming makes life interesting” on screen using function.

```
#include<stdio.h>
#include<conio.h>
void show(void);
void main()
{
    clrscr();
    show();
    getch();
}
void show()
{
    printf("Programming makes life interesting.");
}
```

Output:

Programming makes life interesting.

Process of passing parameters to function:

Parameters are the values that are provided to a function when the function is called. Parameters are given in the parentheses. If there are many parameters, they are separated by commas. If there is no parameters, empty parentheses are used. The sequence and types of parameters in function call must be similar to the sequence and types of parameters in function declaration.

- Parameters in function call are called actual Parameters.
- Parameters in function declaration are called formal Parameters. The formal Parameters are also known as dummy arguments.

The number of actual parameter must be same as number of formal Parameter. The values of actual Parameter are copied to formal Parameters when a function call is executed.

```
void show(int); // function declaration
```

```
void main()
{
int n;
printf("Enter number:");
scanf("%d",&n);
show(n); // function call
printf("End of program");
}
```

Program:

Write a program that inputs two numbers in main function, passes these numbers to a function. The function displays the maximum number.

```
#include<stdio.h>
#include<conio.h>
void max(int a, int b);
void main()
{
int x,y;
clrscr();
printf("Enter two numbers: ");
scanf("%d %d", &x, &y);
max(x,y);
getch();
}
void max(int a, int b)
{
if(a>b)
    printf("maximum number is %d",a);
else
    printf("maximum number is %d",b);
}
```

Output:
Enter two numbers:4 7
Maximum number is 7

Program:

Write a program that takes inputs a number in main function and passes the number to a function. The function displays table of that number.

```
#include<stdio.h>
#include<conio.h>
void table(int a);
void main()
{
int num;
clrscr();
printf("Enter any numbers to print table: ");
scanf("%d",&num);
table(num);
getch();
}
void table(int num)
{
int c;
for(c=1; c<=10; c++)
{
    printf("%d * %d = %d \n",num ,c,num * c);
}
}
```

Program:

<p>Output: Enter any numbers to print table: 5 5 * 1= 5 5 * 2= 10 5 *3 = 15 . . 5 * 10 = 50</p>

Write a program that inputs a number in main function and passes the number to a function. The function displays the factorial of that number.

```
#include<stdio.h>
#include<conio.h>
void factorial(int n);
void main()
{
    int num;
    clrscr();
    printf("Enter any number to print factorial: ");
    scanf("%d",&num);
    factorial(num);
    getch();
}
void factorial(int n)
{
    int i;
    int fact;
    fact=1;
    for(i=1; i<=n; i++)
        fact*=i;
    printf("Factorial of %d is %d", n, fact);
}
```

Output:
Enter any number to print factorial: 3
Factorial of 3 is 6

Function return value:

A function can return a single value. The return type in function declaration indicates the type of value returned by a function. For example, int is used as return type if the function returns integer value. If the function returns no value, the keyword void is used as return type. The keyword return is used to

return the value back to the calling function. When the return statement is executed in a function, the control moves back to the calling function along with the returned value.

Syntax:

The syntax for returning a value is as follows.

```
    return expression;
```

Expression: It can be a variable, constant or an arithmetic expression whose value is returned to the calling function. The calling function can use the returned value in the following ways.

- Assignment statement.
- Arithmetic expression.
- Output statement.

Assignment statement:

The calling function can store the returned value in a variable and then use this variable in the program.

Arithmetic expression:

The calling function can use the returned value directly in an arithmetic expression.

Output statement:

The calling function can use the returned value directly in an output statement.

Program:

Write a function to convert kilograms into grams. Input weight in the main function, convert it into grams using the function and then print the weight in grams.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int kg, gm;
    int grams (int);
    clrscr();
    printf("Enter weight in kilograms: ");
    scanf("%d",&kg);
    gm=grams(kg);
    printf("%d kilogram = %d grams", kg, gm);
    getch();
```

```
}
```

```
int grams(int weight)
```

```
{
```

```
    return weight*1000;
```

```
}
```

Output:

```
Enter weight in kilograms: 3
```

```
3 kilogram = grams 3000
```

Preprocessor Directives:

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directives & Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefined 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.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.

- #error** Prints error message on stderr.
- #pragma** Issues special commands to the compiler, using a standardized method.

Preprocessors Examples:

Analyze the following examples to understand various directives.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE  
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as

```
#ifndef MESSAGE  
#define MESSAGE "You wish!"
```

```
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG  
/* Your debugging statements here */  
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Recursion:

Recursion is the process of repeating items in a self-similar way. Same applies in programming languages as well where if a programming allows you to call a function inside the same function that is called recursive call of the function as follows.

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

The C programming language supports recursion, 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 in infinite loop. Recursive function is very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

Number Factorial:

Following is an example, which calculates factorial for a given number using a recursive function:

```
#include<stdio.h>
#include<conio.h>
int factorial(int i)
{
if(i <=1)
{
return(1);
}
return i * factorial(i - 1);
}
int main()
```

```
{  
int i =5;  
printf("Factorial of %d is %d\n", i, factorial(i));  
getch();  
return(0);  
}
```

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

Factorial of 5 is 120

Fibonacci Series:

Following is another example, which generates Fibonacci series for a given number using a recursive function:

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

```
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
```

END OF CHAPTER 07

Exercise

Multiple Choice Questions.

- 1. Which of the following is type of function available in C language?**
a. User - defined b. Built – in c. Subprogram d. Both a and b
- 2. Another name for built - in function is:**
a. User-defined function b. Library function c. Arithmetic function d. Both a&b
- 3. A type of function that is available as part of language is known as:**
a. User-defined function b. Library function c. Sub – program
d. Both a&b
- 4. Function prototype for built - in functions are specified in:**
a. Source file b. Header file c. Object files d. Image files
- 5. The name of actual and formal parameters:**
a. May or may not be same b. Must be same c. Must be different d. Must be in lower case
- 6. Formal arguments are also called:**
a. Actual arguments b. Dummy arguments c. Original arguments d. Referenced arguments
- 7. The printf is a:**
a. Built - in function b. User - defined function c. Local function d. Keyword
- 8. A built - in function:**

- a. Cannot be redefine b. can be redefined c. Cannot return a value d. should be redefined

9. Function declaration consists of:

- a. Function name b. Function return type
- c. Number and types of parameters d. All of these

10. Function definition can be written:

- a. Before main () function b. After main () function c. In a separate file d. All of these

11. The function definition consists of:

- a. Function header or function declaration b. Function body
- c. Both a and b d. None

12. The statement that activates a function is known as:

- a. Function call b. Function output c. invoking a Function
- d. None

13. What is the variable name that is used by a function to receive passed values?

- a. Function b. Parameter c. Expression d. Constant

14. Which of the following is incorrect?

- a. A function can call another function
- b. A function can be called many times in a program
- c. A function can return values input by the user
- d. A function must have at least one value parameter

15. The parameters in function declaration are called:

- a. Formal parameters b. Actual parameters c. Both a and b
- d. None

16. The scope of a variable refers to its:

- a. Length b. Name c. Accessibility d. Data type

17. A variable declared inside a function is known as:

- a. Local variable b. Global variable c. Automatic variable
- d. a&c

18. A variable declaration outside any function is known as:

- a. Global variable b. Local variable c. External variable
- d. Static

19. Which of the following looks for the prototypes of functions used in a program?

- a. Linker b. Loader c. Compiler d. Parser

20. A type of function written by the programmer is known as:

- a. User-defined b. subprograms c. subroutines d. built - in function

Answers:

1. d	2. b	3. b	4. b	5. a
6. b	7. a	8. a	9. d	10. d
11. c	12. a	13. b	14. d	15. a
16. c	17. d	18. a	19. c	20. a

Short Questions

1. Define function? Why is it used in a programs?

A Function is a named block of code that performs some action. The statement

written in a function are execute when it is called by its name. Each function has

a unique name. Functions are the building blocks if C program. The real reason of

using function is to provide a program into different parts. These parts of a program

can be managed easily.

2. How does function make programming easier?

A lengthy program can divided into small functions, it is easier to write small functions

instead of writing a long program. A programmer can focus the attention on a specific

problem. It makes programming easier.

3. List some benefits of using function?

- | | |
|---------------------------------|---------------------|
| 1. Easier To Code | 2. Easier to Modify |
| 3. Easier To Maintain And Debug | 4. The Usability |

4. List different type of function in C?

A type of Function written by the programmer is as none as user defines

function. User define function has unique name. A Program May contain many user

define functions. These functions are written According to the exact need of the user.

5. Describe built in Functions?

A type of function that is available as a part of language is known as built in

functions or library functions. These functions are ready-made programs. These functions are stored in different header files. Built in functions make programming fast and easier.

6. Define Function Body?

The Set Of statement which are executed inside the function is none as function body. The body of function appear after function header. These statements are written in Curly Braces {}. The variable declaration and program logic are implemented in function body.

7. What is function declaration or function prototype?

Function Deceleration is a Model Of A function. It Is Also Known As Function Prototype. It Provide Information to Compiler About The structure of the function to the used in the program. It consists of functions Name, Function return type and Number and type of Parameters.

8. What is function Definition?

A set Of Statements that explains what a function Does Is Called Function defamnation. The Function definition consists Of Function Header and function body.

9. Differentiate between function definition and function declaration?

Function definition consists of different statements to perform a task .Function declaration consists of single statement that provide information to compiler about function. Function Deceleration is written in Braces but Function definition is not written in braces.

10.What is a Function Call?

The statements that activate a function is known as function call. A Function is call by its name. Function name is followed by Necessary parameters. Parentheses If there are many parameters, these are separated by commas.

Long Questions.

1. Briefly explain the benefits of using functions?
2. Explain Importance of functions?
3. Explain function body and function header?
4. Write a program that inputs two numbers in main function and passes them to a function. The function displays first number raised to the power of second number?
5. Explain local variable?

Arrays and Strings

08

- 8.1. Introduction**
- 8.2. Single and Two-dimensional Arrays**
- 8.3. Strings**

8.1 Introduction

The C language provides a capability that enables the user to design a set of similar data types, called array. For example an int array holds the elements of int types while a float array holds the elements of float types and char array holds the elements of char types.

What are Arrays:

For understanding the arrays properly, let us consider the following program:

```
main( )
{
    int x ;
    x = 5 ;
    x = 10 ;
    printf ( "\nx = %d", x ) ;
}
```

No doubt, this program will print the value of x as 10. Why so? Because when a value 10 is assigned to x, the earlier value of x, 5, is lost. Thus, ordinary variables (the ones which we have used so far) are capable of holding only one value at a time (as in the above example). However, there are situations in which we would want to store more than one value at a time in a single variable.

For example, suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case we have two options to store these marks in memory:

1. Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.

2. Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Obviously, the second alternative is better. A simple reason for this is, it would be much easier to handle one variable than handling 100 different variables. Moreover, there are certain logics that cannot be dealt with, without the use of an array. Now a formal definition of an array. An array is a collective name given to a group of 'similar quantities'. These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees. What is important is that the quantities must be 'similar'. Each member in the group is referred to by its position in the group. For example, assume the following group of numbers, which represent percentage marks obtained by five students.

```
per = { 48, 88, 34, 23, 96 }
```

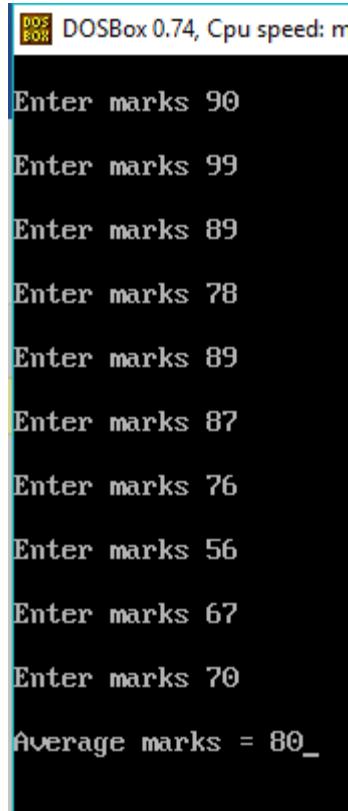
If we want to refer to the second number of the group, the usual notation used is per2. Similarly, the fourth number of the group is referred as per4. However, in C, the fourth number is referred as per[3]. This is because in C the counting of elements begins with 0 and not with 1. Thus, in this example per[3] refers to 23 and per[4] refers to 96. In general, the notation would be per[i], where, i can take a value 0, 1, 2, 3, or 4, depending on the position of the element being referred. Here per is the subscripted variable (array), whereas i is its subscript.

Thus, an array is a collection of similar elements. These similar elements could be all ints, or all floats, or all chars, etc. Usually, the array of characters is called a 'string', whereas an array of ints or floats is called simply an array. Remember that all elements of any given array must be of the same type. i.e. we cannot have an array of 10 numbers, of which 5 are ints and 5 are floats.

A Simple Program Using Array

Let us try to write a program to find average marks obtained by a class of 10 students in a test.

```
#include<stdio.h>
#include<conio.h>
main( )
{
clrscr();
int avg, sum = 0 ;
int i ;
int marks[10] ; /* array declaration */
for ( i = 0 ; i <= 9 ; i++ )
{
printf ( "\nEnter marks " ) ;
scanf ( "%d", &marks[i] ) ; /* store data in array */
}
for ( i = 0 ; i <= 9 ; i++ )
sum = sum + marks[i] ; /* read data from an array*/
avg = sum / 10 ;
printf ( "\nAverage marks = %d", avg ) ;
getch();
}
```



DOSBox 0.74, Cpu speed: m

```
Enter marks 90
Enter marks 99
Enter marks 89
Enter marks 78
Enter marks 89
Enter marks 87
Enter marks 76
Enter marks 56
Enter marks 67
Enter marks 70
Average marks = 80_
```

Array Declaration:

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program we have done this with the statement. `int marks[30] ;`

Here, `int` specifies the type of the variable, just as it does with ordinary variables and the word `marks` specifies the name of the variable. The `[30]` however is new. The number 30 tells how many elements of the type `int` will be in our array. This number is often called the ‘dimension’ of the array. The bracket `([])` tells the compiler that we are dealing with an array.

Accessing Elements of an Array

Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, the number in the brackets following the array name. This number specifies

the element's position in the array. All the array elements are numbered, starting with 0. Thus, marks[2] is not the second element of the array, but the third. In our program we are using the variable *i* as a subscript to refer to various elements of the array. This variable can take different values and hence can refer to the different elements in the array in turn. This ability to use variables as subscripts is what makes arrays so useful.

Entering Data into an Array

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ ) { printf ( "\nEnter marks " ) ; scanf ( "%d", &marks[i] ) ; }
```

The for loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, *i* has a value 0, so the scanf() function will cause the value typed to be stored in the array element marks[0], the first element of the array. This process will be repeated until *i* becomes 29. This is last time through the loop, which is a good thing, because there is no array element like marks[30].

Reading Data from an Array

The balance of the program reads the data back out of the array and uses it to calculate the average. The for loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called sum. When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i <= 29 ; i++ )
    sum = sum + marks[i] ;
avg = sum / 30 ;
printf ( "\nAverage marks = %d", avg ) ;
```

To fix our ideas, let us revise whatever we have learnt about arrays:

- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1

less than the size of the array.

- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory locations. This is a very important point which we would discuss in more detail later on.

Array Initialization:

So far we have used arrays that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how to initialize an array while declaring it. Following are a few examples that demonstrate this.

```
int num[6]={ 2, 4, 12, 5, 45, 5 };  
int n[ ]={ 2, 4, 12, 5, 45, 5 };  
float press[ ]={ 12.3, 34.2 -23.4, -11.3 };
```

Note the following points carefully:

Till the array elements are not given any specific values, they are supposed to contain garbage values.

- If the array is initialized where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.
- If the array is initialized where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

Array Elements in Memory:

Consider the following array declaration:

```
int arr[8] ;
```

What happens in memory when we make this declaration? 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers. And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be auto. If the storage class is declared to be static then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations.

8.2 Two dimensional (2D) arrays

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns. Before we discuss more about two Dimensional array lets have a look at the following C program.

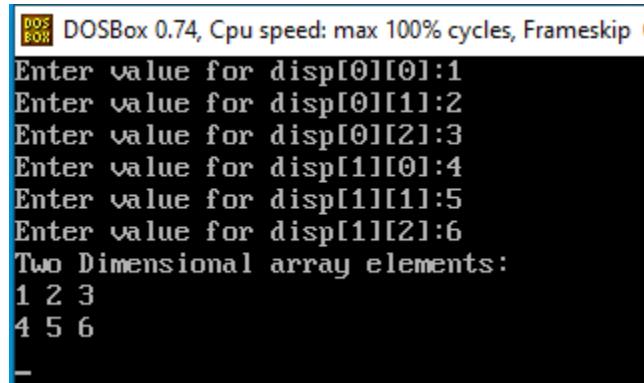
Simple Two dimensional (2D) Array example:

For now don't worry how to initialize a two dimensional array, we will discuss that part later. This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```
#include<stdio.h>
#include<conio.h>
int main(){
clrscr();
/* 2D array declaration*/
int disp[2][3];
/*Counter variables for the loop*/
int i, j;
for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
        printf("Enter value for disp[%d][%d]:", i, j);
        scanf("%d", &disp[i][j]);
    }
}
//Displaying array elements
printf("Two Dimensional array elements:\n");
for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
        printf("%d ", disp[i][j]);
        if(j==2){
            printf("\n");
        }
    }
}
```

```
    }  
}  
getch();  
return 0;  
}
```

Output:



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
-

Initialization of 2D Array:

There are two ways to initialize a two dimensional arrays during declaration.

```
int disp[2][4] = {  
    {10, 11, 12, 13},  
    {14, 15, 16, 17}  
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However, that's not the case with 2D array, you must always specify the second dimension even if you are

specifying elements during the declaration. Let's understand this with the help of few examples.

```
/* Valid declaration*/  
int abc[2][2] = { 1, 2, 3 ,4 }  
/* Valid declaration*/  
int abc[][2] = { 1, 2, 3 ,4 }  
/* Invalid declaration – you must specify second dimension*/  
int abc[] [] = { 1, 2, 3 ,4 }  
/* Invalid because of the same reason mentioned above*/  
int abc[2][] = { 1, 2, 3 ,4 }
```

How to store user input data into 2D array

We can calculate how many elements a two dimensional array can have by using this formula:

The array arr[n1][n2] can have $n1 \times n2$ elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **first subscript** value as 5 and **second subscript** value as 4. So the array abc[5][4] can have $5 \times 4 = 20$ elements.

To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be abc[0][0], abc[0][1], abc[0][2]...so on.

```
#include<stdio.h>  
#include<conio.h>  
int main(){  
clrscr();  
/* 2D array declaration*/  
int abc[5][4];  
/*Counter variables for the loop*/  
int i, j;
```

```

for(i=0; i<5; i++) {
    for(j=0;j<4;j++) {
        printf("Enter value for abc[%d][%d]:", i, j);
        scanf("%d", &abc[i][j]);
    }
}
getch();
return 0;
}

```

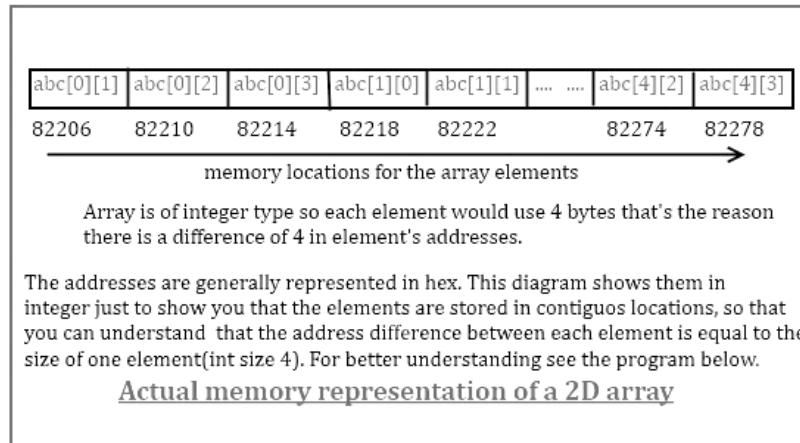
In above example, I have a 2D array abc of integer type.
Conceptually you can visualize the above array like this:

2D array conceptual memory representation

Second subscript				
first subscript	abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
	abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
	abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
	abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
	abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

Here my array is abc [5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

However the actual representation of this array in memory would be something like this:



Strings

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

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

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

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

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C.

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

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

Let us try to print above mentioned string:

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

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

    printf("Greeting message: %s\n", greeting );

    getch();
}
```

Output:

Greeting message: Hello

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

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

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

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
int main ()
{
clrscr();
char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
int len ;

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

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

/* total length of str1 after concatenation */
len = strlen(str1);

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

getch();

return 0;
}
```

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

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

- C Strings are nothing but array of characters ended with null character ('\0').
- This null character indicates the end of the string.
- Strings are always enclosed by double quotes. Whereas, character is enclosed by single quotes in C.

Example for C string:

```
char string[20] = { 'f' , 'r' , 'e' , 's' , 'h' , '2' , 'r' , 'e' , 'f' , 'r' ,  
'e' , 's' , 'h' , '\0'}; (or)
```

- char string[20] = "Pakistan"; (or)
- char string [] = "Pakistan";
- Difference between above declarations are, when we declare char as "string[20]", 20 bytes of memory space is allocated for holding the string value.
- When we declare char as "string[]", memory space will be allocated as per the requirement during execution of the program.

Example program for C string:

```
#include <stdio.h>  
  
#include<conio.h>  
  
int main ()  
{  
    clrscr();  
  
    char string[20] = "Pakistan";  
  
    printf("The string is : %s \n", string );  
  
    getch();  
  
    return 0;
```

```

}
```

Output:
The string is : Pakistan

String Functions:

- String.h header file supports all the string methods in C language. All the string methods are given below.

S.no	String methods	Description
1	<u>strcat ()</u>	Concatenates str2 at the end of str1.
2	<u>strncat ()</u>	appends a portion of string to another
3	<u>strcpy ()</u>	Copies str2 into str1
4	<u>strncpy ()</u>	copies given number of characters of one string to another
5	<u>strlen ()</u>	gives the length of str1.
6	<u>strcmp ()</u>	Returns 0 if str1 is same as str2. Returns <0 if str1 <

		str2. Returns >0 if str1 > str2.
7	<u>strcmpi()</u>	Same as strcmp() function. But, this function negotiates case. “A” and “a” are treated as same.
8	<u>strchr ()</u>	Returns pointer to first occurrence of char in str1.
9	<u>strrchr ()</u>	last occurrence of given character in a string is found
10	<u>strstr ()</u>	Returns pointer to first occurrence of str2 in str1.
11	<u>strrstr ()</u>	Returns pointer to last occurrence of str2 in str1.
12	<u>strdup ()</u>	duplicates the string
13	<u>strlwr ()</u>	converts string to lowercase
14	<u>strupr ()</u>	converts string to uppercase
15	<u>strrev ()</u>	reverses the given string

16	<u>strset ()</u>	sets all character in a string to given character
17	<u>strnset ()</u>	It sets the portion of characters in a string to given character
18	<u>strtok ()</u>	tokenizing given string using delimiter

strcat() function:

strcat() function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for strcat() function is given below.

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

Example:

strcat (str2, str1); - str1 is concatenated at the end of str2.
 strcat (str1, str2); - str2 is concatenated at the end of str1.

As you know, each string in C is ended up with null character ('\0').

In strcat() operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strcat() operation.

Example program for strcat() function in C:

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

```
#include <conio.h>
```

```
#include <string.h>

int main( )
{
    clrscr();
    char source[ ] = " Pakistan" ;
    char target[ ]= " C tutorial" ;

    printf ( "\nSource string = %s", source ) ;
    printf ( "\nTarget string = %s", target ) ;
    strcat ( target, source ) ;
    printf ( "\nTarget string after strcat( ) = %s", target ) ;
    getch();
}
```

Output:

Source string	=	Pakistan
Target string	=	C tutorial
Target string after strcat()	=	C tutorial
Pakistan		

strcpy() function:

strcpy() function copies contents of one string into another string. Syntax for strcpy function is given below.

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

Example:

strcpy (str1, str2) – It copies contents of str2 into str1.

strcpy (str2, str1) – It copies contents of str1 into str2.

- If destination string length is less than source string, entire source string value won't be copied into destination string.
- For example, consider destination string length is 20 and source string Length is 30. Then, only 20 characters from source string will be copied into destination string and remaining 10 characters won't be copied and will be truncated.

Example program for strcpy() function in C

```
#include <stdio.h>
#include <string.h>
int main()
{
    char source[ ] = " Pakistan ";
    char target[20]= "";
    printf ( "\nsource string = %s", source );
    printf ( "\ntarget string = %s", target );
    strcpy ( target, source );
    printf ( "\ntarget string after strcpy( ) = %s", target );
    return 0;
}
```

Output:

```
source string = Pakistan  
target string =  
target string after strcpy( ) = Pakistan
```

strlen() function:

- `strlen()` function in C gives the length of the given string. Syntax for

`strlen()` function is given below.

```
size_t strlen ( const char * str );
```

- `strlen()` function counts the number of characters in a given string and

returns the integer value.

- It stops counting the character when null character is found. Because

, null character indicates the end of the string in C.

Example program for `strlen()` function in C:

```
#include <stdio.h>  
  
#include<conio.h>  
  
#include <string.h>  
  
int main( )  
{  
    clrscr();  
  
    int len;
```

```
char array[20] = "Pakistan" ;  
len = strlen(array) ;  
printf ( "\string length = %d \n" , len ) ;  
getch();  
return 0;  
}
```

Output:

string length = 8

END OF CHAPTER 08

Exercise:

Multiple Choice Questions.

- 1. What is an Array in C language?**
 - A) A group of elements of same data type.
 - B) An array contains more than one element
 - C) Array elements are stored in memory in continuous or contiguous locations.
 - D) All the above.
- 2. What are the Types of Arrays?**
 - A) int, long, float, double
 - B) struct, enum
 - C) char
 - D) All the above
- 3. An array Index starts with.?**
 - A) -1
 - B) 0
 - C) 1
 - D) 2

4. What is the output of C Program.?

```
int main() { int a[] = {1,2,3,4}; int b[4] = {5,6,7,8};  
    printf("%d,%d", a[0], b[0]); }
```

- A) 1,5
- B) 2,6
- C) 0 0
- D) Compiler error

5. What is an array Base Address in C language?

- A) Base address is the address of 0th index element.
- B) An array b[] base address is &b[0]
- C) An array b[] base address can be printed with printf("%d", b);
- D) All the above

6. Which of these best describes an array?

- A) A data structure that shows a hierarchical behaviour
- B) Container of objects of similar types
- C) Arrays are immutable once initialised
- D) Array is not a data structure

7. How do you initialize an array in C?

- A) int arr[3] = (1,2,3);
- B) int arr(3) = {1,2,3};
- C) int arr[3] = {1,2,3};
- D) int arr(3) = (1,2,3);

8. What is the output of the following piece of code?

```
public class array  
{  
    public static void main(String args[])  
    {  
        int []arr = {1,2,3,4,5};  
        System.out.println(arr[2]);  
        System.out.println(arr[4]);  
    }  
}
```

- A) 3 and 5
- B) 5 and 3
- C) 2 and 4
- D) 4 and 2

9. When does the **ArrayIndexOutOfBoundsException** occur?

- A) Compile-time
- B) Run-time
- C) Not an error
- D) Not an exception at all

10. What are the advantages of arrays?

- A) Objects of mixed data types can be stored
- B) Elements in an array cannot be sorted
- C) Index of first element of an array is 1
- D) Easier to store elements of same data type

11. What are the disadvantages of arrays?

- A) Data structure like queue or stack cannot be implemented
- B) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
- C) Index value of an array can be negative
- D) Elements are sequentially accessed

12. Assuming int is of 4bytes, what is the size of int arr[15];?

- A) 15
- B) 19
- C) 11
- D) 60

13. In general, the index of the first element in an array is _____

- A) 0
- B) -1
- C) 2
- D) 1

14. Elements in an array are accessed _____

- A) randomly
- B) sequentially
- C) exponentially
- D) logarithmically

15. What is a String in C Language.?

- A) String is a new Data Type in C
- B) String is an array of Characters with null character as the last element of array.
- C) String is an array of Characters with null character as the first element of array
- D) String is an array of Integers with 0 as the last element of array.

16. Choose a correct statement about C String.

- ```
char ary[]="Hello..!";
```
- A) Character array, ary is a string.
  - B) ary has no Null character at the end
  - C) String size is not mentioned
  - D) String can not contain special characters.

**17. What is the output of C Program with Strings.?**

```
int main()
{
 char ary[]="Discovery Channel";
 printf("%s",ary);
 return 0;
}
```

- A) D
- B) Discovery Channel
- C) Discovery
- D) Compiler error

**18. What is the maximum length of a C String.?**

- A) 32 characters
- B) 64 characters
- C) 256 characters
- D) None of the above

**19. Strcat() function adds null character.**

- A. Only if there is space
- B. Always
- C. Depends on the standard
- D. Depends on the compiler

**20. Any function working with String knows the String has ended when it**

**encounters**

- A. null character
- B. Empty space
- C. "\1"
- D. Pointer

**Answers**

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 1. d  | 2. d  | 3. b  | 4. b  | 5.d   |
| 6. b  | 7. c  | 8. a  | 9. b  | 10. d |
| 11. b | 12. d | 13. a | 14. a | 15 b  |
| 16 a  | 17 b  | 18 d  | 19 b  | 20 a  |

**Short Questions with Answers****1. Define array?**

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

**2. Define 2D array?**

The two dimensional (2D) array in C programming is also known as matrix.

A matrix can be represented as a table of rows and columns.

**3. How to declare simple array explain with example?**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows,  
type arrayName [ arraySize ];

This is called a single-dimensional array. The array size must be an integer, constant, greater than zero and type can be any valid C data type. For example,

to declare a 10-element array called balance of type double, use this statement.

```
double balance[10];
```

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

#### 4. How to initialize array?

You can initialize an array in C either one by one or using a single statement as follows,  
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

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

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write,  
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

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

#### 5. How to store data in 2D array?

We can calculate how many elements a two dimensional array can have by using this formula: The array arr[n1][n2] can have  $n1 * n2$  elements. The array

that we have in the example below is having the dimensions 5 and 4.

These

dimensions are known as subscripts. So this array has **first subscript** value as 5

and **second subscript** value as 4. So the array abc[5][4] can have  $5 * 4 = 20$  elements.

#### 6. How to initialize 2D array?

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
 {10, 11, 12, 13},
 {14, 15, 16, 17}
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

#### 7. define string?

In C programming, a string is a sequence of characters terminated with a null

character \0.

**8. Write declaration syntax of string?**

char str\_name[size];

**9. Define strlen function?**

strlen() returns the number of bytes rather than the number of characters in a string.

**10. Write the syntax of strcat() function?**

char \*strcat(char \*destination, const char \*source)

**Long Question:**

1. Explain strcat() function with example?
2. Write a program that converts all lowercase characters in a given string to its equivalent uppercase character.
3. Write a program to sort a set of names stored in an array in alphabetical order.
4. Write a program to delete all vowels from a sentence. Assume that the sentence is not more than 80 characters long.
5. Explain 2d array with example?

# POINTERS

09

## 9.1 Introduction to pointers

Before we discuss about **pointers in C**, let's take a simple example to understand what we mean by the address of a variable.

### **A simple example to understand how to access the address of a variable without pointers:**

In this program, we have a variable num of **int** type. The value of num is 10 and this value must be stored somewhere in the memory. A memory space is allocated for each variable that holds the value of that variable, this memory space has an address. For example we live in a house and our house has an address, which helps other people to find our house. The same way the value of the variable is stored in a memory address, which helps the C program to find that value when it is needed.

### **Program:**

```
#include <stdio.h>
int main()
{
 int num = 10;
 printf("Value of variable num is: %d", num);
 /* To print the address of a variable we use %p
 * format specifier and ampersand (&) sign just
 * before the variable name like &num.
 */
 printf("\nAddress of variable num is: %p"; &num);
 return 0;
}
```

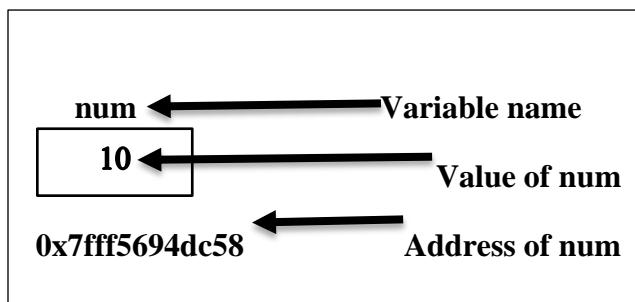
### Output:

Value of variable num is: 10

Address of variable num is: 0x7fff5694dc58

So let's say the address assigned to variable num is 0x7fff5694dc58, which means whatever value we would be assigning to num should be stored at the location: 0x7fff5694dc58.

See the diagram below.



addresses/memory-locations of other variables. A Pointer in C is used to allocate memory dynamically i.e. at run time. **Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.**

The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

An \* (asterisk) symbol followed by the variable name is used for designating variables as pointers.

#### 9.1.1 Declaring Pointers

Like variables, pointers have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form.

Syntax in C:

```
datatype *variable_name;
```

Here

- **data\_type** is the pointer's base type of C's variable types and indicates the type of the variable that the pointer points to.
- **The asterisk** (\*: the same asterisk used for multiplication) which is indirection operator, declares a pointer.

### Some valid pointer declarations are:

```
int *pt1; /*pt1 is a pointer to an integer; it holds the address of an integer variable*/
```

```
char *pt2; /*pt2 is a pointer to a character; it holds the address of a character variable*/
```

```
double *pt3; /*pt3 is a pointer to a double; it holds the address of a double variable */
```

```
float *pt4; /*pt4 is a pointer to a floating point; it holds the address of a floating-point variable*/
```

**If you need a pointer to store the address of integer variable then the data type of the pointer should be int.** Same case is with the other data types.

By using \* operator we can access the value of a variable through a pointer.

#### For example:

```
double a = 10;
```

```
double *p;
```

```
p = &a;
```

\*p would give us the value of the variable 'a'. The following statement would display 10 as output.

```
printf("%d", *p);
```

Similarly if we assign a value to \*pointer like this:

```
*p = 200;
```

It would change the value of variable 'a'. The statement above will change the value of 'a' from 10 to 200.

### 9.1.2 Initializing Pointers

After declaring a pointer, we initialize it like standard variables with a variable address. If pointers are not uninitialized and used in the program, the results are unpredictable and potentially disastrous.

To get the address of a variable, we use the ampersand (&)operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax.

**pointer = &variable;**

### **Example of Pointers in C**

```
#include <stdio.h>
int main()
{
 int a=10; //variable declaration
 int *p; //pointer variable declaration
 p=&a; //store address of variable a in pointer p
 printf("Address stored in a variable p is:%x\n",p);
 //accessing the address
 printf("Value stored in a variable p is:%d\n",*p);
 //accessing the value
 return 0;
}
```

#### **Output:**

Address stored in a variable p is:60ff08

Value stored in a variable p is:10

### **9.1.3 How Pointers work in C Programming Language**

We must understand the use of two operators (& and \*) for using pointers in C.

#### **Unary Ampersand (&) Operator**

The unary operator, & is used for getting the address of the variable. If you use '&' before a variable name, it returns the address of that variable. For example, &y will return the address of the variable y.

**/\*Example C Program\*/**

1. #include<stdio.h>
2. #include<conio.h>
3. int main()
4. {
5. int y;

```

6. printf("%p", &y); /*prints the address of y */
7. return 0;
8. }
```

## Unary Asterisk Operator (\*)

There are two ways to use the \* operator in C language.

**1. Pointer Declaration-** When a pointer is declared, there must be an asterisk operator placed before the pointer name.

```

/*Example C Program*/
#include<stdio.h>
void main()
{
 int y=5;
 int *pt; /* when * is used, pt is considered as a pointer variable */
 pt = &y; /* pt holds the address of the variable y */
}
```

**2. Accessing Pointer Variables-** We can also use the \* operator for accessing the value of stored in any address/memory-location.

```

/*Example C Program*/
#include<stdio.h>
void main()
{
 int x = 6;
 int *pt;
 pt = &x;
 printf("The value of x = %d \n", *pt); /* prints 6 as x value*/
 printf("The address of x = %p\n", pt); /* returns the address of variable x */
 *pt= 10;
 printf("After changing the value of pt, the new value is %d", *pt);
}
```

**Example of Pointer demonstrating the use of & and \***

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

```
int main()
{
 /* Pointer of integer type, this can hold the
 * address of a integer type variable.
 */
 int *p;
 int var = 10;
 /* Assigning the address of variable var to the pointer
 * p. The p can hold the address of var because var is * an integer type
 * variable.
 */
 p= &var;

 printf("Value of variable var is: %d", var);
 printf("\nValue of variable var is: %d", *p);
 printf("\nAddress of variable var is: %p", &var);
 printf("\nAddress of variable var is: %p", p);
 printf("\nAddress of pointer p is: %p", &p);
 return 0;
}
```

### Output:

Value of variable var is: 10  
Value of variable var is: 10  
Address of variable var is: 0x7fff5ed98c4c  
Address of variable var is: 0x7fff5ed98c4c  
Address of pointer p is: 0x7fff5ed98c50

#### 9.1.4 Direct and Indirect Access Pointers

**In C, there are two equivalent ways to access and manipulate a variable content**

- **Direct access:** we use directly the variable name
- **Indirect access:** we use a pointer to the variable

**Let's understand this with the help of program below:**

```
#include <stdio.h> /* Declare and initialize an int variable */
int var = 1; /* Declare a pointer to int */
int *ptr;
int main(void)
{
 ptr = &var; /* Initialize ptr to point to var */
 printf("\nDirect access, var = %d", var); /* Access var directly and
 indirectly */
 printf("\n\nThe address of var = %d", &var); /* Display the address of var two ways */
 printf("\nThe address of var = %d\n", ptr); /*change the content of var through the pointer*/
 *ptr=48;
 printf("\nIndirect access, var = %d", *ptr);
 return 0;
}
```

### **Output:**

Direct access, var = 1

Indirect access, var = 1

The address of var = 4202496

The address of var = 4202496

Indirect access, var = 48

### **9.1.5 Types of Pointers**

Most commonly used types of pointers are:

#### **1. Null Pointer**

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

**Following program illustrates the use of a null pointer:**

```
#include <stdio.h>
int main()
{
 int *p = NULL; //null pointer
 printf("The value inside variable p is:\n%x",p);
 return 0;
}
```

### **Output:**

The value inside variable p is:  
0

## **2. Void Pointer**

In C programming, a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.

### **Following program illustrates the use of a Void Pointer:**

```
#include <stdio.h>
int main()
{
 void *p = NULL; //void pointer
 printf("The size of pointer is:%d\n",sizeof(p));
 return 0;
}
```

### **Output:**

The size of pointer is:4

## **3. Wild Pointer**

A pointer is said to be a wild pointer if it is not being initialized to anything. These types of pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

**Following program illustrates the use of wild pointer:**

```
#include <stdio.h>
int main()
{
 int *p; //wild pointer
 printf("\n%d", *p);
 return 0;
}
```

### **Output:**

```
timeout: the monitored command dumped core
sh: line 1: 95298 Segmentation fault timeout 10s main
```

### **9.1.6 Pointer Arithmetic**

If you want to have complete knowledge of pointers, pointer arithmetic is very important to understand. In this topic we will study how the memory addresses change when you increment a pointer.

#### **a) 16 bit Machine**

In a 16 bit machine, size of all types of pointer, be it **int\***, **float\***, **char\*** or **double\*** is always **2 bytes**. But when we perform any arithmetic function like increment on a pointer, changes occur as per the size of their primitive data type.

#### **Size of datatypes on 16-bit Machine:**

| Type              | Size (in bytes) |
|-------------------|-----------------|
| int or signed int | 2               |
| char              | 1               |
| long              | 4               |
| float             | 4               |

|        |    |
|--------|----|
| double | 8  |
| long   | 10 |
| double |    |

### Examples for Pointer Arithmetic

Now let's take a few examples and understand this more clearly.

```
int* i;
i++;
```

In the above case, pointer will be of 2 bytes. And when we increment it, it will increment by 2 bytes because **int** is also of 2 bytes.

```
float* i;
i++;
```

In this case, size of pointer is still 2 bytes initially. But now, when we increment it, it will increment by 4 bytes because **float** datatype is of 4 bytes.

```
double* i;
i++;
```

Similarly, in this case, size of pointer is still 2 bytes. But now, when we increment it, it will increment by 8 bytes because its data type is **double**.

### b) 32 bit Machine

The concept of pointer arithmetic remains exact same, but the size of pointer and various datatypes is different in a 32 bit machine. Pointer in 32 bit machine is of **4 bytes**.

Following is a table for

#### Size of datatypes on 32-bit Machine :

| Type                 | Size (in bytes) |
|----------------------|-----------------|
| int or<br>signed int | 4               |
| char                 | 2               |

|        |    |
|--------|----|
| long   | 8  |
| float  | 8  |
| double | 16 |

Note: We cannot add two pointers. This is because pointers contain addresses, adding two addresses makes no sense, because you have no idea what it would point to. But we can subtract two pointers. This is because difference between two pointers gives the number of elements of its data type that can be stored between the two pointers.

```
#include <stdio.h>

int main()
{
 int m = 5, n = 10, o = 0;
 int *p1;
 int *p2;
 int *p3;
 p1 = &m; //printing the address of m
 p2 = &n; //printing the address of n
 printf("p1 = %d\n", p1);
 printf("p2 = %d\n", p2);
 o = *p1+*p2;
 printf("*p1+*p2 = %d\n", o);//point 1
 p3 = p1-p2;
 printf("p1 - p2 = %d\n", p3); //point 2
 p1++;
 printf("p1++ = %d\n", p1); //point 3
 p2--;
 printf("p2-- = %d\n", p2); //point 4
 //Below line will give ERROR
 printf("p1+p2 = %d\n", p1+p2); //point 5
 return 0;
}
```

## Program for pointer arithmetic

### Output

p1 = 2680016

p2 = 2680012

\*p1+\*p2 = 15

p1-p2 = 1

p1++ = 2680020

p2-- = 2680008

### Explanation of the above program:

1. **Point 1:** Here, \* means 'value at the given address'. Thus, it adds the value of m and n which is 15.
2. **Point 2:** It subtracts the addresses of the two variables and then divides it by the size of the pointer datatype (here integer, which has size of 4 bytes) which gives us the number of elements of integer data type that can be stored within it.
3. **Point 3:** It increments the address stored by the pointer by the size of its datatype
4. **Point 4:** It decrements the address stored by the pointer by the size of its datatype (here 4).
5. **Point 5:** Addition of two pointers is not allowed.

### Priority operation (precedence)

When working with pointers, we must observe the following priority rules:

- The operators \* and & have the same priority as the unary operators (the negation!, the incrementation++, decrement--).
- In the same expression, the unary operators \*, &, !, ++, -- are evaluated from right to left.

If a P pointer points to an X variable, then \* P can be used wherever X can be written.

**The following expressions are equivalent:**

int X = 10

int \*P = &Y;

For the above code, below expressions are true

| <b>Expression</b> | <b>Equivalent Expression</b> |
|-------------------|------------------------------|
| Y = *P + 1        | Y = X + 1                    |
| *P = *P + 10      | X = X + 10                   |
| *P += 2           | X += 2                       |
| ++*P              | ++X                          |
| (*P)++            | X++                          |

In the latter case, parentheses are needed: as the unary operators \* and ++ are evaluated from right to left, without the parentheses the pointer P would be incremented, not the object on which P points.

## 9.2 Returning Data from Functions

Pointers give greatly possibilities to 'C' functions which we are limited to return one value. With pointer parameters, our functions now can process actual data rather than a copy of data.

In order to modify the actual values of variables, the calling statement passes addresses to pointer parameters in a function.

### 9.2.1 Function Pointers

As we know by definition that pointers point to an address in any memory location, they can also point to at the beginning of executable code as functions in memory.

### How to declare a function pointer?

#### Syntax in C

```
function_return_type(*Pointer_name)(function argument list)
```

#### For example:

```
double (*p2f)(double, char)
```

Here double is a return type of function, p2f is name of the function pointer and (double, char) is an argument list of this function. Which means the first argument of this function is of double type and the second argument is char type. You have to remember that the parentheses around (\*function\_name) are important because without them, the compiler will think the function\_name is returning a pointer of return\_type.

Let's understand this with the help of an example: Here we have a function sum that calculates the sum of two numbers and returns the sum. We have created a pointer f2p that points to this function, we are invoking the function using this function pointer f2p.

#### Example-1:

```
int sum (int num1, int num2)
{
 return num1+num2;
}
int main()
{
 /* The following two lines can also be written in a single
 * statement like this: void (*fun_ptr)(int) = &fun;
 */
 int (*f2p) (int, int);
 f2p = sum;
 //Calling function using function pointer
 int op1 = f2p(10, 13);
```

```
//Calling function in normal way using function name
int op2 = sum(10, 13);

printf("Output1: Call using function pointer:
%d",op1);
printf("\nOutput2: Call using function name: %d", op2);

return 0;
}
```

**Output:**

Output1: Call using function pointer: 23  
Output2: Call using function name: 23

**Some points regarding function pointer:**

1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

```
void (*fun_ptr)(int) = &fun;
```

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

```
void (*fun_ptr)(int) = fun;
```

After defining the function pointer, we have to assign it to a function. For example, the next program declares an ordinary function, defines a function pointer, assigns the function pointer to the ordinary function and after that calls the function through the pointer:

**Program:**

```
#include <stdio.h>
void P_function (int times); /* function */
int main() {
```

```
void (*function_ptr)(int); /* function pointer Declaration */
function_ptr = P_function; /* pointer assignment */
function_ptr (3); /* function call */

return 0;
}

void P_function (int times) {
 int k;
 for (k = 0; k < times; k++)
 printf("Pass\n");
}
```

### Output:

Pass  
Pass  
Pass

### Explanation of the program:

1. We define and declare a standard function which prints a “Pass” text k times indicated by the parameter times when the function is called.
2. We define a pointer function (with its special declaration) which takes an integer parameter and doesn't return anything.
3. We initialize our pointer function with the P\_function which means that the pointer points to the P\_function().
4. Rather than the standard function calling by taping the function name with arguments, we call only the pointer function by passing the number 3 as arguments, and that's it!

Keep in mind that the function name points to the beginning address of the executable code like an array name which points to its first element.

Therefore, instructions like `function_ptr = &P_function` and `(*funptr)(3)` are correct.

**NOTE:** It is not important to insert the address operator & and the indirection operator \* during the function assignment and function call.

## 9.2.2 Pointers as Function Arguments

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will affect the original variable.

### Example-1: Swapping two numbers using Pointer

```
void swap (int *a, int *b);
int main() {
 int m = 25;
 int n = 100;
 printf("m is %d, n is %d\n", m, n);
 swap(&m, &n);
 printf("m is %d, n is %d\n", m, n);
 return 0;
}
void swap (int *a, int *b) {
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;}
}
```

### Output:

m is 25, n is 100

m is 100, n is 25

The program swaps the actual variables values because the function accesses them by address using pointers. Here we will discuss the program process:

1. We declare the function responsible for swapping the two variable values, which takes two integer pointers as parameters and returns any value when it is called.
2. In the main function, we declare and initialize two integer variables ('m' and 'n') then we print their values respectively.

3. We call the swap() function by passing the address of the two variables as arguments using the ampersand symbol. After that, we print the new swapped values of variables.
4. Here we define the swap() function content which takes two integer variable addresses as parameters and declare a temporary integer variable used as a third storage box to save one of the value variables which will be put to the second variable.
5. Save the content of the first variable pointed by 'a' in the temporary variable.
6. Store the second variable pointed by b in the first variable pointed by a.
7. Update the second variable (pointed by b) by the value of the first variable saved in the temporary variable.

## **Example-2 : Passing Pointer to a Function in C Programming**

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable.

### **Program:**

```
#include <stdio.h>
void salaryhike(int *var, int b)
{
 *var = *var+b;
}
int main()
{
 int salary=0, bonus=0;
 printf("Enter the employee current salary:");
 scanf("%d", &salary);
 printf("Enter bonus:");

}
```

```
 scanf("%d", &bonus);
 salaryhike(&salary, bonus);
 printf("Final salary: %d", salary);
 return 0;
}
```

**Output:**

Enter the employee current salary:10000

Enter bonus:2000

Final salary: 12000

### 9.2.3 Functions returning Pointer variables

A function can also **return** a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

**Program:**

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{
 int a = 15;
 int b = 92;
 int *p;
 p = larger(&a, &b);
 printf("%d is larger", *p);
}
int* larger(int *x, int *y)
{
 if(*x > *y)
 return x;
 else
 return y;
}
```

**Output:**

92 is larger

**Safe ways to return a valid Pointer.**

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
2. Or, use **static local variables** inside the function and return them. As static variables have a lifetime until the **main()** function exits, therefore they will be available throughout the program.

**9.2.4 Array of Function Pointers**

An array of function pointers can play a switch or an if statement role for making a decision, as in this program:

```
#include <stdio.h>
int sum(int num1, int num2);
int sub(int num1, int num2);
int mult(int num1, int num2);
int div(int num1, int num2);
int main()
{ int x, y, choice, result;
 int (*ope[4])(int, int);
 ope[0] = sum;
 ope[1] = sub;
 ope[2] = mult;
 ope[3] = div;
 printf("Enter two integer numbers: ");
 scanf("%d%d", &x, &y);
 printf("Enter 0 to sum, 1 to subtract, 2 to multiply, or 3 to divide: ");
 scanf("%d", &choice);
 result = ope[choice](x, y);
 printf("%d", result);
return 0;
}
```

```
int sum(int x, int y) {return(x + y);}
int sub(int x, int y) {return(x - y);}
int mult(int x, int y) {return(x * y);}
int div(int x, int y) {if (y != 0) return (x / y); else return 0;}
```

Enter two integer numbers: 13 48

Enter 0 to sum, 1 to subtract, 2 to multiply, or 3 to divide: 2 624

**Here, we discuss the program details:**

1. We declare and define four functions which take two integer arguments and return an integer value. These functions add, subtract, multiply and divide the two arguments regarding which function is being called by the user.
2. We declare 4 integers to handle operands, operation type, and result respectively. Also, we declare an array of four function pointer. Each function pointer of array element takes two integers parameters and returns an integer value.
3. We assign and initialize each array element with the function already declared. For example, the third element which is the third function pointer will point to multiplication operation function.
4. We seek operands and type of operation from the user typed with the keyboard.
5. We called the appropriate array element (Function pointer) with arguments, and we store the result generated by the appropriate function.

The instruction `int (*ope[4])(int, int);` defines the array of function pointers. Each array element must have the same parameters and return type.

The statement `result = ope[choice](x, y);` runs the appropriate function according to the choice made by the user. The two entered integers are the arguments passed to the function.

### 9.2.5 Functions using void Pointers

Void pointers are used during function declarations. We use a void \* return type permits to return any type. If we assume that our parameters do not change when passing to a function, we declare it as const.

**For example:**

```
void * cube (const void *);
```

**Consider the following program:**

```
#include <stdio.h>
void* cube (const void* num);
int main() {
 int x, cube_int;
 x = 4;
 cube_int = cube (&x);
 printf("%d cubed is %d\n", x, cube_int);
 return 0; }

void* cube (const void *num) {
 int result;
 result = (*(int *)num) * (*(int *)num) * (*(int *) num);
 return result;}
```

**Output:**

4 cubed is 64

**Here, we discuss the program details:**

1. We define and declare a function that returns an integer value and takes an address of unchangeable variable without a specific data type. We calculate the cube value of the content variable (x) pointed by the num pointer, and as it is a void pointer, we have to type cast it to an integer data type using a specific notation (\* datatype) pointer, and we return the cube value.
2. We declare the operand and the result variable. Also, we initialize our operand with value "4."

3. We call the cube function by passing the operand address, and we handle the returning value in the result variable

### 9.3 Pointers and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array **arr**,

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

Assuming that the base address of **arr** is 1000 and each integer requires two bytes, the five elements will be stored as follows:

|         |        |        |        |        |
|---------|--------|--------|--------|--------|
|         |        |        |        |        |
| element | arr[0] | arr[1] | arr[2] | arr[3] |
| Address | 1000   | 1002   | 1004   | 1006   |

arr[4]      1008

Here variable **arr** will give the base address, which is a constant pointer pointing to the first element of the array, **arr[0]**. Hence **arr** contains the address of **arr[0]** i.e **1000**. In short, **arr** has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array. **arr** is equal to **&arr[0]** by default.

We can also declare a pointer of type **int** to point to the array **arr**.

```
int *p;
p = arr;
// or,
p = &arr[0]; //both the statements are equivalent.
```

Now we can access every element of the array **arr** using **p++** to move from one element to another.

**NOTE:** You cannot decrement a pointer once incremented. **p--** won't work.

Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose p is a pointer that currently points to the memory location 0 if we perform following addition operation,  $p+1$

then it will execute in this manner:



Since p currently points to the location 0 after adding 1, the value will become 1, and hence the pointer will point to the memory location 1.

Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

### **Example-1:**

```

#include <stdio.h>
int main()
{
 int a[5]={1,2,3,4,5}; //array initialization
 int *p; //pointer declaration
 /*the ptr points to the first element of the array*/

 p=a; /*We can also type simply ptr==&a[0] */

 printf("Printing the array elements using pointer \n");
 for(int i=0;i<5;i++) //loop for traversing array elements
 {
 printf("\n%x",*p); //printing array elements
 p++; //incrementing to the next element, you can also write p=p+1
 }
 return 0;
}

```

**Output:**

1  
2  
3  
4  
5

In the above program, the pointer `*p` will print all the values stored in the array one by one. We can also use the Base address (`a` in above case) to act as a pointer and print all the values.

Replacing the `printf("%d", *p);` statement of above example, with below mentioned statements. Lets see what will be the result.

`printf("%d", a[i]);` → prints the array, by incrementing index

`printf("%d", i[a] );` → this will also print elements of array

`printf("%d", a+i );` → This will print address of all the array elements

`printf("%d", *(a+i) );` → Will print value of array element.

`printf("%d", *a);` → will print value of `a[0]` only

`a++;` → Compile time error, we cannot change base address of the array.

The generalized form for using pointer with an array,  
`*(a+i)` is same as: `a[i]`

**Example-2:**

```
#include <stdio.h>
int main()
{
 /*Pointer variable*/
 int *p;

 /*Array declaration*/
 int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

 /* Assigning the address of val[0] the pointer
 * You can also write like this:
 * p = var;
 * because array name represents the address of the first
element
 */

 p = &val[0];

 for (int i = 0 ; i<7 ; i++)
 {
 printf("val[%d]: value is %d and address is %p\n", i, *p, p);
 /* Incrementing the pointer so that it points to next element
 * on every increment.
 */
 p++;
 }
 return 0;
}
```

**Output:**

```
val[0]: value is 11 and address is 0x7fff51472c30
val[1]: value is 22 and address is 0x7fff51472c34
val[2]: value is 33 and address is 0x7fff51472c38
val[3]: value is 44 and address is 0x7fff51472c42
val[4]: value is 55 and address is 0x7fff51472c46
val[5]: value is 66 and address is 0x7fff51472c50
```

val[6]: value is 77 and address is 0x7fff51472c54

**Points to Note:**

- 1) While using pointers with array, the data type of the pointer must match with the data type of the array.
- 2) You can also use array name to initialize the pointer like this:

**p = var;**

because the array name alone is equivalent to the base address of the array.

**val==&val[0];**

- 3) In the loop the increment operation(p++) is performed on the pointer variable to get the next location (next element's location), this arithmetic is same for all types of arrays (for all data types double, char, int etc.) even though the bytes consumed by each data type is different.
- 4) Note that there is a difference of 4 bytes between each element because that's the size of an integer. Which means all the elements are stored in consecutive contiguous memory locations in the memory.

In the above example I have used &val[i] to get the address of ith element of the array. We can also use a pointer variable instead of using the ampersand (&) to get the address.

**Pointer logic:**

You must have understood the logic in above code so now it's time to play with few pointer arithmetic and expressions.

if p = &val[0] which means

\*p == val[0]

(p+1) == &val[1] & \*(p+1) == val[1]

(p+2) == &val[2] & \*(p+2) == val[2]

(p+n) == &val[n] & \*(p+n) == val[n]

Using this logic we can rewrite our code in a better way like this:

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

```

int main()
{
 int *p;
 int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
 p = val;
 for (int i = 0 ; i<7 ; i++)
 {
 printf("val[%d]: value is %d and address is %p\n", i, *(p+i), (p+i));
 }
 return 0;
}

```

We don't need the `p++` statement in this program.

## Pointer to Multidimensional Array

A multidimensional array is of form, `a[i][j]`. Let's see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In `a[i][j]`, `a` will give the base address of this array, even `a + 0 + 0` will also give the base address, that is the address of `a[0][0]` element.

Here is the generalized form for using pointer with multidimensional arrays.

`*(*(a + i) + j)`

which is same as,

`a[i][j]`

## 9.4 Pointers and Strings

### 9.4.1 Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of `char` type are treated as string.

`char *str = "Hello";`

The above code creates a string and stores its address in the pointer variable `str`. The pointer `str` now points to the first character of the string "Hello". Another important thing to note here is that the string created using `char` pointer can be assigned a value at **runtime**.

`char *str;`

```
str = "hello"; //this is Legal
```

The content of the string can be printed using **printf()** and **puts()**.

```
printf("%s", str);
puts(str);
```

Notice that **str** is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator **\***.

**A string is an array of char objects, ending with a null character '\0'. We can manipulate strings using pointers. Here is an example that explains this section:**

```
#include <stdio.h>
#include <string.h>
int main()
{
 char str[]="Hello Computer";
 char *p;
 p=str;
 printf("First character is:%c\n",*p);
 p =p+1;
 printf("Next character is:%c\n",*p);
 printf("Printing all the characters in a string\n");
 p=str; //reset the pointer
 for(int i=0;i<strlen(str);i++)
 {
 printf("%c\n",*p);
 p++;
 }
 return 0;
}
```

## Output

First character is:H

Next character is:e

Printing all the characters in a string

H

e

l

l

o

C

o

m

p

u

t

e

r

Another way to deal strings is with an array of pointers like in the following program:

```
#include <stdio.h>
int main(){
 char *materials[] = { "iron", "copper", "gold"};
 printf("Please remember these materials :\n");
 int i ;
 for (i = 0; i < 3; i++) {
 printf("%s\n", materials[i]);
 }
 return 0;
}
```

### **Output:**

Please remember these materials:

iron

copper

gold

### **9.4.2 Array of Pointers**

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
```

```
 "Adnan",
```

```
 "Amna",
```

```
 "Danial"
```

```
};
```

//Now lets see same array without using pointer

```
char name[3][20] = {
```

```
 "Adnan",
```

```
 "Amna",
```

```
 "Danial"
```

```
};
```

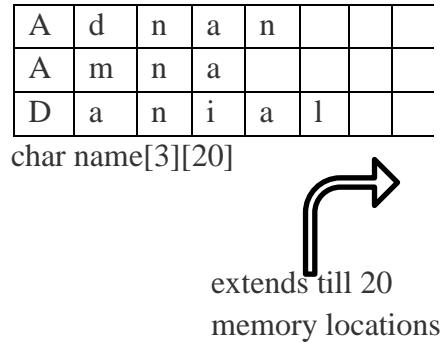
### Using Pointer



```
char* name[3]
```

Only 3 locations for pointers, which will point to the first character of their respective strings.

### Without Pointer



In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

When we say memory wastage, it doesn't mean that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguous memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

## 9.5 Double Indirection: Pointers to Pointers

Pointers are used to store the address of other variables of similar data type. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

### How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '\*' before the name of pointer.

#### Syntax:

`int **p1;`

Here, we have used two indirection operator(\*) which stores and points to the address of a pointer variable i.e, **int \***. If we want to store the address of this (double pointer) variable **p1**, then the syntax would become:

**int \*\*\*p2**

### Simple program to represent Pointer to a Pointer

```
#include <stdio.h>

int main() {
 int a = 10;

 int *p1; //this can store the address of variable a

 int **p2;
 /*

 * this can store the address of pointer variable p1 only.

 *It cannot store the address of variable 'a'

 */

 p1 = &a;

 p2 = &p1;

 printf("Address of a = %u\n", &a);

 printf("Address of p1 = %u\n", &p1);

 printf("Address of p2 = %u\n\n", &p2);

 // below print statement will give the address of 'a'

 printf("Value at the address stored by p2 = %u\n", *p2);

 printf("Value at the address stored by p1 = %d\n\n", *p1);

 printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

 /*

```

\*This is not allowed, it will give a compile time error-

```
* p2 = &a;
* printf("%u", p2);
*/
return 0;
}
```

### Output:

Address of a = 2686724

Address of p1 = 2686728

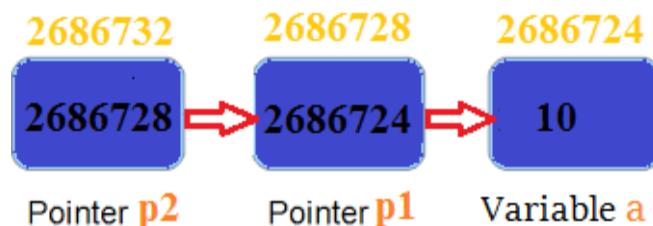
Address of p2 = 2686732

Value at the address stored by p2 = 2686724

Value at the address stored by p1 = 10

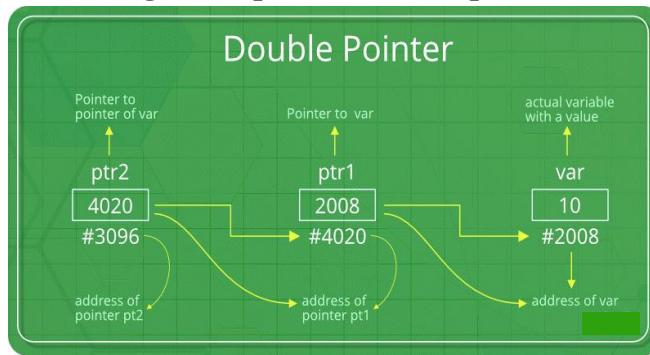
Value of \*\*p2 = 10

### Explanation of the above program

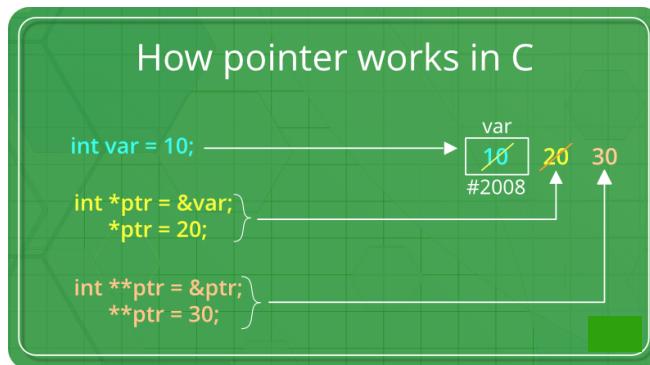


- **p1** pointer variable can only hold the address of the variable **a** (i.e Number of indirection operator(\*)-1 variable). Similarly, **p2** variable can only hold the address of variable **p1**. It cannot hold the address of variable **a**.
- \***p2** gives us the value at an address stored by the **p2** pointer. **p2** stores the address of **p1** pointer and value at the address of **p1** is the address of variable **a**. Thus, \***p2** prints address of **a**.
- \*\***p2** can be read as \*(\***p2**). Hence, it gives us the value stored at the address \***p2**. From above statement, you know \***p2** means the address of variable **a**. Hence, the value at the address \***p2** is 10. Thus, \*\***p2** prints **10**.

**Below diagram explains the concept of Double Pointers:**



The above diagram shows the memory representation of a pointer to pointer. The first pointer **ptr1** stores the address of the variable and the second pointer **ptr2** stores the address of the first pointer.



**Let us understand this more clearly with the help of the below program:**

```
#include <stdio.h>
// C program to demonstrate pointer to pointer
int main()
{
 int var = 789;
 // pointer for var
 int *ptr2;
 // double pointer for ptr2
 int **ptr1;
 // storing address of var in ptr2
 ptr2 = &var;
 // Storing address of ptr2 in ptr1
 ptr1 = &ptr2;
 // Displaying value of var using
 // both single and double pointers
 printf("Value of var = %d\n", var);
 printf("Value of var using single pointer = %d\n", *ptr2);
 printf("Value of var using double pointer = %d\n", **ptr1);

 return 0;
}
```

### **Output:**

Value of var = 789  
Value of var using single pointer = 789  
Value of var using double pointer = 789

### **Advantages of Pointers**

- Pointers are useful for accessing memory locations.

- Pointers provide an efficient way for accessing the elements of an array structure.
- Pointers are used for dynamic memory allocation as well as deallocation.
- Pointers are used to form complex data structures such as linked list, graph, tree, etc.

## Disadvantages of Pointers

- Pointers are a little complex to understand.
- Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
- If an incorrect value is provided to a pointer, it may cause memory corruption.
- Pointers are also responsible for memory leakage.
- Pointers are comparatively slower than that of the variables.
- Programmers find it very difficult to work with the pointers; therefore it is programmer's responsibility to manipulate a pointer carefully.

## Summary

- A pointer is nothing but a memory location where data is stored.
- A pointer is used to access the memory location.
- There are various types of pointers such as a null pointer, wild pointer, void pointer and other types of pointers.
- Pointers can be used with array and string to access elements more efficiently.
- We can create function pointers to invoke a function dynamically.
- Arithmetic operations can be done on a pointer which is known as pointer arithmetic.
- Pointers can also point to function which make it easy to call different functions in the case of defining an array of pointers.
- When you want to deal different variable data type, you can use a typecast void pointer.

END OF CHAPTER 09

---

## EXERCISE

### Multiple Choice Questions:

- 1. Which of the following does not initialize ptr to null (assuming variable declaration of a as int a=0;)?**
- a) int \*ptr = &a;
  - b) int \*ptr = &a - &a;
  - c) int \*ptr = a - a;
  - d) All of the mentioned

- 2. Which is an indirection operator among the following?**
- a) &
  - b) \*
  - c) ->
  - d) .

- 3. What will be the output of the following C code?**

```
1. #include <stdio.h>
2. void main()
3. {
4. int x = 0;
5. int *ptr = &x;
6. printf("%p\n", ptr);
7. }
```

- a) 5
- b) Address of 5
- c) Nothing
- d) Compile time error

- 4. Comment on the following pointer declaration:**

**int \*ptr, p;**

- a) ptr is a pointer to integer, p is not
- b) ptr and p, both are pointers to integer
- c) ptr is a pointer to integer, p may or may not be
- d) ptr and p both are not pointers to integer

**5. Comment on the following C statement.****const int \*ptr;**

- a) You cannot change the value pointed by **ptr**
- b) You cannot change the pointer **ptr** itself
- c) You May or may not change the value pointed by **ptr**
- d) You can change the pointer as well as the value pointed by it

**6. What will be the output of the following C code?**

```
#include <stdio.h>
void main()
{
 int a[3] = {1, 2, 3};
 int *p = a;
 printf("%p\t%p", p, a);
}
```

- a) Same address is printed
- b) Different address is printed
- c) Compile time error
- d) Nothing

**7. What will be the output of the following C code?**

```
#include <stdio.h>
int main()
{
 int ary[4] = {1, 2, 3, 4};
 printf("%d\n", *ary);
}
```

- a) 1
- b) Compile time error
- c) Some garbage value
- d) Undefined variable

**8. What are the different ways to initialize an array with all elements as zero?**

- a) int array[5] = {};
- b) int array[5] = {0};
- c) int a = 0, b = 0, c = 0;  
int array[5] = {a, b, c};

d) All of the mentioned

**9. What is the size of \*ptr in a 32-bit machine (Assuming initialization as int \*ptr = 10;)?**

- a) 1
- b) 2
- c) 4
- d) 8

**10. What is the correct way to declare and assign a function pointer? (Assuming the function to be assigned is "int multi(int, int);")**

- a) int (\*fn\_ptr)(int, int) = multi;
- b) int \*fn\_ptr(int, int) = multi;
- c) int \*fn\_ptr(int, int) = &multi;
- d) none of the mentioned

**11. Comment on the following C statement.**

**int (\*a)[7];**

- a) An array “a” of pointers
- b) A pointer “a” to an array
- c) A ragged array
- d) None of the mentioned

**12. Comment on the output of following code:**

```
#include <stdio.h>
main()
{
char *p = 0;
*p = 'a';
printf("value in pointer p is %c\n", *p);
}
```

- a) It will print a
- b) It will print 0
- c) Compile time error
- d) Run time error

**13. An entire array is always passed by \_\_\_ to a called function.**

- a) Call by value
- b) Call by reference
- c) Address relocation
- d) Address restructure

**14. What is the ASCII value of NULL or \0.?**

- a) 0
- b) 1
- c) 10
- d) 49

**15. Choose the best answer.**

**Prior to using a pointer variable.**

- a) It should be declared
- b) It should be initialized.
- c) It should be both declared and initialized
- d) None of these.

**16. The statement int \*\*a;**

- a) is illegal
- b) is legal but meaningless
- c) is syntactically and semantically correct
- d) None of these

**17. Which of the following is the correct way of declaring a float pointer:**

- a) float ptr;
- b) float \*ptr;
- c) \*float ptr;
- d) None of the above

**18. Find the output of the following program.**

```
void main()
{
 char *msg = "hi";
 printf(msg);
}
```

- a) hi
- b) h
- c) hi followed by garbage value
- d) Error

**19. What is the base data type of a pointer variable by which the memory would be allocated to it?**

- a) int
- b) float
- c) Depends upon the type of the variable to which it is pointing.
- d) unsigned int

**20. {char \*ptr;**

```
char myString[] = "abcdefg";
ptr = myString;
ptr += 5;}
```

**what string does ptr point to in the sample code above?**

- a) fg
- b) efg
- c) defg

d) cdefg

### **Answers:**

|       |       |       |       |
|-------|-------|-------|-------|
| 1) a  | 2) b  | 3) d  | 4) a  |
| 5) a  | 6) a  | 7) a  | 8) d  |
| 9) c  | 10) a | 11) b | 12) d |
| 13) b | 14) a | 15) c | 16) c |
| 17) c | 18) a | 19) d | 20) a |

### **Short Questions/Answer**

#### **1. What is a Pointer?**

**Ans:** A Pointer in C is used to allocate memory dynamically i.e. at run time. **Pointer is just like another variable, the main difference is that it stores address of another variable rather than a value.**

The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc. An \* (asterisk) symbol followed by the variable name is used for designating variables as pointers.

#### **2. What is a Void pointer?**

**Ans:** Void pointers are used during function declarations. We use a void \* return type permits to return any type. If we assume that our parameters do not change when passing to a function, we declare it as const.

**For example:**

```
void * cube (const void *);
```

#### **3. Write some valid pointer declarations.**

**Ans:** some valid pointer declarations are

```
int *pt1;
char *pt2;
double *pt3;
float *pt4;
```

#### 4. What is a Null Pointer?

**Ans:** We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

#### 5. How to declare a function pointer?

**Ans: Syntax in C**

*function\_return\_type(\*Pointer\_name)(function argument list)*

**For example:**

*double (\*p2f)(double, char)*

#### 6. Write the logic of this statement p = &val[0];

**Ans:** if p = &val[0] which means

\*p == val[0]

(p+1) == &val[1] & \*(p+1) == val[1]

(p+2) == &val[2] & \*(p+2) == val[2]

(p+n) == &val[n] & \*(p+n) == val[n]

#### 7. What is call by reference in pointers?

**Ans:** Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is known as **call by reference**. When a function is called by reference any change made to the reference variable will affect the original variable.

### 8. What are pointer and character strings?

**Ans:** Pointer can also be used to create strings. Pointer variables of **char** type are treated as string.

```
char *str = "Hello";
```

The above statement creates a string and stores its address in the pointer variable **str**. The pointer **str** now points to the first character of the string "Hello". Another important thing to note here is that the string created using **char** pointer can be assigned a value at **runtime**.

```
char *str;
```

```
str = "Hello";
```

The content of the string can be printed as:

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

### 9. What is a double pointer?

**Ans:** Pointers are used to store the address of other variables of similar data type. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

### 10. Write some disadvantages of Pointers?

**Ans:** Some disadvantages of Pointers are:

- a. Pointers are a little complex to understand.
- b. If an incorrect value is provided to a pointer, it may cause memory corruption.
- c. Pointers are also responsible for memory leakage.
- d. Pointers are comparatively slower than that of other variables.

## **Long Questions**

1. How Pointers work in C Programming Language? Explain it with a program.
2. Explain the types of Pointers with examples.
3. Write a program of Swapping two numbers using Pointers.
4. What are Advantages and Disadvantages of Pointers?
5. Write a note on Pointers and Strings.

\*\*\*\*\*

# STRUCTURES AND UNIONS

10

## 10.1 Structures

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful. Structure is a group of variables of different data types represented by a single

name. Let's take an example to understand the need of a structure in C programming.

Let's say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. We will understand this with the help of example.

In structure, data is stored in form of records.

### How to create a structure in C Programming?

#### 10.1.1 Defining a structure

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type**.

##### Syntax in C:

```
struct struct_name {
 DataType member1_name;
 DataType member2_name;
```

```
DataType member3_name;
...
};
```

Here struct\_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

**Note:** The closing curly brace in the structure type declaration must be followed by a semicolon(;) )

### Example of Structure

If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

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

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

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

#### 10.1.2 Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration

is similar to the declaration of any normal variable of any other datatype.  
Structure variables can be declared in following two ways:

### Syntax in C:

```
struct struct_name var_name;
or
struct struct_name {
 DataType member1_name;
 DataType member2_name;
 DataType member3_name;
 ...
} var_name;
```

### 1) Declaring Structure variables separately

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

### 2) Declaring Structure variables with structure definition

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

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

### 10.1.3 Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order

to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot ‘.’ operator also called **period** or **member access** operator.

#### Syntax:

```
var_name.member1_name;
var_name.member2_name;
...
```

#### Example of Accessing Structure Members:

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

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

int main()
{
 struct Student s1;
 /* s1 is a variable of Student type and
 age is a member of Student */
 s1.age = 18;
 /* using string function to add name */
 strcpy(s1.name, "Adnan");
```

```
/*
 displaying the stored values
*/
printf("Name of Student 1: %s\n", s1.name);
printf("Age of Student 1: %d\n", s1.age);

return 0;
}
```

**Output:**

Name of Student 1: Adnan

Age of Student 1: 18

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

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

#### 10.1.4 Structure Initialization

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

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

struct Patient p1 = { 180.75 , 73, 23 }; //initialization
```

**OR**

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

## How to assign values to structure members?

There are three ways to do this.

### 1) Using Dot(.) operator

```
var_name.member_name = value;
```

### 2) All members assigned in one statement

```
struct struct_name var_name =
{value for member1, value for member2 ...so on for all the members}
```

### 3) Designated initializers to set values of Structure members

This is useful when we are doing assignment of only few members of the structure. In the following example the structure variable s2 has only one member assignment. We can explain it with the following example.

```
#include <stdio.h>
struct numbers
{
 int num1, num2;
};
int main()
{
 // Assignment using designated initialization
 struct numbers s1 = { .num2 = 22, .num1 = 11 };
 struct numbers s2 = { .num2 = 30 };

 printf ("num1: %d, num2: %d\n", s1.num1, s1.num2);
 printf ("num1: %d", s2.num2);
 return 0;
}
```

Output:

```
num1: 11, num2: 22
num1: 30
```

Example of Structure in C

```
#include <stdio.h>
/* Created a structure here. The name of the structure is
```

```
* StudentData.
*/
struct StudentData{
 char *stu_name;
 int stu_id;
 int stu_age;
};
int main()
{
 /* student is the variable of structure StudentData*/

 struct StudentData student;

 /*Assigning the values of each struct member here*/
 student.stu_name = "Steve";
 student.stu_id = 1234;
 student.stu_age = 30;

 /* Displaying the values of struct members */
 printf("Student Name is: %s", student.stu_name);
 printf("\nStudent Id is: %d", student.stu_id);
 printf("\nStudent Age is: %d", student.stu_age);
 return 0;
}
```

Output:

Student Name is: Steve  
Student Id is: 1234  
Student Age is: 30

### 10.1.5 Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable.

Example : struct employee emp[5];

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

**Program:1**

```
#include<stdio.h>
struct Employee
{
 char ename[10];
 int sal;
};
struct Employee emp[5];
int i, j;
void ask()
{
 for(i = 0; i < 3; i++)
 {
 printf("\nEnter %dst Employee record:\n", i+1);
 printf("\nEnter Employee name:\t");
 scanf("%s", emp[i].ename);
 printf("\nEnter Salary:\t");
 scanf("%d", &emp[i].sal);
 }
 printf("\nDisplaying Employee record:\n");
 for(i = 0; i < 3; i++)
 {
 printf("\nEmployee name is %s", emp[i].ename);
 printf("\nSalary is %d", emp[i].sal);
 }
}
void main()
{
 ask();
}
```

**Program:2**

Here we want to store data of 5 persons for this purpose, we would be required to use 5 different structure variables, from sample1 to sample 5. To have 5

The structure type person is having 2 elements: Name an array of 25 characters

and integer type variable age.

Using the statement struct person sample[5]; we are declaring a 5 element array of structures. Here, each element of sample is a separate structure of type person. We, then defined 2 variables into index and an array of 8 characters, info. Here, the first loop executes 5 times, with the value of index varying from 0 to 4. The first printf() function displays. For the first time this name you enter will go to sample[0] . name. The second for loop is responsible

for printing the information stored in the array of structures.

### **10.1.6 Nested Structures: (Struct inside another struct)**

You can use a structure inside another structure, which is fairly possible. As explained above that once you declared a structure, the **struct struct\_name** acts as a new data type so you can include it in another struct just like the data type of other data members.

### **Example of Nested Structure**

Let's say we have two structure like this:

#### **Structure 1: stu\_address**

```
struct stu_address
{
 int street;
 char *state;
 char *city;
 char *country;
```

```
}
```

### **Structure 2: stu\_data**

```
struct stu_data
{
 int stu_id;
 int stu_age;
 char *stu_name;
 struct stu_address stuAddress;
}
```

As you can see here that we have nested a structure inside another structure.

#### **Assignment for struct inside struct (Nested struct)**

Let's take the example of the two structure that we see above to understand the logic.

```
struct stu_data mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..
```

How to access nested structure members?

Using chain of “.” operator.

Suppose you want to display the city alone from nested struct .

```
printf("%s", mydata.stuAddress.city);
```

Example:2

```
struct Student
{
 char[30] name;
 int age;
 /* here Address is a structure */
 struct Address
 {
 char[50] locality;
 char[50] city;
```

```
 int pincode;
}addr;
};
```

### 10.1.7 Structure as Function Arguments

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

Example:

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

```
struct Student
```

```
{
```

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

```
 int roll;
```

```
};
```

```
void show(struct Student st);
```

```
void main()
```

```
{
```

```
 struct Student std;
```

```
 printf("\nEnter Student record:\n");
```

```
 printf("\nStudent name:\t");
```

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

```
 printf("\nEnter Student rollno.:\t");
```

```
 scanf("%d", &std.roll);
```

```
 show(std);
```

```
}
```

```
void show(struct Student st)
```

```
{
```

```
 printf("\nstudent name is %s", st.name);
```

```
 printf("\nroll is %d", st.roll);
```

```
}
```

### 10.1.8 Use of `typedef` in Structure

`typedef` makes the code short and improves readability. In the above discussion

we have seen that while using structs every time we have to use the lengthy

syntax, which makes the code confusing, lengthy, complex and less readable.

The simple solution to this issue is use of `typedef`. It is like an alias of struct.

#### Code without `typedef`

```
struct home_address {
 int local_street;
 char *town;
 char *my_city;
 char *my_country;
};
...
struct home_address var;
var.town = "Agra";
```

#### Code using `typedef`

```
typedef struct home_address{
 int local_street;
 char *town;
 char *my_city;
 char *my_country;
}addr;
..
..
addr var1;
var.town = "Agra";
```

Instead of using the `struct home_address` every time you need to declare struct

variable, you can simply use `addr`, the `typedef` that we have defined.

## Application of **typedef**

**typedef** can be used to give a name to user defined data type as well. Let's see its use with structures.

```
typedef struct
{
 type member1;
 type member2;
 type member3;
} type_name;
```

Here **type\_name** represents the structure definition associated with it. Now this **type\_name** can be used to declare a variable of this structure type.

```
type_name t1, t2;
```

## Advantages of structure

Following are the benefits for using structure:

- Structures gather more than one piece of data about the same subject together in the same place.
- It is helpful when you want to gather the data of similar data types and parameters like first name, last name, etc.
- It is very easy to maintain as we can represent the whole record by using a single name.
- In structure, we can pass complete set of records to any function using a single parameter.
- You can use an array of structure to store more records with similar types.

### Disadvantages of structure

Here are some drawbacks for using structure:

- If the complexity of IT project goes beyond the limit, it becomes hard to manage.
- Change of one data structure in a code necessitates changes at many other places. Therefore, the changes become hard to track.
- Structure is slower because it requires storage space for all the data.
- You can retrieve any member at a time in structure whereas you can access one member at a time in the union.
- Structure occupies space for each and every member written in inner parameters while union occupies space for a member having the highest size written in inner parameters.
- Structure supports flexible array. Union does not support a flexible array.

## 10.2 Unions

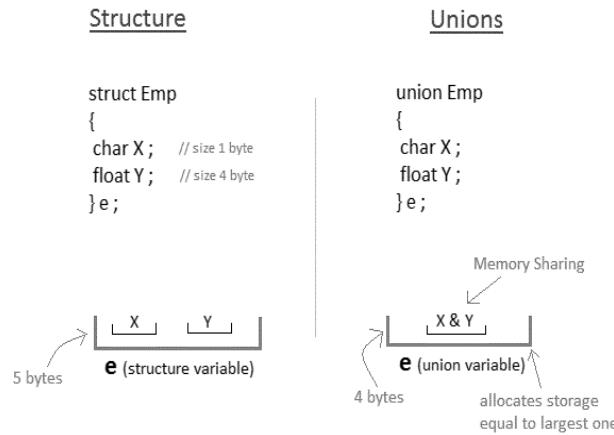
**Unions** are conceptually similar to **structures**. The syntax to declare/define a

union is also similar to that of a structure. The only differences is in terms

of storage. In **structure** each member has its own storage location, whereas

all members of **union** uses a single shared memory location which is equal to

the size of its largest data member.



This implies that although a **union** may contain many members of different

types, **it cannot handle all the members at the same time.**

We can use the unions in the following locations.

1. Share a single memory location for a variable and use the same location for another variable of different data type.
2. Use it if you want to use, for example, a long variable as two short type variables.
3. We don't know what type of data is to be passed to a function, and you pass union which contains all the possible data types.

### 10.2.1 Defining a Union in C?

Union can be defined by the keyword `union` followed by list of member variables contained in curly braces.

**Syntax:**

```
union tagname
{
```

```
 data_type member_1;
 data_type member_2;
 data_type member_3;
 ...
 data_type member_N;
};
```

**Example:**

```
union Employee{
 int age;
 long salary;
};
```

Here we have defined a union with the name union\_name and it has two members i.e. age of type int and salary of type long.

**10.2.2 Declaring a Union in C?**

Just like structure you can declare union variable with union definition or separately.

```
union tagname
{
 data_type member_1;
 data_type member_2;
 data_type member_3;
 ...
 data_type member_N;
} var_union;
```

```
union tagname var_union_2;
```

We can declare the union in various ways. By taking the above example we

can declare the above defined union as.

```
union Employee{
 int age;
 long salary;
} employee;
```

So employee will be the variable of type Employee . We can also declare the above union as:

```
Employee employee;
```

### **Example-2**

```
union item
{
 int m;
 float x;
 char c;
}It1;
```

This declares a variable **It1** of type **union item**. This **union** contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the **union** variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union. In the union declared above the member **x** requires **4 bytes** which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

#### **10.2.3 Initializing a Union in C?**

We can initialize the union in various ways. For example

```
union Employee{
```

```
int age;
long salary;
} employee={20000};
```

or we can initialize it as

```
employee.age= 30;
employee.salary=20000;
```

Normally when we declare the union it is allocated the memory that its biggest member can occupy. So here in our example employee will occupy the memory which a long type variable can occupy.

#### **10.2.4 How is the size of union decided by compiler?**

Size of a union is taken according the size of largest member in union.

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

```
union test1 {
 int x;
 int y;
} Test1;
```

```
union test2 {
 int x;
 char y;
} Test2;
```

```
union test3 {
 int arr[10];
 char y;
} Test3;
```

```
int main()
{
```

```
 printf("sizeof(test1) = %lu, sizeof(test2) =
%lu,
 "sizeof(test3) = %lu",
 sizeof(Test1),
 sizeof(Test2), sizeof(Test3));
 return 0;
}
```

**Output:**

sizeof(test1) = 4, sizeof(test2) = 4, sizeof(test3) = 40

### 10.2.5 Accessing a Union Member in C

Syntax for accessing any **union** member is similar to accessing structure members,

union test

```
{
 int a;
 float b;
 char c;
}t;
```

t.a; //to access members of union t

t.b;

t.c;

#### Example: Accessing Union Members

```
#include <stdio.h>
union Job {
 float salary;
 int workerNo;
} j;

int main() {
 j.salary = 12.3;
```

```
// when j.workerNo is assigned a value,
// j.salary will no longer hold 12.3
j.workerNo = 100;

printf("Salary = %.1f\n", j.salary);
printf("Number of workers = %d", j.workerNo);
return 0;
}
```

**Output:**

Salary = 0.0

Number of workers = 100

**Example-1**

```
#include <stdio.h>
union item
{
 int a;
 float b;
 char ch;
};

int main()
{
 union item it;
 it.a = 12;
 it.b = 20.2;
 it.ch = 'z';

 printf("%d\n", it.a);
 printf("%f\n", it.b);
 printf("%c\n", it.ch);
 return 0;
}
```

**Output:**

-26426

20.1999

z

As you can see here, the values of a and b get corrupted and only variable c prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

In the above example, value of the variable c was stored at last, hence the value of other variables is lost.

### **Similarly union values can be accessed via pointer variables.**

```
union Employee{
 int age;
 long salary;
} *employee;

(*employee).age;
or;
employee->age;
```

If we have a union variable then we can access members of union using dot operator (.) , similarly if we have pointer to union then we can access members of union using arrow operator (->) .

### **Program-1:**

#### **Pointers to unions?**

Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

```
#include <stdio.h>

union test {
 int x;
 char y;
```

```
};

int main()
{
 union test p1;
 p1.x = 65;

 // p2 is a pointer to union p1
 union test* p2 = &p1;

 // Accessing union members using pointer
 printf("%d %c", p2->x, p2->y);
 return 0;
}
```

**Output:**

65 A

**Example of Union****Program-2**

```
1 #include<stdio.h>
2
3 /*
4 union is defined above all functions so it is
5 global.
6 */
7 union data
8 {
9 int var1;
10 double var2;
11 char var3;
12 };
13
14 int main()
15 {
16 union data t;
17
18 t.var1 = 10;
```

```

19 printf("t.var1 = %d\n", t.var1);
20
21 t.var2 = 20.34;
22 printf("t.var2 = %f\n", t.var2);
23
24 t.var3 = 'a';
25 printf("t.var3 = %c\n", t.var3);
26
27 printf("\nSize of structure: %d",
28 sizeof(t));
28
29 return 0;
30 }
```

***Output:***

*t.var1 = 10*  
*t.var2 = 20.340000*  
*t.var3 = a*

*Size of structure: 8*

**Explanation of Program:**

In lines 7-12, a union **data** is declared with three members namely **var1** of type

**int**, **var2** of type **double** and **var3** of type **char**. When the compiler sees the definition of union it will allocate sufficient memory to hold the largest member

of the union. In this case, the largest member is **double**, so it will allocate **8** bytes

of memory. If the above definition would have been declared as a structure, the compiler would have allocated **13** bytes (**8+4+2**) of memory (here we are ignoring holes, click [here](#) to learn more about it).

In line 16, a union variable **t** of type **union data** is declared.

In line 18, the first member of **t** i.e **var1** is initialized with a value of **10**. The

important thing to note is that at this point the other two members contain garbage values.

In line 19, the value of **t.var1** is printed using the **printf()** statement.

In line 21, the second member of **t** i.e **var2** is assigned a value of **20.34**. At this point, the other two members contain garbage values.

In line 22, the value of **t.var2** is printed using **printf()** statement.

In line 24, the third member of **t** i.e **var3** is assigned a value of '**a**' . At this point, the other two members contain garbage values.

In line 25, the value of **t.var3** is printed using **printf()** statement.

In line 27, the **sizeof()** operator is used to print the size of the union. Since we know that, in the case of a union, the compiler allocates sufficient memory to hold

the largest member. The largest member of union **data** is **var2** so the **sizeof()**

operator returns **8** bytes which is then printed using the **printf()** statement.

## Example-2

For example in the following C program, both **x** and **y** share the same location.

If we change **x**, we can see the changes being reflected in **y**.

```
#include <stdio.h>

// Declaration of union is same as structures
union test {
 int x, y;
};

int main()
{
 // A union variable t
 union test t;

 t.x = 2; // t.y also gets value 2
 printf("After making x = 2:\n x = %d,
y = %d\n\n", t.x, t.y);

 t.y = 10; // t.x is also updated to 10
 printf("After making y = 10:\n x = %d,
y = %d\n\n", t.x, t.y);
 return 0;
}
```

### Output:

After making x = 2:

x = 2, y = 2

After making y = 10:

x = 10, y = 10

### 10.2.6 Initializing Union Variable

In the above program, we have seen how we can initialize individual members of a union variable. We can also initialize the union variable at the time of declaration, but there is a limitation. Since union share same memory all the members can't hold the values simultaneously. So we can only initialize one of the

members of the union at the time of declaration and this privilege goes to the first member. **For example:**

***union data***

```
{
 int var1;
 double var2;
 char var3;
};
```

***union data j = {10};***

This statement initializes the union variable **j** or in other words, it initializes only the first member of the union variable **j**.

### **Designated initializer**

Designated initializer allows us to set the value of a member other than the first member of the union. Let's say we want to initialize the **var2** member of union data at the time of declaration. Here is how we can do it.

**union data k = { .var2 = 9.14 };**

This will set the value of **var2** to **9.14**. Similarly, we can initialize the value of the third member at the time of declaration.

**union data k = { .var3 = 'a' };**

### **10.2.7 Difference Between Structure and Pointer**

**The following program demonstrates the difference between a structure and a pointer.**

```
1 #include<stdio.h>
2 /*
3 union is defined above all functions so it is
global.
4 */
```

```
5
6 struct s
7 {
8 int var1;
9 double var2;
10 char var3;
11 };
12
13 union u
14 {
15 int var1;
16 double var2;
17 char var3;
18 };
19
20 int main()
21 {
22 struct s a;
23 union u b;
24
25 printf("Information about structure
variable \n\n");
26
27 printf("Address variable of a = %u\n",
&a);
28 printf("Size of variable of a = %d\n",
sizeof(a));
29
30 printf("Address of 1st member i.e var1 =
%u\n", &a.var1);
31 printf("Address of 2nd member i.e var2 =
%u\n", &a.var2);
32 printf("Address of 3rd member i.e var3 =
%u\n", &a.var3);
33
34 printf("\n");
```

```
35
36 printf("Information about union variable
37 \n\n");
38 printf("Address of variable of b = %u\n",
39 &b);
40 printf("Size of variable of b = %d\n",
41 sizeof(b));
42 printf("Address of 1st member i.e var1 =
43 %u\n", &b.var1);
44 printf("Address of 2nd member i.e var2 =
45 %u\n", &b.var2);
46 printf("Address of 3rd member i.e var3 =
47 %u\n", &b.var3);
48 printf("\n\n");
49
50 return 0;
51 }
```

**Expected Output:**

Address variable of a = 2686728

Size of variable of a = 24

Address of 1st member i.e var1 = 2686728

Address of 2nd member i.e var2 = 2686736

Address of 3rd member i.e var3 = 2686744

Information about union variable

Address of variable of b = 2686720

Size of variable of b = 8

Address of 1st member i.e var1 = 2686720

Address of 2nd member i.e var2 = 2686720

Address of 3rd member i.e var3 = 2686720

### How it works:

In lines 6-11, a structure of type **s** is declared with three members namely **var1** of type **int**, **var2** of type **float** and **var3** of type **char**.

In line 13-18, a union of type **u** is declared with three members namely **var1** of type **int**, **var2** of type **float** and **var3** of type **char**.

In line 22 and 23 declares a structure variable **a** of type **struct s** and union variable **b** of type **union u** respectively.

In line 27, the address of structure variable **a** is printed using **&** operator.

In line 28, the size of structure variable is printed using **sizeof()** operator.

Similarly the **printf()** statements in line 38 and 39 prints address and size of union variable **b** respectively.

All the member of a union shares the same memory that's why the next three **printf()** statements prints the same address.

### 10.2.8 Advantages and Disadvantages of Union

#### Advantages of Union

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

#### Disadvantages of Union

- You can use only one union member at a time.
- All the union variables cannot be initialized or used with varying values at a time.

- Union assigns one common storage space for all its members.

### **10.2.9 Similarities and Differences between Structure and Union**

#### **Similarities between Structure and Union**

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators

The two structures or unions in the assignment must have the same members and member types.

4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. ‘.’ operator is used for accessing members.

#### **Differences between Structure and Union**

| Structure                                                                                 | Union                                                                                          |
|-------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| 1.You can use a struct keyword to define a structure.                                     | 1.You can use a union keyword to define a union.                                               |
| 2.Every member within structure is assigned a unique memory location.                     | 2. In union, a memory location is shared by all the data members.                              |
| 3. Changing the value of one data member will not affect other data members in structure. | 3. Changing the value of one data member will change the value of other data members in union. |
| 4. It enables you to initialize several members at once.                                  | 4. It enables you to initialize only the first member of union.                                |

|                                                                                 |                                                                                        |
|---------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| 5. The total size of the structure is the sum of the size of every data member. | 5. The total size of the union is the size of the largest data member.                 |
| 6. It is mainly used for storing various data types.                            | 6. It is mainly used for storing one of the many data types that are available.        |
| 7. It occupies space for each and every member written in inner parameters.     | 7. It occupies space for a member having the highest size written in inner parameters. |
| 8. You can retrieve any member at a time.                                       | 8. You can access one member at a time in the union.                                   |
| 9. It supports flexible array.                                                  | 9. It does not support a flexible array.                                               |

Let's take an example to demonstrate the difference between unions and Structures:

### Program:

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

```
union unionJob
```

```
{
```

```
//defining a union
```

```
char name[32];
```

```
float salary;
```

```
int workerNo;
```

```
} uJob;
```

```
struct structJob
```

```
{
```

```
char name[32];
```

```
float salary;

int workerNo;

} sJob;

int main(){

printf("size of union = %d bytes", sizeof(uJob));

printf("\nsize of structure = %d bytes", sizeof(sJob));

return 0;
}
```

Output:

size of union = 32

size of structure = 40

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

Here, the size of sJob is 40 bytes because

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

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

With a union, all members share the same memory.

### 10.3 Unions of Structures

Just as one structure can be nested within another, a Union too can be

Nested in another union. Not only that, there can be a union in a structure, or a structure in a union.

The members of the union share the same address while the members of structure

don't. The difference in size of structure and union variable also suggests that in

some cases union may provide a more economical use of memory. Another important point I want to emphasise is that size of the structure may be greater than the sum of members due to the boundary alignment discussed earlier, the same thing is true for unions.

A structure can be a member one of the union. Similarly, a union can be a member of the structure.

After learning about unions we know that at a time only one member of union variable will be usable, that means the union is perfect for defining quantity. So instead if storing different quantity as members of structure why not create a union of a quantity that way for any goods only one member of the union will be usable.

**Example:**

```
struct goods
{
 char name[20];

 union quantity
 {
 int count;
 float weight;
 float volume;
 } quant;
} g;
```

**Instead of nesting union quantity we can define it outside the goods structure.**

**Example:**

```
union quantity
{
 int count;
 float weight;
 float volume;
```

```
};

struct goods
{
 char name[20];
 union quantity quant;
} g;
```

If we want to access the value of count we can write:

**g.quant.count**

Similarly to access the value of weight we can write:

**g.quant.weight**

**The following program demonstrates how we can use a union as  
a  
member of the structure.**

```
1 #include<stdio.h>
2
3 /*
4 union is defined above all functions so it is
5 global.
6 */
7 union quantity
8 {
9 int count;
10 float weight;
11 float volume;
12 };
13
14 struct goods
15 {
16 char name[20];
```

```

17 union quantity q;
18 };
19
20 int main()
21 {
22 struct goods g1 = { "apple",
23 {.weight=2.5} };
24 struct goods g2 = { "balls", {.count=100}
25 };
26
27 printf("Goods name: %s\n", g1.name);
28 printf("Goods quantity: %.2f\n\n",
29 g1.q.weight);
30
31 printf("Goods name: %s\n", g2.name);
32 printf("Goods quantity: %d\n\n",
33 g2.q.count);
34
35 return 0;
36 }
```

**Output:**

Goods name: apple  
 Goods quantity: 2.50

Goods name: balls  
 Goods quantity: 100

**How it works:**

In lines 7-12, a union **quantity** is declared with three members namely **count** of type **int**, **weight** of type **float** and **volume** of type **float**.  
 In lines 14-18, structure **goods** is declared with 2 members namely **name** which is an array of characters and **w** of type **union** **quantity**.  
 In line 22, structure variable **g1** is declared and initialized. The important thing to note how designated initializer is used to initialize the **weight** member

of the union. If we would have wanted to initialize the first element, we would

have done it like this:

```
struct goods g1 = { "apple", {112} };
```

or

```
struct goods g1 = { "apple", 112 };
```

In line 23, structure variable **g2** is declared and initialized.

In line 25 and 26, **name** and **weight** of the first goods is printed using **printf()**

statement. Similarly in line 28 and 29, **name** and **weight**

of the second goods is printed using **printf()** statement.

### What are applications of union?

Unions can be useful in many situations where we want to use the same memory for two or more members. For example, suppose we want to implement

a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {
 struct NODE* left;
 struct NODE* right;
 double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

```
struct NODE {
 bool is_leaf;
 union {
 struct
 };
```

```
struct NODE* left;
struct NODE* right;
} internal;
double data;
} info;
};
```

END OF CHAPTER 10

---

## EXERCISE

### **Multiple Choice Questions:**

- 1. Which of the following are themselves a collection of different data types?**
  - a) String
  - b) Structure
  - c) Char
  - d) All of the mentioned
  
- 2. User-defined data type can be derived by\_\_\_\_\_.**
  - a) struct
  - b) enum
  - c) typedef
  - d) All of the mentioned
  
- 3. Which of the following cannot be a structure member?**
  - a) Another structure
  - b) Function
  - c) Array
  - d) None of the mentioned

**4. Number of bytes in memory taken by the below structure is?**

```
struct test
```

```
{
 int k;
 char c;
};
```

- a) Multiple of integer size
- b) integer size+character size
- c) Depends on the platform
- d) Multiple of word size

**5. What would be the size of the following union declaration?**

```
union uTemp
```

```
{
 double a;
 int b[10];
 char c;
}u;
```

(Assuming size of double = 8, size of int = 4, size of char = 1)

- a) 4
- b) 8
- c) 40
- d) 80

**6. Members of a union are accessed as\_\_\_\_\_.**

- a) union-name.member
- b) union-pointer->member
- c) Both a & b
- d) None of the mentioned

**7. Which of the following share a similarity in syntax?**

**1. Union, 2. Structure, 3. Arrays and 4. Pointers.**

- a) 3 and 4
- b) 1 and 2
- c) 1 and 3

d) 1, 3 and 4

**1. Which of the following data types are accepted while declaring bit-fields?**

- a) char
- b) float
- c) double
- d) None of the mentioned

**9. Which of the following is not allowed?**

- a) Arrays of bit fields
- b) Pointers to bit fields
- c) Functions returning bit fields
- d) None of the mentioned

**10. Bit fields can only be declared as part of a structure.**

- a) false
- b) true
- c) Nothing
- d) Varies

**11. What will be the size of the following structure?**

```
#include <stdio.h>
struct temp
{
 int a[10];
 char p;
};
```

- a) 5
- b) 11
- c) 41
- d) 44

**1. Which among the following is never possible in C**

**when members in a structure are same as that in a union?**

//Let P be a structure

//Let Q be a union

- a) sizeof(P) is greater than sizeof(Q)
- b) sizeof(P) is equal to sizeof(Q)
- c) sizeof(P) is less than to sizeof(Q)
- d) None of the above

**13. What is the similarity between a structure and union?**

- a) All of them let you define new values
- b) All of them let you define new data types
- c) All of them let you define new pointers
- d) All of them let you define new structures

**14. How will you free the allocated memory ?**

- a) remove(var-name);
- b) free(var-name);
- c) delete(var-name);
- d) dalloc(var-name);

**2. The \_\_\_\_\_ allocates enough storage in a number to accommodate  
the largest element in the union.**

- a) Interpreter
- b) Compiler
- c) Assembler
- d) Both b and c

**3.Just as one structure can be nested within another, a \_\_\_\_\_  
too can be nested in another union.**

- a) Union
- b) Array
- c) String
- d) None of the above

**17. To declare a structure you must start with the keyword \_\_\_\_\_.**

- a) stct
- b) struct
- c) structure();
- d) both a and b

**18. A union consist of number of elements that**

- a) All have the same type
- b) Must be structures
- c) Are grouped next to each other in memory
- d) All occupy the same space in memory

**19. The structure name is often referred to as its \_\_\_\_\_.**

- a) Label
- b) Title
- c) Tag
- d) Name

**20. The fields of structures can be accessed with the operator called the \_\_\_\_\_ operator.**

- a) Dot
- b) Colon
- c) Semicolon
- d) Comma

### **Answers:**

|       |       |       |       |
|-------|-------|-------|-------|
| 1) b  | 2) d  | 3) d  | 4) b  |
| 5) c  | 6) c  | 7) b  | 8) a  |
| 9) d  | 10) b | 11) d | 12) c |
| 13) b | 14) c | 15) b | 16) a |
| 17) b | 18) d | 19) c | 20) a |

## Short Questions/Answer

### **3. Define structure?**

**Ans:** Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

### **4. Define Unions?**

**Ans:** Unions are conceptually similar to structures. The syntax to declare/define

a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.

### **3. How many data types are used in structure elements or fields?**

**Ans:** The elements or fields, which make up the structure, use the four basic data types. i.e. int, float, char and double.

### **4. What are the benefits of using Structures?**

**Ans:** Following are the benefits for using structure:

- Structures gather more than one piece of data about the same subject together in the same place.
- It is helpful when you want to gather the data of similar data types and parameters like first name, last name, etc.
- It is very easy to maintain as we can represent the whole record by using a single name.

**5. How to declare a structure variable?**

**Ans:** We can declare a structure variable as:

**Syntax in C:**

```
struct struct_name var_name;
```

or

```
struct struct_name {
```

```
 DataType member1_name;
```

```
 DataType member2_name;
```

```
 DataType member3_name;
```

...

```
} var_name;
```

**6. How can we access members of Union?**

**Ans:** If we have a union variable then we can access members of union using dot operator (.) , similarly if we have pointer to union then we can access members of union using arrow operator (->) .

Syntax for accessing any **union** member is similar to accessing structure members,  
union test

```
{
 int a;
 float b;
 char c;
}t;
t.a; //to access members of union t
t.b;
t.c;
```

**7. Define array of structures?**

**Ans:** We can also declare an array of **structure** variables, in which each element

of the array will represent a **structure** variable.

**example :** *struct employee emp[5];*

The above line defines an array *emp* of size 5. Each element of the array *emp* is

of type *employee*.

### 8. What is the syntax of a union ?

**Ans:** Union can be defined by the keyword `union` followed by list of member variables contained in curly braces.

#### Syntax:

```
union tagname
{
 data_type member_1;
 data_type member_2;
 data_type member_3;
 ...
 data_type member_N;
};
```

#### Example:

```
union Employee{
 int age;
 long salary;
};
```

### 9. Define union of structures?

**Ans:** Just as one structure can be nested within another, a Union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union.

**10. What is the use of `typedef` in structure?**

**Ans:** `typedef` makes the code short and improves readability. We have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of `typedef`. It is like an alias of struct.

**Long Questions**

1. How to create a Structure in C programming. Explain it with examples.
2. What are Unions? How can we define , declare and initialize the Unions?
3. Differentiate between Structure and Pointer. Explain it with a program.
4. What are the differences between Structure and Union?
5. Write a note on Union of Structures.

\*\*\*\*\*

## INTRODUCTION TO FILE HANDLING IN C

11

New way of dealing with data is file handling.

Data is stored onto the disk and can be retrieved whenever required.

Output of the program may be stored onto the disk

In C we have many functions that deals with file handling

A file is a collection of bytes stored on a secondary storage device(generally a disk)

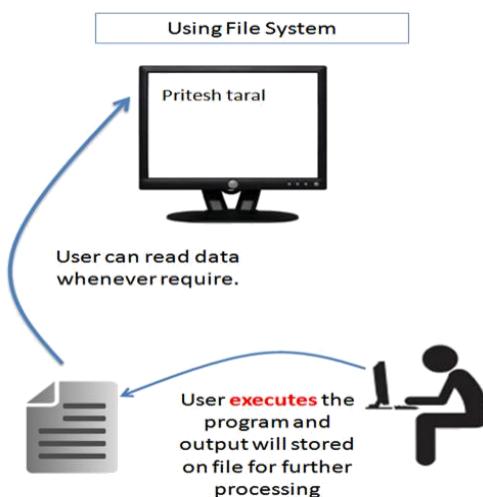
Collection of byte may be interpreted as –

Single character

Single Word

Single Line

Complete Structure.



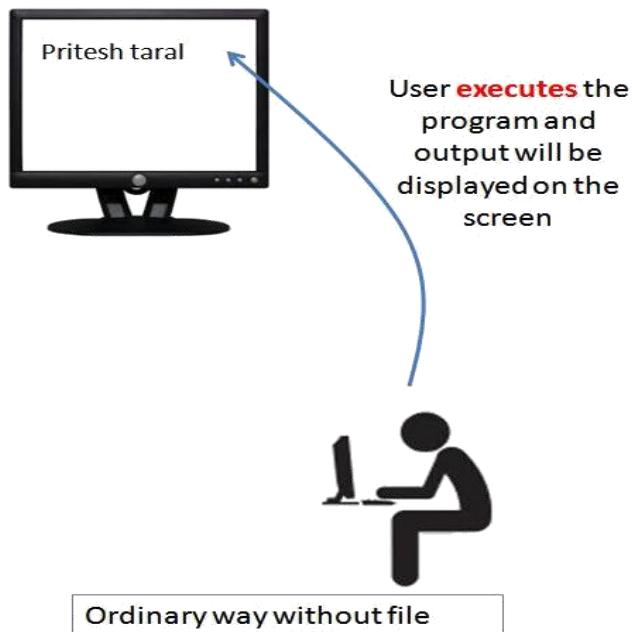
## DRAWBACKS OF TRADITIONAL I/O SYSTEM

Until now we are using Console Oriented I/O functions.

—Console Application means an application that has a text-based interface.  
(black screen window))

Most applications require a large amount of data , if this data is entered through console then it will be quite time consuming task

Main drawback of using Traditional I/O :- data is temporary (and will not be available during re-execution )



Consider example –

We have written C Program to accept person detail from user and we are going

to print these details back to the screen.

Now consider another scenario, suppose we want to print same data that we have entered previously.

We cannot save data which was entered on the console before.

Now we are storing data entered (during first run) into text file and when we need this data back (during 2nd run), we are going to read file.

### **11.1 Explain Types of Disk I/O**

#### **File I/O Streams in C Programming Language**

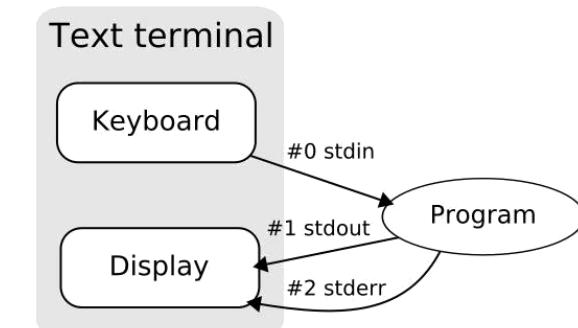
In C all input and output is done with streams Stream is nothing but the sequence of bytes of data

A sequence of bytes flowing into program is called input stream

A sequence of bytes flowing out of the program is called output stream Use of Stream make I/O machine independent. Predefined Streams:

### **11.2. Explain Standard Input/Output**

|        |                 |
|--------|-----------------|
| stdin  | Standard Input  |
| stdout | Standard Output |



**Standard Input Stream Device**

stdin stands for (Standard Input)

Keyboard is standard input device .

Standard input is data (Often Text) going into a program.

The program requests data transfers by use of the read operation.

Not all programs require input.

### **Standard Output Stream Device**

stdout stands for (Standard Output)

Screen(Monitor) is standard output device .

Standard output is data (Often Text) going out from a program. The program sends data to output device by using write operation. Difference Between Std.

Input and Output Stream Devices :

Some Important Summary :

| <b>Point</b>      | <b>Input Stream</b> | <b>Output Stream</b> |
|-------------------|---------------------|----------------------|
| Standard Device 1 | Keyboard            | Screen               |
| Standard Device 2 | Scanner             | Printer              |
| IO Function       | scanf and gets      | printf and puts      |
| IO Operation      | Read                | Write                |

|      |                       |                        |
|------|-----------------------|------------------------|
| Data | Data goes from stream | data comes into stream |
|------|-----------------------|------------------------|

### 11.3 Explain Binary and Text Mode Files

#### Text files Format in C Programming

Text File is Also Called as —Flat File—.

Text File Format is Basic File Format in C Programming.

Text File is simple Sequence of ASCII Characters.

Each Line is Characterized by EOL Character (End of Line).

```

1000233 Miralda John
1000234 Faley Nick
1000235 Baylog Cathy
1000236 Gallardo Mike
1000237 Christian Daniel
1000238 Baufield Daniel
1000239 Frazier Robert
1000240 Garrido Edward
1000241 Williams Zachary
1000242 Morel David
1000243 Padilla Damian
1000244 Rosenberg Wayne
1000245 Blanchard Phong s
1000246 Wiggins David
1000247 Miller Jeffrey
1000248 Coon Terry
1000249 Chretien Walter
1000250 Myers Timothy

1000233 Miralda John
1000234 Faley Nick
1000235 Baylog Cathy

```

#### Text File Formats

Text File have .txt Extension.

Text File Format have Little contains very little formatting .

The precise definition of the .txt format is not specified, but typicallymatches the format accepted by the system terminal or simple text editor.

Files with the .txt extension can easily be read or opened by any program that reads text and, for that reason, are considered universal (or platform independent). Text Format Contain Mostly English Characters

#### What are Binary Files

Binary Files Contain Information Coded Mostly in Binary Format.

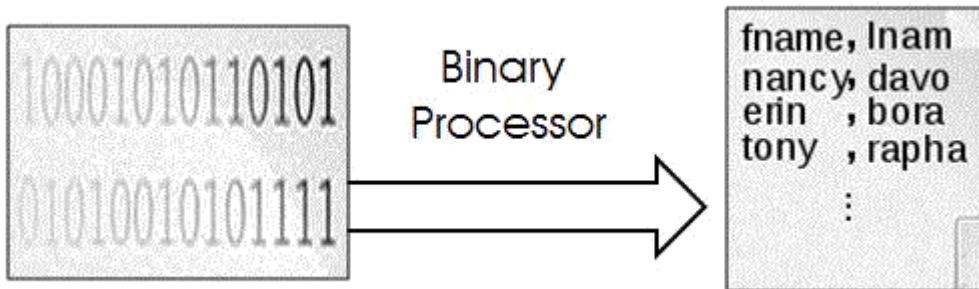
Binary Files are **difficult to read for human.**

Binary Files can be processed by **certain applications or processors.**

Only **Binary File Processors can understand Complex Formatting** Information Stored in Binary Format.

Humans can read binary **files only after processing.**

All Executable Files are **Binary Files.**



Explanation :

As shown in fig. Binary file is stored in Binary Format (in 0/1). This Binary file is difficult to read for humans. So generally Binary file is given as input to the Binary file Processor. Processor will convert binary file into equivalent readable file.

#### **Some Examples of the Binary files :**

Executable Files

Database files

Before opening the file we must understand the **basic concept of file in C Programming**, **Types of File**. If we want to display some message on the console from the file then we must open it in read mode.

#### **Opening and Defining FILE in C Programming**

Before storing data onto the secondary storage , firstly we must specify following things –

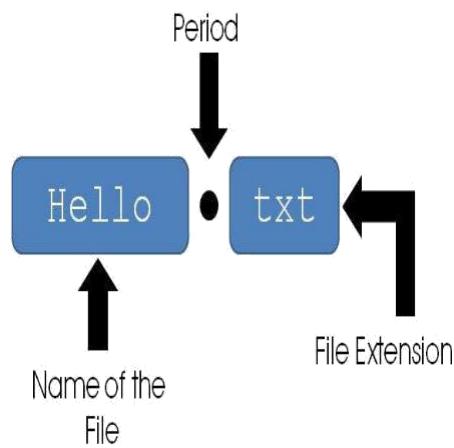
File name

Data Structure

Purpose / Mode

Very first task in File handling is to open file

File name : Specifies Name of the File



File name consists of **two fields**

First field is **name field** and second field is of **extension field**

**Extension field is optional**

Both File name and extension are separated by period or dot.

### **Data Structure**

Data structure of file is defined as FILE in the library of standard I/O functions In short we have to declare the pointer variable of type FILE

## **11.4 Program Record Input/Output**

### **Mode of FILE opening**

In C Programming we can open file in different modes such as reading mode,writing mode and appending mode depending on purpose of handling file. Following are the different Opening modes of File :

| Opening Mode     | Purpose                                             | Previous Data |
|------------------|-----------------------------------------------------|---------------|
| <b>Reading</b>   | File will be opened just for reading purpose        | Retained      |
| <b>Writing</b>   | File will be opened just for writing purpose        | Flushed       |
| <b>Appending</b> | File will be opened for appending something in file | Retained      |

### Different Steps to Open File

Step 1 : Declaring FILE pointer

Firstly we need pointer variable which can point to file. below is the syntax for declaring the file pointer.

```
FILE *fp;
```

Step 2 : Opening file hello.txt

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

Example : Opening the File and Defining the File

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

```
int main()
{
FILE *fp;
char ch;

fp = fopen("INPUT.txt","r") // Open file in Read mode

fclose(fp); // Close File after Reading
```

```

return(0);
}

```

If we want to open file in different mode then following syntax will be used –

Reading Mode fp = fopen("hello.txt","r");

Writing Mode fp = fopen("hello.txt","w");

Append Mode fp = fopen("hello.txt","a");

## Opening the File

```

#include<stdio.h>

void main()
{
FILE *fp;
char ch;
fp = fopen("INPUT.txt","r"); // Open file in Read mode

while(1)
{
ch = fgetc(fp); // Read a Character
if(ch == EOF) // Check for End of File
break ;

printf("%c",ch);
}
fclose(fp); // Close File after Reading
}

```

### File Opening Mode Chart

| <b>Mode</b> | <b>Meaning</b> | <b>fopen Returns if FILE-</b> |                   |
|-------------|----------------|-------------------------------|-------------------|
|             |                | <b>Exists</b>                 | <b>Not Exists</b> |
| r           | Reading        | –                             | NULL              |
| w           | Writing        | Over write on Existing        | Create            |

|    |                        |                                                                   |                       |
|----|------------------------|-------------------------------------------------------------------|-----------------------|
|    |                        |                                                                   | New<br>File           |
| a  | Append                 | -                                                                 | Create<br>New<br>File |
| r+ | Reading +<br>Writing   | New data is written at the beginning<br>overwriting existing data | Create<br>New<br>File |
| w+ | Reading +<br>Writing   | Over write on Existing                                            | Create<br>New<br>File |
| a+ | Reading +<br>Appending | New data is appended at the end of<br>file                        | Create<br>New<br>File |

Explanation :

File can be opened in **basic 3 modes** : Reading Mode, Writing Mode, Appending Mode If File is not present on the path specified then **New File can be created using Write and Append Mode.**

Generally we used to open **following types of file in C –**

| File Type     | Extension |
|---------------|-----------|
| C Source File | .c        |
| Text File     | .txt      |
| Data File     | .dat      |

Writing on the file will **overwrite previous content**

EOF and feof function >> stdio.h >> File Handling in C

Syntax :

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

What it does?

Macro tests if end-of-file has been reached on a stream.

feof is a macro that tests the given stream for an end-of-file indicator.

Once the indicator is set, read operations on the file return the indicator until rewind is called, or the file is closed.

The end-of-file indicator is reset with each input operation. Ways of Detecting End of File A ] In Text File :

Special Character EOF denotes the end of File

As soon as Character is read, End of the File can be detected

EOF is defined in stdio.h

Equivalent value of EOF is -1

Printing Value of EOF :

```
void main()
```

```
{
```

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

```
}
```

B ] In Binary File :

feof function is used to detect the end of file

feof Returns TRUE if end of file is reached

Syntax :

```
int feof(FILE *fp);
```

```
if(feof(fp) == 1) // as if(1) is TRUE
```

```
printf("End of File");
```

Way 2 : In While Loop

```
while(!feof(fp))
```

```
{
```

-----

-----

```
}
```

The C standard library I/O functions allow you to read and write data to both files and devices. There are no predefined file structures in C, all data being

treated as a sequence of bytes. These I/O functions may be broken into two different categories : stream I/O and low-level I/O.

The stream I/O functions treat a data file as a stream of individual characters. The appropriate stream function can provide buffered, formatted or unformatted input and output of data, ranging from single characters to complicated structures. Buffering streamlines the I/O process by providing temporary storage for data which takes away the burden from the system of writing each item of data directly and instead allows the buffer to fill before causing the data to be written.

The low-level I/O system on the other hand does not perform any buffering or formatting of data --instead it makes direct use of the system's I/O capabilities to transfer usually large blocks of information.

### **Stream I/O**

The C I/O system provides a consistent interface to the programmer independent of the actual device being accessed. This interface is termed a **stream** in C and the actual device is termed a **file**. A device may be a disk or tape drive, the screen, printer port, etc. but this does not bother the programmer because the stream interface is designed to be largely device independent. All I/O through the keyboard and screen that we have seen so far is in fact done through special standard streams called **stdin** and **stdout** for input and output respectively. So in essence the console functions that we have used so far such as `printf()`, etc. are special case versions of the file functions we will now discuss.

There are two types of streams : text and binary. These streams are basically the same in that all types of data can be transferred through them however there is one important difference between them as we will see.

#### **Text Streams**

A text stream is simply a sequence of characters. However the characters in the stream are open to translation or interpretation by the host environment. For example the newline character, '`\n`', will normally be converted into a carriage return/linefeed pair and `^Z` will be interpreted as EOF. Thus the number of characters sent may not equal the number of characters received.

#### **Binary Streams**

A binary stream is a sequence of data comprised of bytes that will not be interfered with so that a one-to-one relationship is maintained between data sent and data received.

### Common File Functions

|                     |                                                |
|---------------------|------------------------------------------------|
| fopen()             | open a stream                                  |
| fclose()            | close a stream                                 |
| putc()& fputc()     | write a character to a stream                  |
| getc()& fgetc()     | read a character from a stream                 |
| fprintf()& fscanf() | formatted I/O                                  |
| fgets() & fputs()   | string handling                                |
| fseek()             | position the file pointer at a particular byte |
| feof()              | tests if EOF                                   |

### Opening and Closing Files

A stream is associated with a specific file by performing an open operation. Once a file is opened information may be exchanged between it and your program. Each file that is opened has a unique file control structure of type FILE ( which is defined in <stdio.h> along with the prototypes for all I/O functions and constants such as EOF (-1) ). A **file pointer** is a pointer to this FILE structure which identifies a specific file and defines various things about the file including its name, read/write status, and current position. A file pointer variable is defined as follows

```
FILE *fptr ;
```

The fopen() function opens a stream for use and links a file with that stream returning a valid file pointer which is positioned correctly within the file if all is correct. fopen() has the following prototype

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

where filename is a pointer to a string of characters which make up the name and path of the required file, and mode is a pointer to a string which specifies how the file is to be opened. The following table lists some values for mode.

|     |                                                        |
|-----|--------------------------------------------------------|
| r   | opens a text file for reading (must exist)             |
| w   | opens a text file for writing (overwritten or created) |
| a   | append to a text file                                  |
| rb  | opens a binary file for reading                        |
| wb  | opens a binary file for writing                        |
| ab  | appends to a binary file                               |
| r+  | opens a text file for read/write (must exist)          |
| w+  | opens a text file for read/write                       |
| a+  | append a text file for read/write                      |
| rb+ | opens a binary file for read/write                     |
| wb+ | opens a binary file for read/write                     |
| ab+ | append a binary file for read/write                    |

If fopen( ) cannot open "test.dat" it will return a NULL pointer which should always be tested for as follows.

```
FILE *fp ;
if ((fp = fopen("test.dat", "r")) == NULL)
{
 puts("Cannot open file");
 exit(1);
}
```

This will cause the program to be exited immediately if the file cannot be opened.

The fclose() function is used to disassociate a file from a stream and free the stream for use again.

```
fclose(fp);
```

fclose() will automatically flush any data remaining in the data buffers to the file.

### Reading & Writing Characters

Once a file pointer has been linked to a file we can write characters to it using the fputc() function.

```
fputc(ch, fp);
```

If successful the function returns the character written otherwise EOF.  
Characters may be read from a file using the fgetc() standard library function.

```
ch = fgetc(fp);
```

When EOF is reached in the file fgetc( ) returns the EOF character which informs us to stop reading as there is nothing more left in the file.

For Example :- Program to copy a file byte by byte

```
#include <stdio.h>

void main()
{
 FILE *fin, *fout ;
 char dest[30], source[30], ch ;
 puts("Enter source file name");
 gets(source);
 puts("Enter destination file name");
 gets(dest) ;
 if ((fin = fopen(source, "rb")) == NULL) // open as binary as we
 don't
 { // know what is in file
 puts("Cannot open input file ");
 puts(source) ;
```

```
 exit(1) ;

}

if ((fout = fopen(dest, "wb")) == NULL)

{

 puts("Cannot open output file ") ;

 puts(dest) ;

 exit(1) ;

}

while ((ch = fgetc(fin)) != EOF)

 fputc(ch , fout) ;

fclose(fin) ;

fclose(fout) ;

}
```

When any stream I/O function such as fgetc() is called the current position of the file pointer is automatically moved on by the appropriate amount, 1 character/ byte in the case of fgetc() ;

### Working with strings of text

This is quite similar to working with characters except that we use the functions fgets() and fputs() whose prototypes are as follows :-

```
int fputs(const char *str, FILE *fp) ;

char *fgets(char *str, int maxlen, FILE *fp) ;
```

For Example : Program to read lines of text from the keyboard, write them to a file and then read them back again.

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

```

void main()
{
 char file[80], string[80] ;
 FILE *fp ;
 printf("Enter file Name : ");
 gets(file);
 if ((fp = fopen(file, "w")) == NULL)//open for writing
 {
 printf("Cannot open file %s", file) ;
 exit(1) ;
 }
 while (strlen (gets(str)) > 0)
 {
 fputs(str, fp) ;
 fputc('\n', fp) ; /* must append \n for readability -- not stored
by gets() */
 }

 fclose(fp) ;
 if ((fp = fopen(file, "r")) == NULL)//open for reading
 {
 printf("Cannot open file %s", file) ;
 exit(1) ;
 }
 while (fgets(str, 79, fp) != EOF)// read at most 79 characters
 puts(str) ;
 fclose(fp) ;
}

```

### Formatted I/O

For Example :- Program to read in a string and an integer from the keyboard, write them to a disk file and then read and display the file contents on screen.

```

#include <stdio.h>
#include <stdlib.h>
void main()
{
 FILE *fp ;
 char s[80] ;

 int t ;

```

```

if ((fp = fopen("test.dat", "w")) == NULL)
{
 puts("Cannot open file test.dat");
 exit(1);
}
puts("Enter a string and a number");
scanf("%s %d", s, &t);
fprintf(fp, "%s %d", s, t);
fclose(fp);
if ((fp = fopen("test.dat", "r")) == NULL)
{
 puts("Cannot open file");
 exit(1);
}
fscanf(fp, "%s %d", s, &t);
printf("%s, %d\n", s, t);
fclose(fp);
}

```

There are several I/O streams opened automatically at the start of every C program.

|        |     |                                                 |
|--------|-----|-------------------------------------------------|
| stdin  | --- | standard input ie. keyboard                     |
| stdout | --- | standard output ie. screen                      |
| stderr | --- | again the screen for use if stdout malfunctions |

It is through these streams that the console functions we normally use operate.  
For example in reality a normal printf call such as

```
printf("%s %d", s, t);
```

is in fact interpreted as

```
fprintf(stdout, "%s %d", s, t);
```

### **fread() and fwrite()**

These two functions are used to read and write blocks of data of any type.

Their prototypes are as follows where size\_t is equivalent to unsigned.

|        |                                                                        |
|--------|------------------------------------------------------------------------|
| size_t | fread( void *buffer, size_t num_bytes, size_t count, FILE *fp );       |
| size_t | fwrite( const void *buffer, size_t num_bytes, size_t count, FILE *fp ) |

where **buffer** is a pointer to the region in memory from which the data is to be read or written respectively, **num\_bytes** is the number of bytes in each item to be read or written, and **count** is the total number of items ( each num\_bytes long ) to be read/written. The functions return the number of items successfully read or written.

For Example :-

```

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

struct tag {
 float balance ;
 char name[80] ;
} customer = { 123.232, "John" } ;

void main()
{
FILE *fp ;
double d = 12.34 ;
int i[4] = { 10 , 20, 30, 40 } ;
if ((fp = fopen ("test.dat", "wb+"))==NULL)
{
 puts("Cannot open File") ;
 exit(1) ;
}
fwrite(&d, sizeof(double), 1, fp) ;
fwrite(i, sizeof(int), 4, fp) ;
fwrite(&customer, sizeof(struct tag), 1, fp) ;
rewind(fp) ; /* repositions file pointer to start */
fread(&d, sizeof(double), 1, fp) ;
fread(i, sizeof(int), 4, fp) ;
fread(&customer, sizeof(struct tag), 1, fp) ;
fclose(fp) ;
}

```

Unlike all the other functions we have encountered so far fread and fwrite read and write **binary** data in the same format as it is stored in memory so if we try to edit one these files it will appear completely garbled. Functions like fprintf, fgets, etc. read and write displayable data. fprintf will write a double as a series of digits while fwrite will transfer the contents of the 8 bytes of memory where the double is stored directly.

## 11.5 Explain Random Access Files

### ***Random Access I/O***

The fseek() function is used in C to perform random access I/O and has the following prototype.

```
int fseek (FILE *fp, long num_bytes, int origin) ;
```

where **origin** specifies one of the following positions as the origin in the operation

|          |     |                   |
|----------|-----|-------------------|
| SEEK_SET | --- | beginning of file |
| SEEK_CUR | --- | current position  |
| SEEK_END | --- | EOF               |

and where **num\_bytes** is the offset in bytes to the required position in the file. `fseek()` returns zero when successful, otherwise a non-zero value.

For Example if we had opened a file which stored an array of integers and we wish to read the 50<sup>th</sup> value we might do the following

```
fseek (fp, (49 * sizeof(int)), SEEK_SET) ;
fscanf(fp, "%d", &i) ;
```

from anywhere in the program.

### 11.6 Explain Error Handling in File I/O

Low level file input and output in C does not perform any formatting or buffering of data whatsoever, transferring blocks of anonymous data instead by making use of the underlying operating system's capabilities.

Low level I/O makes use of a file handle or descriptor, which is just a non-negative integer, to uniquely identify a file instead of using a pointer to the FILE structure as in the case of stream I/O.

As in the case of stream I/O a number of standard files are opened automatically :-

|                        |     |          |
|------------------------|-----|----------|
| <b>standard input</b>  | --- | <b>0</b> |
| <b>standard output</b> | --- | <b>1</b> |
| <b>standard error</b>  | --- | <b>2</b> |

The following table lists some of the more common low level I/O functions, whose prototypes are given in `<io.h>` and some associated constants are contained in `<fcntl.h>` and `<sys\stat.h>`.

|         |                                  |
|---------|----------------------------------|
| open()  | opens a disk file                |
| close() | closes a disk file               |
| read()  | reads a buffer of data from disk |
| write() | writes a buffer of data to disk  |

The open function has the following prototype and returns -1 if the open operation fails.

```
int open (char *filename, int oflag [, int pmode]);
```

where filename is the name of the file to be opened, oflag specifies the type of operations that are to be allowed on the file, and pmode specifies how a file is to be created if it does not exist.

**oflag** may be any logical combination of the following constants which are just *bit flags* combined using the bitwise OR operator.

|          |                                        |
|----------|----------------------------------------|
| O_APPEND | appends to end of file                 |
| O_BINARY | binary mode                            |
| O_CREAT  | creates a new file if it doesn't exist |
| O_RDONLY | read only access                       |
| O_RDWR   | read write access                      |
| O_TEXT   | text mode                              |
| O_TRUNC  | truncates file to zero length          |
| O_WRONLY | write only access                      |

This will actually set the read / write access permission of the file at the operating system level permanently unlike oflag which specifies read / write access just while your program uses the file.

The close() function has the following prototype

```
int close (int handle);
```

and closes the file associated with the specific handle.

The read() and write() functions have the following prototypes

```
int read(int handle, void *buffer, unsigned int count);
```

```
int write(int handle, void *buffer, unsigned int count);
```

where handle refers to a specific file opened with open(), buffer is the storage location for the data ( of any type ) and count is the maximum number of bytes to be read in the case of read() or the maximum number of bytes written in the case of write(). The function returns the number of bytes actually read or written or -1 if an error occurred during the operation.

Example : Program to read the first 1000 characters from a file and copy them to another.

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
void main()
{
 char buff[1000];
 int handle ;
 handle=open(" test.dat", O_BINARY|O_RDONLY, S_IREAD | S_IWRITE);
 if (handle == -1) return ;
 if (read(handle, buff, 1000)== 1000)
 puts("Read successful");
 else
{
```

```

 puts(Read failed") ;

 exit(1);
}
close(handle) ;
handle = open("test.bak",
 O_BINARY|O_CREAT|O_WRONLY| O_TRUNC,
 S_IREAD | S_IWRITE) ;
if (write(handle, buff, 1000) == 1000)
 puts("Write successful") ;
else
{
 puts("Write Failed") ;
 exit(1) ;
}

close(handle) ;
}

```

Low level file I/O also provides a seek function **lseek** with the following prototype.

```
long _lseek(int handle, long offset, int origin);
```

**\_lseek** uses the same origin etc. as *fseek()* but unlike *fseek()* returns the offset, in bytes, of the new file position from the beginning of the file or -1 if an error occurs.

For Example : Program to determine the size in bytes of a file.

```

#include <stdio.h>
#include <io.h>

#include <fcntl.h>

#include <sys\stat.h>

void main()

{
 int handle ;

 long length ;

 char name[80] ;

```

```
printf("Enter file name : ") ;
gets(name) ;

handle=open(name,O_BINARY| O_RDONLY, S_IREAD |
S_IWRITE);
lseek(handle, 0L, SEEK_SET) ;
length = lseek(handle, 0L, SEEK_END) ;
close(handle) ;
printf("The length of %s is %ld bytes \n", name, length) ;
}
```

### 11.7 Explain Redirection.

One way to get input into a program or to display output from a program is to use *standard input* and *standard output*, respectively. All that means is that to read in data, we use `scanf()` (or a few other functions) and to write out data, we use `printf()`.

When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

```
% a.out < inputfile
```

This allows us to use the same `scanf()` calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., `inputfile` above) instead of the keyboard.

Similarly, there is *output redirection*:

```
% a.out > outputfile
```

that allows us to use `printf()` as before, but that causes the output of the program to go to a file (e.g., `outputfile` above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

```
% a.out < inputfile > outputfile
```

---

---

END OF CHAPTER 11

## Multiple Choice Questions.

Select the best answer.

**1. Select which is true about a stream...**

- a) it is a general name given to a flow of data
- b) it is simply a sequence of bytes.
- c) it is a logical interface to the data file.
- d) all of these.

**2. Which of the following function sends output to a file.**

- a) printf()
- b) fprintf()
- c) scanf()
- d) both (a) & (b)

**3. which of the following mode can create new file for user.**

- a) "w"
- b) "w+"
- c) "r+"
- d) "a-\*\_"

**4. Which of the following is used as end of string?**

- a) \ri
- B) \O
- C) /O
- D) /n

**5. Which of the following functions is used to write a string in a file?**

- a) puts()
- b) putc()
- c) fputs()
- d) fgets()

**6. Which of the following true about FILE \*fp**

- A. FILE is a keyword in C for representing files and fp is a variable of FILE type.
- B. FILE is a stream
- C. FILE is a buffered stream
- D. FILE is a structure and fp is a pointer to the structure of FILE type

**7. Which of the following mode argument is used to truncate?**

- A. a
- B. w

- C. f
- D. t

**8. The first and second arguments of fopen() are**

- A. A character string containing the name of the file & the second argument is the mode
- B. A character string containing the name of the user & the second argument is the mode
- C. A character string containing file pointer & the second argument is the mode
- D. None of the mentioned

**9. A files is stored in**

- a) RAM
- b) hard disk
- c) Rom
- d) cache

**10. Which of the following mode open only an existing file for both reading and writing**

- a) "w"
- b) "w+"
- c) "r+"
- d) "a+"

**11. which of the following function is used to write a string to a file?**

- a) puts()
- b) putc()
- c) fputs ()
- d) fgets()

**12. On successfully closing a file, the fclose() returns**

- a) .NULL
- b) 0 (zer0)
- c) 1 (one)
- d) FILE pointer

**13. An array subscript should be**

- a) int
- b) float
- c) double
- d) an array

**14. The two main systems of I/O available in C are \_\_\_\_\_ I/O .**

- a) Standard
- b) Linear
- c) System
- d. None of these.

**15. The first three ways to store the following data**

- a) Text
- b) Numbers
- c) ASCII
- d) All

**16. The fourth way, record I/O causes numerical data to be stored in \_\_\_\_\_ format.**

- a) Text
- b) Integers
- c) Binary format
- d) None of these

**17. In standard I/O the function used to close a file is**

- a) fputs()
- b) fgets()
- c) fclose()
- d) fwrite()

**18. In mode C newlines are translated into MS-DOS CR/LF pair, and the character IA indicates an EOF**

- a) Binary
- b) Text
- c) Both a and b
- d) None

**19. An oflag can**

- a) Signal when a file is unreadable
- b) specify if a file should be read or written
- c) Signal when EOF is detected
- d) specify binary mode

**20. The advantage of using redirection is that the input to a program may then come from.**

- a) The keyboard
- b) pipes
- c) filters
- d) other programs

## **QUESTIONS WITH SHORT ANSWERS**

**Q1. What are data files?**

The file in which data to be input to the computer program is stored are called data files. The data output by a computer program is also stored in data files.

**Q2. What is the use of data file?**

Computer program often need large amounts of input data. It is very difficult to input the data from the keyboard each time the program is executed. In such cases, the input data is stored in a file on the disk. The computer program reads the data from the data file.

**Q3. Write two types of file access methods?.**

Sequential Access Method and Direct or Random Access Method.

**Q4. What are input and output streams?**

Stream is a general term. It refers to flow of data from a source to a destination. The process of inputting data from the source is known as

reading, extracting, getting or fetching. The process of outputting data to the destination is known as writing, inserting, putting or storing.

#### **Q.5 What is text stream?**

A text stream is a sequence of characters. In a text stream, certain character conversions occur. For example, the new line is represented as a carriage return/line feed pair. This means that there may not be a one-to-one correspondence between the characters written and those saved on an external device.

#### **Q. 6 What is binary stream?**

A binary stream is a sequence of bytes. The number of bytes written or read is always the same as those on the external device. This means that there is a one-to-one correspondence between the bytes read or written and those on the external device.

#### **Q.7 Compare text stream and binary stream?.**

A text stream is a sequence of characters. A binary stream is a sequence of bytes.

#### **Q.8 What is a text file?**

A text file is a file that only contains readable and printable character and has no special formatting such as bold text, italic text, images etc.

#### **Q.9 Distinguish between BOF and EOF?**

A data file in C consists of a series of bytes. It has a beginning and an end. The start of the first byte in the file is called Beginning of File(BOF). The end of the last byte is called End of File(EOF).

#### **Q.10 What is a file pointer?**

A pointer is basically a variable like any other variable. However, what is different about a pointer is that instead of containing actual data. It contains the address of ( a pointer to) the memory location where information can be found. It is used in file read & write operation in C.

## LONG QUESTION

- Q.1 Explain Formatted File input and output? What functions are used in C for formatted file input and output?
- Q.2 Explain fgets() function?
- Q.3 Explain how data is read/written one string at a time in a data file in C Language?
- Q.4 What are string variables? How are they declared?
- Q.5 How is the End-of-File detected in a data file?.

## LARGER PROGRAMS

12

### **12.1 Making Stand -alone Executable**

In this book our programs have consisted of a single file. We have written, compiled, and linked this one file to create an executable program. The program might have had several functions in it, but they were all treated as a single entity for the purpose of compilation. However, it is possible to break a program apart into separate files, each file consisting of one or more functions.

#### **Compiling and Linking**

##### **Description:**

C programs are written in human readable source code that is not directly executable by a computer. It takes a three step process to transform the source code into executable code. These three steps are: preprocessing, compiling and linking.

- **preprocessing** - Processes directives (commands that begin with a # character) which can modify the source code before it is compiled.
- **Compiling** - The modified source code is compiled into binary object code. This code is not yet executable.
- **Linking** - The object code is combined with required supporting code to make an executable program. This step typically involves adding in any libraries that are required.

Let's examine some simple examples of this process; then we'll discuss why we might want to use separate compilation. Here's a complete C source file, called mainprogram.c.

```
// mainprog.c

// main program to test separate compilation

#include <stdio.h> //for printf(), scanf()
```

```
int formula (int, int) ; //prototype

void main (void)
{
 int a, b, ans;

 printf ("Type two integers: ");

 scanf ("%d%d", &a, &b);

 ans = formula (a, b);

 printf ("The sum of the squares is %d.", ans);
}
```

“Functions,” multifun.c, which calculated the sum of the squares of two numbers. However, part of the multifun.c program is missing here. We’ll see where it went in a moment.

Type in this source file with the Turbo C++ editor and save it as mainprog.c. You can compile this file with no problem, but you can’t link it. Why not? If you try it, the linker displays the error message.

Undefined symbol formula() in module mainprog.c

The linker doesn’t know what to do when it encounters the reference to the function formula() in the next-to-last statement of the listing.

```
ans = formula (a, b);
```

We’re going to solve this problem by providing this function but in a separate file. Type in and save the following file, giving it the name formula.c. Remember that you can have two or more files open at the same time in the IDE, in separate windows. It’s convenient to keep both mainprog.c and formula.c available this way.

```
// formula.c

// function returns sum of squares of arguments
```

```
int formula (int x, int y)
{
 return (x*x + y*y);
}
```

The formula.c file contains, as you can see, the missing function, formula()

How do we turn these two source files mainprog.c and formula.c into a complete program? We use an unexamined capability of Turbo C++.

## 12.2 Separate Compilation

In our example, in which mainprog() and formula() were linked, two numbers were passed from mainprog() to formula() using arguments, and the result from formula() was returned to mainprog() with a return statement. This kind of data communication between separately compiled functions is easy to implement. It requires no special statement in either function, but suppose we wanted functions in separately compiled files to have access to the same external variable?

A function in file B can access an external variable defined in file A. provided that file B declares the variable using the keyword `extern`.

### 1) Define and declare:

What do we mean by declare? Let's pause briefly to make sure we understand the distinction between define and declare. When a variable is defined, it is given a data type (int or whatever) and a name. Also ----this is the key point--- memory is set aside for it , it is given physical existence. Defining variables is what we have done so far in this book.

However, a variable can also be declared, which specifies the name and type of the variable, but does not set aside any memory. A declaration is simply an announcement that a variable with a particular name and type exists. A variable can be defined only once, but it can be declared many times.

To show how this works, we'll rewrite the mainpro.c and formula.c files this way:

```
// mainpro2.c

// main program to test separate compilation

#include <stdio.h> //for printf() and scanf()

int formula (void) ; //prototype

int a, b;

//definition of external variables

void main (void)

{

int ans;

printf ("Type two integers: ");

scanf ("%d%d", &a, &b);

ans = formula () ;

printf ("The sum of the squares is %d." ,

ans) ;

}
```

This is similar to the mainprog.c file shown earlier, except that we have moved the variables a and b, which hold the two integers entered by the user, to a position outside the function, making them external variables. Now when mainpro2.c calls formula2, it no longer needs to pass any arguments.

Here's the revised formula.c file:

```
// formula2.c

// function returns sum of squares of arguments
```

```
// uses global variables for input

int formula (void)

{

extern int a, b; //declaration of external variables

return (a*a + b*b) ;

}
```

Here you can see that we no longer need to declare any variables as formal arguments to the function but that we do declare the two variables a and b to be of type extern int. What does this mean?

## 2) The Extern Keyword:

“Memory” an external variable is visible in the file in which it’s defined, from the point if the definition onward. For an external variable to be visible in a file other than the one in which it’s defined, it must be declared using the extern keyword. This keyword tells the compiler: “Somewhere in another file we have defined this variable. Don’t worry that it isn’t defined in this file but leave a message for the linker about it.”

When the linker gets this message from the computer, it looks for the definition of the variable in all the other files. When the linker finds the definition, it makes the appropriate connections. In this example that means that references to a and b in the formula2.c file will be correctly linked to the definitions of a and b in the mainpro2.c file. To combine these two files, create a project file called sumsqr2.prj , whose contents are mainpro2.c and formula2.c. Then make your project as described above.

## 12.3 Conditional Compilation

### 12.3.1 Preprocessor Directives

#### Description:

The preprocessor will process directives that are inserted into the C source code. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself.

Preprocessor directives begin with a hash (#) symbol and may have several arguments.

The following are some of the preprocessor directives that you can use in your source code:

| Directive       | Description                                                                                                                   | Example            |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------|--------------------|
| <u>#include</u> | Include another C file into the current file at the location of the #include statement prior to compiling the source code.    | #include <stdio.h> |
| <u>#define</u>  | Define a macro which can be used as a constant throughout the source code.                                                    | #define AGE 50     |
| <u>#undef</u>   | Clear a macro which was previously defined.                                                                                   | #undef AGE         |
| <u>#if</u>      | Conditional expression which can be used to include source code for compilation.                                              | #if AGE > 50       |
| <u>#ifdef</u>   | Allows the inclusion of source code if the provided macro identifier has been defined. Equivalent to #if defined(identifier). | #ifdef SOLARIS     |
| <u>#ifndef</u>  | Allows the inclusion of source code if the provided macro identifier has not been defined.                                    | #ifndef WINDOWS    |
| <u>#elif</u>    | Provides an alternate inclusion of source code when used with the #if, #ifdef, or #ifndef                                     | #elif YEARS_OLD >  |

|        |                                                                                                                        |        |
|--------|------------------------------------------------------------------------------------------------------------------------|--------|
|        | directives and the #elif condition evaluates to true.                                                                  | 10     |
| #else  | Allows the inclusion of source code if the preceeding #if, #ifdef, or #ifndef directive expression evaluates to false. | #else  |
| #endif | Signals the end of a #if, #ifdef or #ifndef condition .                                                                | #endif |

## Explanation:

### 1. #include Directive

#### Description:

In the C Programming Language, the #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outside of the current source file.

#### Syntax

The syntax for the #include directive in the C language is:

`#include <header_file>`

OR

`#include "header_file"`

#### **header\_file**

The name of the header file that you wish to include. A header file is a C file that typically ends in ".h" and contains declarations and macro definitions which can be shared between several source files.

**Note:** The difference between these two syntaxes is subtle but important. If a header file is included within <>, the preprocessor will search a predetermined directory path to locate the header file. If the header file is enclosed in "", the preprocessor will look for the header file in the same directory as the source file.

**Example:**

Let's look at an example of how to use #include directives in your C program.

In the following example, we are using the #include directive to include the *stdio.h* header file which is required to use the printf standard C library function in your application.

```
/* Example using #include directive */
#include <stdio.h>
int main()
{
 /*
 * Output "C Programming in 2000" using the C standard library printf
 * function
 * defined in the stdio.h header file
 */
 printf("C Programming in %d\n", 2000);
 return 0;
}
```

In this example, the program will output the following:

C Programming in 2000

## 2. #define Directive (macro definition)

**Description:**

In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.

Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

### Syntax

The syntax for creating a **constant** using #define in the C language is:

#define CNAME value

OR

#define CNAME (expression)

CNAME

The name of the constant. Most C programmers define their constant names in uppercase, but it is not a requirement of the C Language.

value

The value of the constant.

expression

Expression whose value is assigned to the constant. The expression must be enclosed in parentheses if it contains operators.

**Note:**

Do NOT put a semicolon character at the end of #define statements. This is a common mistake.

### **Example:**

Let's look at how to use #define directives with numbers, strings, and expressions.

#### **Number**

The following is an example of how you use the #define directive to define a numeric constant:

**#define AGE 10**

In this example, the constant named *AGE* would contain the value of 10.

#### **String**

You can use the #define directive to define a string constant.

For example:

**#define NAME "Programming"**

In this example, the constant called *NAME* would contain the value of "Programming".

Below is an example C program where we define these two constants:

```
#include <stdio.h>
#define NAME "Programming"
#define AGE 10
int main()
{
 printf("%s is over %d years old.\n", NAME, AGE);
 return 0;
}
```

This C program would print the following:

**Programming is over 10 years old.**

### **Expression:**

You can use the #define directive to define a constant using an expression.

For example:

**#define AGE (20 / 2)**

In this example, the constant named *AGE* would also contain the value of 10.

Below is an example C program where we use an expression to define the constant:

```
#include <stdio.h>
#define AGE (20 / 2)
int main()
{
 printf("Programming is over %d years old.\n", AGE);
 return 0;
}
```

This C program would also print the following:

**Programming is over 10 years old.**

### **3. #undef Directive**

**Description:**

In the C Programming Language, the #undef directive tells the preprocessor to remove all definitions for the specified macro. A macro can be redefined after it has been removed by the #undef directive.

Once a macro is undefined, an **#ifdef directive** on that macro will evaluate to false.

**Syntax**

The syntax for the #undef directive in the C language is:

**#undef macro\_definition**

**macro\_definition**

The name of the macro which will be removed by the preprocessor.

**Example:**

The following example shows how to use the #undef directive:

```
/* Example using #undef directive by Programming */

#include <stdio.h>

#define YEARS_OLD 12

#undef YEARS_OLD

int main()

{

 #ifdef YEARS_OLD

 printf("Programming is over %d years old.\n", YEARS_OLD);

 #endif

 printf("Programming is a great resource.\n");
```

```
 return 0;
```

```
}
```

In this example, the YEARS\_OLD macro is first defined with a value of 12 and then undefined using the **#undef** directive. Since the macro no longer exists, the statement **#ifdef** YEARS\_OLD evaluates to false. This causes the subsequent **printf function** to be skipped.

### Output:

Programming is a great resource

## 4. #if Directive

### Description:

In the C Programming Language, the **#if** directive allows for conditional compilation. The preprocessor evaluates an expression provided with the **#if** directive to determine if the subsequent code should be included in the compilation process.

### Syntax:

The syntax for the **#if** directive in the C language is:

**#if conditional\_expression**

**conditional\_expression**

Expression that the preprocessor will evaluate to determine if the C source code that follows the **#if** directive will be included into the final compiled application.

### Note:

The **#if** directive must be closed by an **#endif directive**.

### Example:

The following example shows how to use the **#if** directive in the C language:

```
/* Example using #if directive by Programming */
```

```
#include <stdio.h>
#define WINDOWS 1
int main()
{
 printf("Programming is a great ");
 #if WINDOWS
 printf("Windows ");
 #endif
 printf("resource.\n");
 return 0;
}
```

Here is the output of the executable program:

Programming is a great Windows resource.

## 5. #ifdef Directive

### Description:

In the C Programming Language, the #ifdef directive allows for conditional compilation. The preprocessor determines if the provided macro exists before including the subsequent code in the compilation process.

### Syntax

The syntax for the #ifdef directive in the C language is:

`#ifdef macro_definition`

### **macro\_definition**

The macro definition that must be defined for the preprocessor to include the C source code into the compiled application.

### Note:

The #ifdef directive must be closed by an `#endif directive`.

**Example:**

The following example shows how to use the #ifdef directive in the C language:

```
/* Example using #ifdef directive by Programming */
#include <stdio.h>
#define YEARS_OLD 10
int main()
{
 #ifdef YEARS_OLD
 printf("Programming is over %d years old.\n", YEARS_OLD);
 #endif
 printf("Programming is a great resource.\n");
 return 0;
}
```

Here is the output of the executable program:

**Programming is over 10 years old.**

**Programming is a great resource.**

A common use for the #ifdef directive is to enable the insertion of platform specific source code into a program.

The following is an example of this:

```
/* Example using #ifdef directive for inserting platform specific source code
by Programming */
#include <stdio.h>
#define UNIX 1
int main()
{
 #ifdef UNIX
 printf("UNIX specific function calls go here.\n");
 #endif
 printf("Programming is over 10 years old.\n");
 return 0;
}
```

The output of this program is:

**UNIX specific function calls go here.****Programming is over 10 years old.**

In this example, the UNIX source code is enabled. To disable the UNIX source code, change the line #define UNIX 1 to #undef UNIX.

**6. #endif Directive****Description:**

In the C Programming Language, the #endif directive closes off the following directives: #if, #ifdef, or #ifndef. When the #endif directive is encountered, preprocessing of the opening directive (#if, #ifdef, or #ifndef) is completed.

**Syntax**

The syntax for the #endif directive in the C language is:

**#endif****Example:**

The following example shows how to use the #endif directive in the C language:

```
/* Example using #endif directive by Programming */
#include <stdio.h>
#define WINDOWS 1
int main()
{
 printf("Programming is a great ");
 #if WINDOWS
 printf("Windows ");
 #endif
 printf("resource.\n");
 return 0;
}
```

Here is the output of the executable program:

**Programming is a great Windows resource.**

## 7. #else Directive

### Description:

In the C Programming Language, the #else directive provides an alternate action when used with the #if, #ifdef, or #ifndef directives. The preprocessor will include the C source code that follows the #else statement when the condition for the #if, #ifdef, or #ifndef directive evaluates to false.

### Syntax

The syntax for the #else directive in the C language is:

**#else**

### Note:

The #else directive must be closed by an #endif directive.

### Example:

The following example shows how to use the #else directive in the C language:

```
/* Example using #else directive by Programming */
#include <stdio.h>
#define YEARS_OLD 12
int main()
{
 #if YEARS_OLD < 10
 printf("Programming is a great resource.\n");
 #else
 printf("Programming is over %d years old.\n", YEARS_OLD);
 #endif
 return 0;
}
```

Here is the output of the executable program:

**Programming is over 12 years old.**

## 8. #elif Directive

**Description:**

In the C Programming Language, the #elif provides an alternate action when used with the #if, #ifdef, or #ifndef directives. The preprocessor will include the C source code that follows the #elif statement when the condition of the preceding #if, #ifdef or #ifndef directive evaluates to false and the #elif condition evaluates to true.

The #elif directive can be thought of as #else if.

**Syntax**

The syntax for the #elif directive in the C language is:

**#elif conditional\_expression**

**conditional\_expression**

Expression that must evaluate to true for the preprocessor to include the C source code into the compiled application.

**Note:**

The #elif directive must be closed by an #endif directive.

**Example:**

The following example shows how to use the #elif directive in the C language:

```
/* Example using #elif directive by Programming */
#include <stdio.h>
#define YEARS_OLD 12
int main()
{
 #if YEARS_OLD <= 10
 printf("Programming is a great resource.\n");
 #elif YEARS_OLD > 10
 printf("Programming is over %d years old.\n", YEARS_OLD);
 #endif
 return 0;
}
```

In this example, YEARS\_OLD has a value of 12 so the statement **#if YEARS\_OLD <=10** evaluates to false. As a result, processing is passed to

the `#elif YEARS_OLD > 10` statement which evaluates to true. The C source code following the `#elif` statement is then compiled into the application.

Here is the output of the executable program:

**Programming is over 12 years old.**

## 9. #ifndef Directive

The reverse of the `#ifdef` directive is `#ifndef`, which tells the compiler to compile the code that follows it only if a constant is not defined. You'll find this directive used frequently in the Turbo C++ include files to avoid including another file more than once. Here's the idea:

```
#ifndef NULL

#include <null.h>

#endif
```

The constant `NULL` may be needed in several include files(`stdio.h` and `stdlib.h`, for example). It could be defined in both files, but suppose the programmer includes both files in his program? It's bad form to `#define` something more than once, so `NULL` is defined in the `_null.h` file, and any file that wants to use this definition includes this file conditionally, if `NULL` is already defined, the file is not included, thus avoiding multiple definitions.

Conditional compilation comes in handy if we require an output only when certain condition(s) hold true. Normally we use `if` keyword for checking some condition so we have to use something different so that compiler can determine whether to compile the code or not. The different thing is `#if`.

It is the process of defining **compiler** directives that cause different parts of the code to be **compiled**, and others to be ignored. This technique can be used in a cross-platform development scenario to specify parts of the code that are **compiled** specific to a particular platform.

Many **programming languages** support conditional compilation. Typically **compiler directives** define or "undefine" certain variables; other directives test these variables and modify compilation accordingly.

In C and some languages with a similar syntax, this is done using an `#ifdef` directive.

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands `#ifdef` and `#endif`, which have the general form:

```
#ifdef macroname
statement 1 ;
statement 2 ;
statement 3 ;
#endif
```

If `macroname` has been `#defined`, the block of code will be processed as usual; otherwise not.

Now consider the following code to quickly understand the scenario:

**Program:**

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

```
#define x 10

int main()
{
 #ifdef x
 printf("hello\n"); // this is compiled as x is defined
 #else
 printf("bye\n"); // this is not compiled
 #endif

 return 0;
}
```

**Conditional Compilation example in C language**

```
#include <stdio.h>

int main()
{
 #define COMPUTER "An amazing device"

 #ifdef COMPUTER
 printf(COMPUTER);
 #endif
 return 0;
```

}

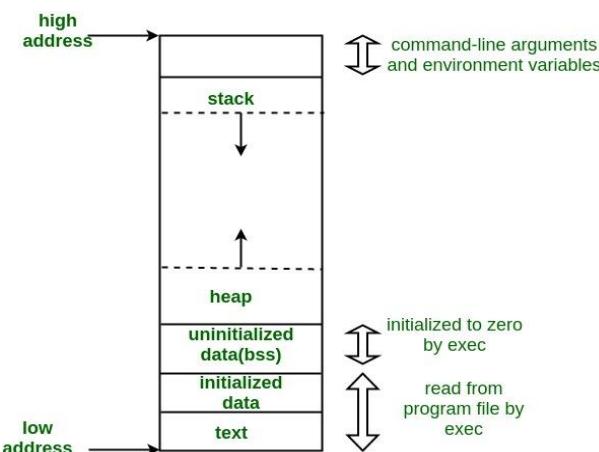
Conditional compilation is used during pre-processing stage to include / exclude code to be compiled depending upon your need. It's very useful in creating platform independent code where porting the code to another platform say from Unix to Windows or to different CPU architecture requires no changes to the source files.

## 12.3 Memory Models

### 12.3.1 Memory Layout of C Programs

A typical memory representation of C program consists of following sections:

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



A typical memory layout of a running process:

#### 1. Text Segment:

A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

## **2. Initialized Data Segment:**

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by `char s[] = "hello world"` in C and a C statement like `int debug=1` outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like `const char* string = "hello world"` makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable `string` in initialized read-write area.

Ex: `static int i = 10` will be stored in data segment and `global int i = 10` will also be stored in data segment.

## **3. Uninitialized Data Segment:**

Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol."

Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared `static int i;` would be contained in the BSS segment.

For instance a global variable declared `int j;` would be contained in the BSS segment.

**4. Stack:**

The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction.

A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

**5. Heap:**

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfil the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

### 12.3.2. Memory Models in C programming

**Memory Models:**

In C there are six type of memory model.

If you want to see all memory model in Turbo C++ IDE then open Turbo C++ IDE and the go:

**Options menu -> Compiler -> Code generation**

**These memory models are:**

- (a) TINY
- (b) SMALL
- (c) MEDIUM
- (d) COMPACT
- (e) LARGE
- (f) HUGE

If you want to change the memory model then go to:

**Options menu -> Compiler -> Code generation**

And select any memory model and click **OK** button.

| Memory model | Default pointer |      |
|--------------|-----------------|------|
|              | Code            | Data |
| TINY         | near            | near |
| SMALL        | near            | near |
| MEDIUM       | far             | near |
| COMPACT      | near            | far  |
| LARGE        | far             | far  |
| HUGE         | far             | far  |

**1. Small** model supports **one data segment** and **one code segment**. All data and code are near by default.

Use the small model for average size applications. The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

**2. Large** model supports **multiple code** and **multiple data** segments. All data and code are far by default.

**3. Medium** and compact models are in-between. Medium model supports **multiple code** and **single data** segments.

The medium model is best for large programs that don't keep much data in memory. Far pointers are used for code but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1MB.

So, there you have it. In the medium model you use far pointers to access code (the code has far call and far return instructions and manipulates with far function pointers) and you can have multiple segments. The standard 16-bit DOS .EXE file format supports multiple segments. In the small model all pointers are near and so you can't and don't change the default code segment in the program.

**4. Compact** model supports **multiple data** segments and a **single code** segment.

**5. Huge** model implies that *individual data items* are **larger** than a single segment, but the implementation of huge data items must be coded by the programmer.

**6. Tiny**-model programs run only under MS-DOS. Tiny model places all data and code in a **single** segment. Therefore, the total program file size can occupy no more than 64K. Since the assembler provides no direct support for this feature, huge model is essentially the same as large model. In each of the models, you can override the default. For example, you can make large *data* items far in small model, or internal procedures near in large model.

## Properties of Memory Model in C:

1. Memory model decides the default type of pointer in C.

**Note:**

**Code:** A pointer to function is called code.

**Data:** A pointer to variable is called data.

**Examples:**

1. What will be output of following C program?

```
#include<stdio.h>
int main(){
 int *ptr;
 printf("%d",sizeof ptr);

 return 0;
}
```

**Output:** Depends upon memory model.

**Explanation:** If memory model is TINY, SMALL or MEDIUM then default pointer will be near and output will be 2 otherwise output will be 4.

2. What will be output of following C program?

```
#include<stdio.h>
int main(){
 char (*fun)();
 printf("%d",sizeof fun);

 return 0;
}
```

**Output:** Depends upon memory model.

**Explanation:** fun is pointer to function. If memory model is TINY, SMALL or COMPACT then default pointer will near and output will be 2 otherwise output will be 4.

### 3. What will be output of following C program?

```
#include<stdio.h>

int main(){
 int near *p,*q;
 printf("%d , %d",sizeof(p),sizeof(q));

 return 0;
}
```

**Output:** 2, Depend upon memory model.

**Explanation:** p is near pointer while type of pointer q will depend what is default type of pointer.

### 4. What will be output of following C program?

```
#include<stdio.h>
int main(){
 char huge **p;
 printf("%d , %d",sizeof(p),sizeof(*p));

 return 0;
}
```

**Output:** 4, Depend upon memory model.

**Explanation:** p is huge pointer while type of pointer \*p will depend what is default type of pointer.

**Write a C program to find the memory model of your computer?**

**Program:**

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

```
int main(){
 #if defined __TINY__
 printf("Memory model is: TINY");
 #elif defined __SMALL__
 printf("Memory model is:SMALL ");
 #elif defined __MEDIUM__
 printf("Memory model is:MEDIUM ");
 #elif defined __COMPACT__
 printf("Memory model is:COMPACT ");
 #elif defined __LARGE__
 printf("Memory model is:LARGE ");
 #elif defined __HUGE__
 printf("Memory model is:HUGE ");
 #endif

 return 0;
}
```

Memory models decide the default size of segment.

**Memory Models:**

**How the segment registers are used and the default size of pointers.**

| Model | Data   | Code | Definition |
|-------|--------|------|------------|
| Tiny* | near   |      | CS=DS=SS   |
| Small | near** | near | DS=SS      |

|         |        |      |                                                             |
|---------|--------|------|-------------------------------------------------------------|
| Medium  | near** | far  | DS=SS, multiple code segments                               |
| Compact | far    | near | single code segment, multiple data segments                 |
| Large   | far    | far  | multiple code and data segments                             |
| Huge    | huge   | far  | multiple code and data segments; single array may be >64 KB |

**Note:**

\* In the Tiny model, all four segment registers point to the same segment.

\*\* In all models with *near* data pointers, SS equals **DS**.

\*\*\* Stack is always limited to at most 64 KByte.

In computing, **Intel Memory Model** refers to a set of six different memory models of the x86 CPU operating in real mode which control how the segment registers are used and the default size of pointers.

Four registers are used to refer to four segments on the 16-bit x86 segmented memory

architecture. **DS** (data segment), **CS** (code segment), **SS** (stack segment), and **ES** (extra segment). Another 16-bit register can act as an offset into a given segment, and so a logical address on this platform is written segment:offset, typically in hexadecimal notation. In real mode, in

order to calculate the physical address of a byte of memory, the hardware shifts the contents of the appropriate segment register 4 bits left (effectively multiplying by 16), and then adds the offset.

For example, the logical address 7522:F139 yields the 20-bit physical address:

$$\begin{array}{r} 75220 \\ + F139 \\ \hline 84359 \end{array}$$

Note that this process leads to aliasing of memory, such that any given physical address has up to 4096 corresponding logical addresses. This complicates the comparison of pointers to different segments.

In protected mode a segment cannot be both writable and executable.[2][3] Therefore, when implementing the Tiny memory model the code segment register must point to the same physical address and have the same limit as the data segment register. This defeated one of the features of the 80286, which makes sure data segments are never executable and code segments are never writable (which means that self-modifying code is never allowed). However, on the 80386, with its paged memory management unit it is possible to protect individual memory pages against writing.[4][5]

Memory models are not limited to 16-bit programs. It is possible to use segmentation in 32-bit protected mode as well (resulting in 48-bit pointers) and there exist C language compilers which support that.[6] However segmentation in 32-bit mode does not allow to access a larger address space than what a single segment would cover, unless some segments are not always present in memory and the linear address space is just used as a cache over a larger segmented virtual space.[citation needed] It allows better protection for access to various objects (areas up to 1 MB long can benefit from a one-byte access protection granularity, versus the coarse 4 KiB granularity offered by sole paging), and is therefore only used in specialized applications, like telecommunications software.[citation needed] Technically, the "flat" 32-bit address space is a "tiny" memory model for the segmented address space. Under both reigns all four segment registers contain one and the same value.

---

END OF CHAPTER 12

---

## **EXERCISE**

### **Multiple Choice Questions:**

- 1. One advantage of separate compilation of program modules is**
  - a) the resulting program will be more modular
  - b) compilation time can be faster
  - c) program modules can be protected from inadvertent alteration during program development
  - d) all of the above
  
- 2. Separately compiled modules are combined using the**
  - a) Linker
  - b) Compiler
  - c) Both a and b
  - d) None of the above
  
- 3. For a variable to be visible in a file other than that where it was defined, it must be \_\_\_\_\_**
  - a) Initialized
  - b) Declared
  - c) Both a and b
  - d) None of the above
  
- 4. For a variable to be visible in a file other than that where it was defined, it must be declared using the keyword \_\_\_\_\_**

- a) printf
- b) int \*ptr
- c) Exam
- d) None of the above

**5. Which of the following are ways to pass information from one function to another function in a separately compiled file?**

- a) as arguments
- b) as static variables
- c) as external variables
- d) a and c

**6. Separate compilation is an aid to modular programming because**

- a) functions cannot access variables in other files
- b) only functions relating to the same program can be combined in one file
- c) external variables can be restricted to one file
- d) None of the above

**7. Conditional compilation is used to**

- a) compile only those programs that are error free
- b) compile some sections of programs and not others
- c) remove comments and compress the source file
- d) none of the above

**8. The \_\_\_\_\_ preprocessor directive is used as a switch to turn compilation on and off.**

- a) #class

- b) #delcare
- c) #define
- d) None of the above

**9. The preprocessor directive #if define (ADV) causes the code that follows it to be compiled only when**

- a) ADV is an integer
- b) ADV is equal to a predefined constant
- c) ADV is TRUE
- d) ADV is #defined

**10. Using different memory models is useful when a program or data becomes too \_\_\_\_\_**

- a) Large
- b) Small
- c) both a and b
- d) none of the above

**11. The medium memory model permits**

- a) more than one code segment
- b) only one data segment
- c) only one code segment
- d) a and b

**12. Memory models are necessary because of the \_\_\_\_\_**

- a) stack-based architecture of the microprocessor
- b) segmentation of memory

c) need to keep programmers guessing

d) b and c

**13. The disadvantage of using a memory model that is larger than necessary is**

a) memory is wasted

b) simple data type cannot be used

c) program files are harder to link

d) program instructions take longer to execute

**14. Which of the following process are used to make an executable file ?**

a) Linking

b) Compile

c) Code written

d) All of the above

**15. \_\_\_\_\_ which calculated the sum of the squares of two numbers.**

a) multifun.h

b) multifun.c

c) multifun.exe

d) multifun.obj

**16. You can have \_\_\_\_\_ files open at the same time in the IDE, in separate windows.**

a) Must be more than two

b) Only one

c) Two or more

d) All of the above

**17. When a variable is \_\_\_\_\_, it is given a data type (int or whatever) and a name.**

a) Defined

b) Declared

c) Both a and b

d) None of the above

**18. \_\_\_\_\_ an external variable is visible in the file in which it's defined, from the**

**point of the definition onward.**

a) String

b) Memory

c) Function

d) None of the above

**19. \_\_\_\_\_ allow the compiler to produce differences in the executable produced**

**controlled by parameters.**

a) Compilation

b) Conditional compilation

c) Decision compilation

d) Both b and c

**20. The \_\_\_\_\_ TIMER directive tells the compiler to compile the statements that follow**

**it only if TIMER is #defined.**

- a) #fprintf()
- b) #fwrite
- c) #ifdef
- d) #ifndef

### Answers:

|              |              |              |              |
|--------------|--------------|--------------|--------------|
| <b>1) d</b>  | <b>2) a</b>  | <b>3) b</b>  | <b>4) c</b>  |
| <b>5) d</b>  | <b>6) c</b>  | <b>7) b</b>  | <b>8) c</b>  |
| <b>9) d</b>  | <b>10) a</b> | <b>11) d</b> | <b>12) d</b> |
| <b>13) d</b> | <b>14) a</b> | <b>15) b</b> | <b>16) c</b> |
| <b>17) a</b> | <b>18) b</b> | <b>19) b</b> | <b>20) c</b> |

### Short Questions/Answer

#### 5. Write three steps to convert source code into executable code.

**Ans:** C programs are written in human readable source code that is not directly executable by a computer. It takes a three step process to transform the source code into executable code. These three steps are: preprocessing, compiling and linking.

#### 6. What is the difference between define and declare a variable?

**Ans:** When a variable is defined, it is given a data type (int or whatever) and a name. Also ----this is the key point---memory is set aside for it , it is given physical existence. Defining variables is what we have done.

A variable can also be declared, which specifies the name and type of the variable, but does not set aside any memory. A declaration is simply an announcement that a variable with a particular name and type exists. A variable can be defined only once, but it can be declared many times.

### **3. Define conditional compilation.**

**Ans:** Conditional compilation is used during pre-processing stage to include / exclude code to be compiled depending upon your need. It's very useful in creating platform independent code where porting the code to another platform say from Unix to Windows or to different CPU architecture requires no changes to the source files.

### **4. What are preprocessor directives?**

**Ans:** The preprocessor will process directives that are inserted into the C source code. These directives allow additional actions to be taken on the C source code before it is compiled into object code. Directives are not part of the C language itself.

Preprocessor directives begin with a hash (#) symbol and may have several arguments.

### **5. Write the six types of Memory Models.**

**Ans:** These memory models are:

- (a) Tiny      (b) Small      (c) Medium
- (d) Compact    (e) Large      (f) Huge

### **6. What will be output of following C program?**

```
#include<stdio.h>
int main(){
 int *ptr;
 printf("%d",sizeof ptr);

 return 0;
}
```

**Ans: Output:** Depends upon memory model.

**Explanation:** If memory model is TINY, SMALL or MEDIUM then default pointer will be near and output will be 2 otherwise output will be 4.

## 7. What is Text Segment?

**Ans:** A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

## 8. what is #endif Directive.

**Ans:** In the C Programming Language, the #endif directive closes off the following directives: #if, #ifdef, or #ifndef. When the #endif directive is encountered, preprocessing of the opening directive (#if, #ifdef, or #ifndef) is completed.

### Syntax

The syntax for the #endif directive in the C language is:

**#endif**

## 9. What will be output of following C program?

```
#include<stdio.h>
int main(){
 char huge **p;
 printf("%d , %d",sizeof(p),sizeof(*p));
 return 0;
}
```

**Ans: Output:** 4, Depend upon memory model.

**Explanation:** p is huge pointer while type of pointer \*p will depend what is default type of pointer.

**10. write a function returns sum of squares of arguments.**

**Ans:** int formula (int x, int y)

```
{
 return (x*x + y*y) ;
}
```

**Long Questions**

6. Write a note on Stand-alone Executable.
7. What are Preprocessor Directives? Explain any three preprocessor directives with examples.
8. Explain Separate Compilation.
9. Explain the six Memory Models of C programming.
10. Explain, How the segment registers are used and the default size of pointers.

\*\*\*\*\*