**Yogi Vemana University::Kadapa**

**I Semester BSC – Computer Science**

**PROBLEM SOLVING IN C**

**UNIT I**

General Fundamentals: Introduction to computers: Block diagram of a computer, characteristics and limitations of computers, applications of computers, types of computers, computer generations.

Introduction to Algorithms and Programming Languages: Algorithm–Key features of Algorithms, Flow Charts, Programming Languages – Generations of Programming Languages – Structured Programming Language- Design and Implementation of Correct, Efficient and Maintainable Programs.

**UNIT II**

Introduction to C: Introduction–Structure of C Program–Writing the first C Program–File used in C Program – Compiling and Executing C Programs – Using Comments – Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C- Operators in C- Programming Examples.

Decision Control and Looping Statements: Introduction to Decision Control Statements–Conditional Branching Statements – Iterative Statements – Nested Loops – Break and Continue Statement – Goto Statement

**UNIT III**

Arrays: Introduction–Declaration of Arrays–Accessing elements of the Array–Storing Values in Array– Operations on Arrays – one dimensional, two dimensional and multi-dimensional arrays, character handling and strings.

**UNIT IV**

Functions: Introduction–using functions–Function declaration/ prototype–Function definition – function call – return statement – Passing parameters – Scope of variables – Storage Classes – Recursive functions.

Structure, Union, and Enumerated Data Types: Introduction–Nested Structures–Arrays of Structures – Structures and Functions– Union – Arrays of Unions Variables – Unions inside Structures – Enumerated Data Types.

**UNIT V**

Pointers: Understanding Computer Memory–Introduction to Pointers–declaring Pointer Variables – Pointer Expressions and Pointer Arithmetic – Null Pointers - Passing Arguments to Functions using Pointer – Pointer and Arrays – Memory Allocation in C Programs – Memory Usage – Dynamic Memory Allocation – Drawbacks of Pointers

Files: Introduction to Files–Using Files in C–Reading Data from Files–Writing Data to Files – Detecting the End-of-file – Error Handling during File Operations – Accepting Command Line Arguments.

**UNIT – I**

**Computer**

It is an electronic device which is used to perform both arithmetic and logical operations. Earlier the size of the computer is huge in size

**Characteristics of computer**:

1. **Speed:**

A Computer processes the data with high speed. It can perform millions of arithmetic and logical instructions per second. The speed of computers is usually given in nanoseconds and picoseconds where 1 nanosecond=1 x 10-9 seconds and 1 picoseconds = 1 x 10-12 seconds.

1. **Accuracy:**

Computer can perform each and every calculation very accurately. A computer is fast, reliable and robust electronic device.

1. **Automation:**

Besides being very fast and accurate, computers are automatable devices that can perform a task without any user intervention. The user just needs to assign the task to the computer, after which it automatically controls different devices attached to it and executes the program instructions.

1. **Diligence:**

Unlike human beings, a Computer is free from monotony, tiredness, lack of concentration etc. Hence it can work continuously without any errors and with the same accuracy.

1. **Storage:** A Computer can store and retrieve any amount of information.
2. **Versatility**: A Computer can be used for diversified functions.

**Limitations of Computer**:

1. **No IQ**:

Although the trend today is to make computer intelligent by inducing artificial intelligence in them, they still do not have any decision-making abilities of their own. Thus, their IQ level is zero. They need guidance to perform various tasks.

1. **No Feelings**

Lack of feeling is another limitation of computer. A computer cannot feel like us. It does not have emotions, feelings, knowledge, etc. It does not get tired and keep on doing its tasks. It can do very risky works which are not capable by human beings.

1. **Lack of common-sense**

This is one of the major limitations of computer systems. No matter how efficient, fast and reliable computer systems might be but yet do not have any common sense because no full-proof algorithm has been designed to programme logic into them. As computers function based on the stored programme(s), they simply lack common sense.

1. **Economical**

Today, computers are considered as short-term investments for achieving long-term gains. Using computers also reduces manpower requirements and leads to an elegant and efficient way of performing various tasks. Hence, computer saves time, energy and money

**Evolution of Computer**

The development of the computer is based on the idea of **calculator**. The first calculating device is **ABACUS** which was invented by Chinese in 2000 years ago. It is a wooden frame in which beads are threaded on wires. It is still used in the nursery schools for teaching basic arithmetic operations.

**In 1642, BLAISE PASCAL** a French mathematician developed a calculating machine called **PASCALINE**. The machine was also used for addition and subtraction purpose. The device was operated by dialing a set of wheels. **In 1671 LEIBNIZ** improved on Pascal's adding machine and invented the LEIBNIZ's Calculator.

**In 1823 Charles Babbage** an Englishman invented a **Difference Engine** to calculate algebraic expressions. The purpose of this device was to calculate the roots of polynomial equations and prepare astronomy table for the British Navy. He upgraded this in 1833 to, invent an **Analytical engine**, which could store program instructions initially coded on punched cards and subsequently shared internally. Therefore Charles Babbage is known as the father of computers.

**Dr. HERMANN HOLLERITH**, an American made an improved type of mechanical computer in **1890** which works with punched cards. This equipment read the holes punched in the card and mechanically performed the statistical analysis.

The first pure electronic computer was invented by **J. V. ATANASOFF** and **C. BERRY** which is known as **ATANASOFF-BERRY COMPUTER** or **ABC**. It used vacuum tubes for both data storage and data computation. Subsequently the first electronic calculator called **ENIAC** (Electronic Numerical Integrator and calculator) was designed by **Dr. JOHN MAUCHLY** and **J.P ECKERT** in 1945 and it is accepted as the general purpose computer.

In 1948 **JOHN VON NEUMANN** developed a computer called EDVAC (Electronic Discrete Variable Automatic Computer) in which the first stored program concept used. The next important development was the UNIVAC (UNIVERSAL AUTOMATIC COMPUTER) developed in 1951 and this is considered as the first generation computer.

**Applications of Computers**

When the first computers were developed, they were used only in the fields of mathematics and science. In fact, the first effective utilization of computers was for decoding messages in military applications.

Computers are made use of in almost all fields, ranging from ***business, science, engineering, banking, medicine, transportation, education, entertainment*** and ***many more***. Some of the applications are

1. **CAD**
2. Computer-Aided Design is a phrase that refers to the integration of computers and graphics design packages for the purpose of aiding in the design and drafting process.
3. The equipment found in a CAD installation includes a computer and a variety of input devices and output units.
4. An engineer can see on the computer screen a 3-dimensional picture of a machine analyze its characteristics, and then subject it to simulated stresses. If the machine fails a test, its specifications are changed on the screen and it is re-tested.
5. **CAM**
6. The use of computers in the production process is called Computer Aided Manufacturing.
7. CAM is a way of manufacturing higher-quality products with more uniform tolerance than were previously possible.
8. **Electronic Fund Transfer System:**
9. **Electronic Funds Transfer** is a term that identifies any system used to transfer payments or funds electronically.
10. EFT refers to any transfer of funds, other than a transaction originated by check, draft or similar paper instrument, that is initiated through an electronic terminal, telecommunications instrument, computer or magnetic tape, to order, instructs or authorize a financial institution to debit or credit an account.
11. At present, people pay for goods purchased and services received, by credit cards instead of paper money.
12. The increased use of these cards has promoted financial institutions to print some codes that can be sensed by special sensing devices and converted to respective electrical signals. These electrical signals can be set between computers and on-line financial terminals to adjust the accounts of the parties involved in a transaction.
13. No cash or cheques are used in such an Electronic Funds Transfer System
14. The ATM is one such electronic fund transfer station allowing people to deposit or withdraw cash from their bank accounts at any time of the day or night by inserting the ATM card into the ATM machine
15. **Electronic Mail System**
16. Electronic Mail System is a way of communicating with each other electronically.
17. A spoken or keyed message is sent to one individual or a group of individuals who are connected to the mail-system, at any time of the day or night.
18. The receiver can periodically link up with the system and receive stored messages at a time that is convenient. This helps in connecting remote locations and also saves time.
19. **Digital video or audio composition:**
20. Computers make audio or video composition and editing very simple
21. This has drastically reduced the cost of equipment to compose music or make a film.
22. Graphics engineers use computers for developing short or full-length films and creating 3D models and special effects in science fiction and action.
23. **Desktop Publishing:**

It is a software system that combines a computer and suitable peripherals (laser printer, digitizer, etc.,) to produce attractive page layouts complete with pictures and text printed in a variety of typefaces or fonts. Ideal for use in newspaper and magazine publishing companies

1. **Robotics:**
2. Robotics is a general term that refers to the study of robot technology.
3. A robot is a computer controlled manipulator that is designed to move materials, parts, tools or any other specialized devices through a series of programmed steps.
4. Used extensively in the manufacturing industry.

**GENERATIONS OF COMPUTERS**

**Ans:**

Computers developed after ENIAC have been classified into the following five generations. **The generations of computers are classified into 5 types** based on the **size of computer and speed of computer**.

1. **FIRST GENERATION COMPUTERS** (1940-1956):
   * The first generation computers used **vacuum tubes** to store and process information.
   * The vacuum tubes consumed lot of power and generated too much heat.
   * They required cooling systems to keep the temperature low.
   * The first generation computers were huge in size and had limited memory and processing power.
   * They were slow, expensive and often undependable. The first generation computers are Very big in size, weight and occupied large space.
   * During this period, computer programming was mainly done in machine language.
   * UNIVAC AND ENIAC are prime examples of first generation computing devices. Moreover, UNIVAC was the first commercial computer delivered to a business client(US Census Bureau in 1951)

Ex: UNIVAC, IBM-701

**Advantages:**

1. They were the fastest calculating devices of their time.

**Disadvantages:**

1. They generates a lot of heat
2. They consumed a lot of electricity
3. They were very bulky in size.
4. They were expensive
5. These computers had limited commercial use because they were difficult to program.
6. **SECOND GENERATION COMPUTERS** (1956-1963)

* In 1948 three scientists, **John Bardeen**, **William Shockley** and **Walter Brattain**, invented transistor at the AT&T‘s Bell Labs. This invention started a big revolution in electronics. These transistors replaced vacuum tubes in the second generation computers.
* The transistors did the same function of vacuum tubes more efficiently.
* Each transistor was equivalent to 40 vacuum tubes.
* It requires less power and generates less heat than vacuum tubes. The size of second generation computer is reduced and speed increased.

Ex: UNIVAC-1108, MARK-III

**Advantages:**

1. They consumed less electricity and thus generates less heat as compared to the first generation computers.
2. They were faster, cheaper, smaller and more reliable than their first-generation computers.
3. They could be programmed using assembly language and high-level languages.

**Disadvantages:**

1. Second generation computers were manufactured using transistors, which had to be assembled manually. This made commercial production of computers difficult and expensive.
2. **THIRD GENERATION COMPUTERS** (1965-1970)

* The development of the IC was the hallmark of the third-generation computers. Several electronic components such as transistors, resistors and capacitors were miniaturized and placed on silicon chips, called integrated chips, which drastically increased the speed and efficiency of computers.
* Integrated chips were smaller, less expensive, more reliable and faster in operation, consumed less power, and generated less heat than the components used earlier.
* Introduction of mini computers.
* Introduction of programming languages.

Ex: PDP-ii, ICL-2903

**Advantages:**

1. They were faster than second generation computers and could perform one million transactions per second.
2. They were smaller, cheaper and more reliable than their predecessors
3. The computers had faster and larger primary memory and secondary storage.
4. These computers were widely used for scientific as well as business applications.
5. In this generation of computers, standardization of existing high-level languages and invention of new high level languages happened.

**Disadvantages:**

1. These computers were difficult to maintain
2. They got heated very quickly
3. **FOURTH GENERATION COMPUTERS** (1971-1985)

* Replacement of LSICs by VLSI (Very large scale Integrated) Chips.
* Introductions to micro computers (PCs)
* Requires Low Power.
* Very high speed in operation.
* Works more efficiently than third generation computers.

Ex: HCL, Apple –II

**Advantages:**

1. These computers were smaller, cheaper, faster and more reliable than their predecessors
2. They consumed less electricity and generated less heat
3. They had faster and larger primary memory and secondary storage.
4. They could be used as general-purpose computers
5. Networks allowed sharing of resources, thereby enabling efficient utilization of computer hardware and software.

**Disadvantages:**

1. They were not intelligent systems
2. **FIFTH GENERATION COMPUTERS** (1985 onwards)

* These generation computers are completely based on the new concept of **Artificial Intelligence**. Although such computers are still in development, there are certain applications such as voice recognition that are widely being used today.
* Parallel processing and superconductor technology have made AI a reality.
* Quantum computation and nanotechnology will radically change the face of computers in the year to come.
* In the fifth generation computers, the aim is to develop devices that respond to natural language input and are capable of learning and self-organization.
* Large memories.
* Acceptance of spoken and written commands.
* Super high in operations.
* Introduction to super computers.

**TYPES OF COMPUTERS**

Computers vary widely in terms of their size and purpose they serve. Computers can be classified on several bases as follows.

1. **On the basis of electronics or the operating principle, they can be classified into:**
   1. Analog Computers
   2. Digital Computer
   3. Hybrid computer
2. **On the basis of size, Digital computers are classified into**
   1. Portable computer
   2. Desktop computer
   3. Minicomputer
   4. Mainframe computer
   5. Super computer
3. **On the basis of purpose**
   1. Special Purpose computer
   2. General Purpose computer
4. **On the basis of electronics or the operating principle, they can be classified into:**
   * 1. **ANALOG COMPUTER**:
5. Analog computer operates on inputs of continuously varying electrical voltage.
6. It measures the input rather than counting.
7. The name that is derived from the Greek word ‘analog’ denotes that the computer functions by establishing similarities between two quantities that are usually expressed as voltages or currents.
8. Analog computers are powerful tools to solve differential equations.
9. They are mainly used to be constructed to do a specific job and will respond very quickly to changes in the measured inputs.
   * 1. **DIGITAL COMPUTER**:
10. A digital computer operates essentially by counting. All quantities are expressed as discrete digits or numbers and computations are done with numerical digits.
11. Mathematical expressions are represented as binary digits and all operations are done using these binary digits at a very high rate.
12. The digital basically knows addition. It converts other operations into addition then Calculate.
13. It is much faster than analog computer.
    * 1. **HYBRID COMPUTER**:
14. Hybrid computer combines features of both analog and digital computers.
15. In this computer some calculations are done in the analog portion of the computer and some are done in the digital portion of it.
16. **On the basis of size, Digital computers are classified into**
17. **PORTABLE COMPUTER:**

Portable computer is very small, easy to use microcomputer. The users can carry it wherever they go. Business executives, traveling salesman etc, carry it during travel for personal use. This group includes pen based computer, hand held computer, note book computer and laptop computer.

1. **DESKTOP COMPUTER:**

The desktop computer is a small general-purpose microcomputer, but larger than portable computers. It is normally installed on a desktop and hence the name desktop computer. It is a self-contained system, usually designed for use by one person at a time. Since the desktop computers can be easily linked to large computers.

1. **MINI COMPUTER:**

It is a medium sized computer that is costlier and more powerful than a microcomputer. This can support several users at a time, with multi-terminal, time-sharing system. Minis are the popular data processing systems among the business organizations today. They are mainly used as departmental computers in large and medium-sized organizations. They are also used in government departments, universities.

1. **MAINFRAME COMPUTERS:**

The earliest computers were called mainframes due to their size. The term is still used for the large computers today. They have large storage capacities, very high speed of processing, very high speed of processing and can support a large number of terminals for use by a variety of users simultaneously. They are kept in air-conditioned environment in a special computer room. They are used by big companies, banks, government departments etc as their main computer.

1. **SUPER COMPUTER:**

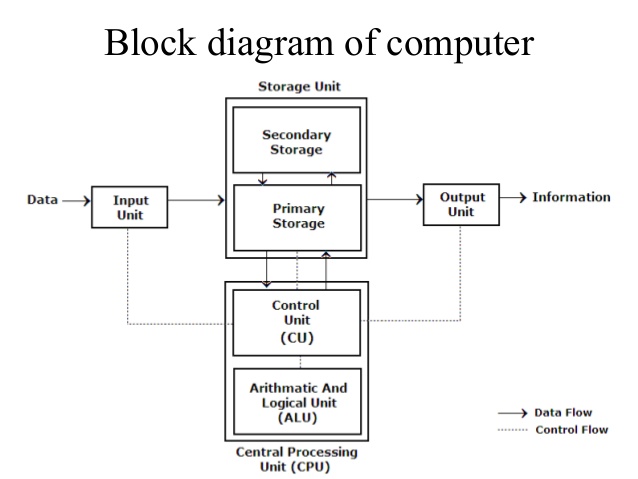
This has extremely large storage capacities and computing speeds that are at least ten times faster than that of other computers. They super computer is used for large-scale numerical problems in scientific and engineering disciplines. These include applications in electronics, petroleum engineering and weather forecasting. This has traditionally been used in scientific and military organizations.

1. **On the basis of purpose**
2. **Special Purpose computers**

These are computers designed to carry out specific tasks. They have in-built programs which are stored in a part of the main memory called ROM. The content of this part of the memory can be accessed and executed by the computer, but cannot be modified by the programmer or the user. Thus, the operations that can be carried out by this type of computer are pre-determined at the time of manufacture. The computer cannot be used for any other purpose.

1. **General Purpose Computer**

These are computers that are not specifically designed or built for specific jobs. They solve various kinds of problems depending on the program or software loaded into them. Their main memory is typically Random Access Memory (RAM) - a temporary storage that looses its contents when the computer is switched off. It is easy to change the contents of RAM, substituting one program for another and this is what makes them general-purpose computers.

**Block diagram of digital computer:**

**INPUT UNIT:**

It is used to send data or programs in to the computer system. The following functions are performed by input unit.

* It accepts the list of instructions and data from the outside world.
* It converts these instructions and data in computer acceptable form.
* It supplies the converted instructions and data to the computer system for further processing.

There are various kinds of input devices available. They are keyboard, mouse, Light pen, Joystick, Scanners, Touch Screen, OMR, etc.

**CPU:**

It stands for Central Processing Unit. The CPU is the brain / heart of any computer system. The CPU is also responsible for activating and controlling the operations of their units of a computer system. It is made of three units .They are –

1. Memory Unit
2. Arithmetic Logical Unit
3. Control Unit

**MEMORY UNIT:**

The function of the memory is to store data or instructions or information in the form of ***Zero/off (0)*** or ***one/on (1)*** states.0 or 1 is called as BITS (binary digits). The memory unit is of 2 types. They are –

* + - 1. **RAM:**
* It stands for ***Random Access Memory***.
* Here data can be stored temporarily, so this type of memory is called as a **temporary memory** or **volatile memory** because when power fails the data from the RAM will be erased.
* The information stored in the RAM is basically loaded from the computer’s hard disk and includes data related to the operating system and applications that are currently being executed by the processor.
* RAM is considered random access because any memory cell can be directly accessed if its address is known. RAM is of different types, such **SRAM, DRAM AND VRAM**.
  + - 1. **ROM:**
* It stands for Read Only Memory.
* In this the data will be provided by the manufacturers regarding the system, so this information can simply be read by the user but cannot add new data or it cannot be modified.
* ROMs are of different types
* **PROM**

A programmable read-only memory (**PROM**) is a form of digital memory where the setting of each bit is locked by a fuse or antifuse. It is one type of read-only memory (ROM). The data in them are permanent and cannot be changed.

* **EPROM**

Erasable Programmable Read Only Memoryis fullerasable programmable read-only memory form of **computer** memory that does not lose its content when the power supply is cut off and that can be erased and reused.

* **EEPROM**

**EEPROM** stands for **electrically erasable programmable read-only memory** and is a type of [non-volatile memory](https://en.wikipedia.org/wiki/Non-volatile_memory) used in computers, integrated in [microcontrollers](https://en.wikipedia.org/wiki/Microcontrollers) for [smart cards](https://en.wikipedia.org/wiki/Smart_card) and [remote keyless systems](https://en.wikipedia.org/wiki/Remote_keyless_system), and other electronic devices to store relatively small amounts of data by allowing individual bytes to be erased and reprogrammed.

**CONTROL UNIT (CU):**

* It stands for Control unit.
* It manages and coordinates the entire computer system.
* It obtains instructions from the program stored in main memory, interprets the instructions, and issues signals that cause other units of the system to execute them.

**ARITHMETIC LOGICAL UNIT (ALU):**

* It stands for Arithmetic and Logic Unit.
* It performs the arithmetic and logical operations.
* Arithmetic operations mean addition, subtraction, multiplication, division etc., whereas the logical operation means comparisons and decision-making.

**OUTPUT UNIT:**

The job of an output unit is just the reverse of that of an input unit. As computers work with binary code, the results produced are also in the binary form. Hence before supplying the results to the outside world it must be converted to human acceptable form. This task is accomplished by the output unit.

**Algorithm**

Computer-based problem solving should proceed systematically. There should be a step-by-step progression towards the solution. **An algorithm is a step-by-step procedure to solve a given problem**. Thus algorithm is a sequence of precise and unambiguous instructions for solving a problem in a finite number of operations. **The word algorithm is derived from the name of a 9th century Arab mathematician MOHAMMED IBN-MUSA AL-KHWARIZMI** who developed methods for solving problems which used specific step-by-step instructions.

**Features**

1. **Precision** – The steps are precisely stated
2. **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
3. **Finiteness** – The algorithm stops after a finite number of instructions are executed.
4. **Input** – The algorithm receives input
5. **Output** – The algorithm produces output.
6. **Generality** – The algorithm applies to a set of inputs.
7. It should be clear with no ambiguity
8. It should lead to a unique solution of the problem
9. It should have the capability to handle some unexpected situations which may arise during the solution of a problem

**Advantages**

1. It is a step by step solution to a given problem which is very easy to understand.
2. It has got a definite procedure which can be executed within a set of period of time
3. It is easy to first develop an algorithm, then convert it into a flowchart and then into a computer program.
4. It is independent of programming language.
5. It is easy to debut as every step has got its own logical sequence.

**Disadvantages**

1. It is time consuming and Cumbersome as an algorithm is developed first which is converted into a flowchart and then into a computer program.

**Control Structures used in Algorithm:**

An algorithm has a finite number of steps and some steps may involve decision-making and repetition. **An algorithm uses three control structures, namely sequence, decision and repetition**.

1. **SEQUENCE:**

Sequence means that each step of the algorithm is executed in the specified order.

1. **DECISION:**

Decision statements are used when the execution of a process depends on the outcome of some condition.

1. **REPETITION:**

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as the **while, do-while and for loops.** These loops execute one or more steps until some condition is true.

**Flowchart:**

* **A flowchart is a visual or graphical representation of an algorithm.** It indicates the process of solution, the relevant operations and computations the point of decisions and other information as part of the solution.
* **Flowcharts are constructed by using special geometrical symbols.Each symbol represents an activity.The activity could be*input / output of data***, ***computation / processing of data***, ***taking a decision or terminating the solution***, etc. The symbols are joined by arrows to obtain a complete flowchart.
* The geometrical symbols used to design flowcharts are

|  |  |
| --- | --- |
| **Symbols** | **Description** |
|  | Terminal (Start , Stop) |
|  | Input / Output |
|  | Process / Computation |
|  | Decision Making |
|  | Sub-routine |
|  | Connector |
|  | Flow lines |

**Significance of Flowchart:**

1. A flowchart is a diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem.
2. It is usually drawn in the early stages of formulating computer solutions.
3. It facilitates communication between programmers and users. Once a flowchart is drawn, programmers can make users understand the solution easily and clearly.
4. Flowcharts are very important in the programming of a problem as they help the programmers to understand the logic of complicated and lengthy problems. Once a flowchart is drawn, it becomes easy for the programmers to write the program in any high-level language. Hence, the flowchart has become a necessity for better documentation of complex programs.
5. A flowchart follows the top-down approach in solving problems

**Advantages of Flowchart**

1. Flowcharts are very good communication tools to explain the logic of a system to all concerned. They help to analyze the problem in a more efficient manner.
2. They are also used for program documentation. They are even more helpful in the case of complex programs.
3. They act as a guide or blueprint for the programmers to code the solutions in any programming language. They direct the programmers to go from the starting point of the program to the ending point without missing any step in between. This results in error-free programs.

**Limitations of Flowchart:**

1. Drawing flowcharts is a difficult and time-consuming activity.
2. Often, the flowchart of a complex program becomes complex and clumsy.
3. At times, a little bit of alteration in the solution may require complete re-drawing of the flowchart
4. There is no well-defined standards that limit the details that must be incorporated into a flowchart.

**Examples**

|  |  |
| --- | --- |
| **1Q) Write an algorithm and flowchart to calculate sum of two values** | |
| **ALGORITHM** | **FLOWCHART** |
| **Step 1:** Start  **Step 2:** set a=100,b=75  **Step 3:** compute c= a+b  **Step 4:** print c  **Step 5:** Stop |  |

|  |  |
| --- | --- |
| **2Q) Write an algorithm and flowchart to calculate sum of n natural numbers** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  **Step 2:** read n  **Step 3:** compute s= n\*(n+1)/2  **Step 4:** print s  **Step 5:** Stop |  |

|  |  |
| --- | --- |
| **3Q) Write an algorithm and flowchart to swap the two given values** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  **Step 2:** read a, b  **Step 3:** compute a=a + b  **Step 4:** compute a=a - b  **Step 5:** compute b=a - b  **Step 6:** print a, b  **Step 5:** Stop |  |

|  |  |
| --- | --- |
| **4Q) Write an algorithm and flowchart to find the highest among them** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  **Step 2:** read a, b  **Step 3:** if (a > b) then  **Step 4:** print “ A is Big”  otherwise  **Step 5:** Print “ B is Big”  **Step 6:** Stop |  |

|  |  |
| --- | --- |
| **5Q) Write an algorithm and flowchart to find whether the given number is even or odd.** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  **Step 2:** read n  **Step 3:** if (n%2=0) then  **Step 4:** print “Even No.”  otherwise  **Step 5:** Print “Odd No.”  **Step 6:** Stop |  |

|  |  |  |
| --- | --- | --- |
| **6Q) Write an algorithm to find the weekday based on the weekday number** | | |
| **Algorithm** | **Flowchart** | |
| **Step 1:** Start  **Step 2:** read wd  **Step 3:** if (wd=0) then  **Step 4:** print “Sunday”  otherwise  **Step 5:** if(wd=1) then  **Step 6:** Print “Monday”  otherwise  **Step 7:** if(wd=2) then  **Step 8:** Print “Tuesday”  otherwise  **Step 9:** if(wd=3) then  **Step 10:** Print “Wednesday”  otherwise  **Step 11:** if(wd=4) then  **Step 12:** Print “Thursday”  otherwise  **Step 13:** if(wd=5) then  **Step 14:** Print “Friday”  otherwise  **Step 15:** if(wd=6) then  **Step 16:** Print “Saturday”  Otherwise  **Step 17:** print “Invalid”  **Step 18:** Stop | Read wd  If(wd=0) then  sunday  Monday  Tuesday  wed  Thursday  Friday  If(wd=1) then  If(wd=2) then  If(wd=3) then  If(wd=4) then  If(wd=5) then  Saturday  If(wd=5) then  Invalid | |
| **7Q) Write an algorithm to accept rno,name and 3 subject marks. Calculate total, average and issue grade as per conditions**  **3 subject marks avg grade**  **>=35 >=75 Distinction**  **>=35 >=60 && avg< 75 First**  **>=35 >=50 &&<60 Second**  **>=35 >=35 &&<50 Third**  **- - Fail** | | |
| **Algorithm** | | **Flowchart** |
| **Step 1:** Start  **Step 2:** read rno,nm,m1,m2,m3  **Step 3:** compute tot=m1+m2+m3  **Step 4:** compute avg=tot/3  **Step 5:** print tot, avg  **Step 6:** if (m1<35 or m2<35 or m3<35) then  **Step 7:** print “Grade = Fail”  otherwise  **Step 8:** if(avg>=75) then  Step **9**: Print “Grade=Distinction”  otherwise  **Step 10:** if(avg>=60 and avg<752) then  Step **11:** Print “First”  otherwise  **Step 12:** if(avg>=50 and avg<60) then  Step **13:** Print “Second”  otherwise  **Step 14:** if(avg>=35 and avg<50) then  **Step15:** Print “Third”  **Step 16:** Stop | |  |

|  |  |
| --- | --- |
| **8Q) Write an algorithm and flowchart to display squares of numbers from 0 to 9** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  **Step 2:** read n  **Step 3:** for i=1 to n step 1  begin  **Step 4:** print i  End  **Step 5:** Stop | cforloopflowchart067.png (389×313) |

|  |  |
| --- | --- |
| **9Q) Write an algorithm to display ur name from 5 times** | |
| **Algorithm** | **Flowchart** |
| **Step 1:** Start  Step 2: Read n  Step 3: Set Fact=1  **Step 4:** for i=1 to n step 1  begin  **Step 5:** compute fact = fact \* i  End  **Step 6:** Print fact  **Step 7:** Stop |  |

**Programming Languages**

A programming language is a language specifically designed to express computations that can be performed by a computer. Programming languages are used to create programs that control the behavior of a system, to express algorithms, or as a mode of human communication.

The term programming language refers to high level language such as BASIC (Beginners All-purpose symbolic instruction code), C, C++, COBOL (Common Business Oriented Language), FORTRAN, ADA and PASCAL. Each of these languages has a unique set of keywords and a special syntax for organizing program instructions.

Though high-level programming languages are easy for humans to read and understand, the computer can understand only machine language, which consists of only numbers. Each type of CPU has its own unique machine language.

In between machine languages and high-level languages, there is another type of language known as assembly language. Assembly languages are similar to machine languages, but they are much easier to program because they allow a programmer to substitute names for numbers.

**Generations of Programming Languages**

The concept of generations of programming languages is closely connected to the advances in technology. **The five generations of programming languages include*machine language***, ***assembly language,high-level language*** (also known as 3GL), ***very high level language*** (also known as 4GL) and the ***fifth generation language that includes artificial intelligence***.

1. **Machine Language**

* All the instructions are written in machine level language are in the form of 0’s and 1’s.
* It is very difficult to learn and difficult to understand. This is also called as the low level language.
* Machine language was used to program the first stored program computer system. This is the lowest level of programming language and is the only language that a computer understands. All the commands and data values are expressed using 0s and 1s, corresponding to the off and on electrical states in a computer.
* In 1950s, each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as add and subtract. Although there were similarities between each of the machine languages, a computer could not understand programs written in another machine language.
* The main advantage of machine language is that the execution of the code is very fast and efficient since it is directly executed by the CPU. However, on the downside, machine language is difficult to learn and is far more difficult to edit if errors occur.

**Advantages:**

1. Machine languages make efficient use of storage.
2. Instructions of a machine language program are immediately executable.
3. Program execution is faster
4. The computer can understood instructions directly and therefore no translation is needed.

**Disadvantages:**

1. Machine language is machine dependent
2. Code is difficult to write.
3. Code is difficult to understand by other people
4. Code is difficult to maintain.
5. It is difficult to detect and correct errors.
6. **ASSEMBLY LANGUAGE**

* All the instructions are written in Assembly level languages are in the form of **mnemonics**. **Mnemonic means symbolic name.** ADD for addition, MUL for multiplication, SUB for subtraction, DIV for Division etc.
* Assembly language instructions computer cannot understand directly. So, there is a need of translator (assembler). **Assembler is a system, which translate assembly level instructions into machine understandable form.**
* These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since it is close to machine language, assembly language is also a low-level language. Nearly all computer systems have an assembly language available for use.
* Assembly language developed in the mid-1950s. It used symbolic codes, also known as mnemonic codes, which are easy-to-remember abbreviations, rather than numbers.
* Assembly language programs consist of a series of individual statements or instructions to instruct the computer what to do. Basically, an assembly language statement consists of a label, an operation code and one or more operands.
* A symbolic program written by a programmer in assembly language is called a source program. After the source program has been converted into machine language by an assembler, it is referred to as an object program.

**ADVANTAGES**

1. It is easier to write programs in assembly language than in machine language.
2. It is not required to keep track of memory locations.
3. It is easy to detect and correct errors
4. Assembly language programs are easier for people to modify than machine language programs

**DISADVANTAGES**

1. Programs are machine dependent and thus non-portable.
2. Programmers must have a good knowledge of the hardware and internal architecture of the CPU.
3. The code cannot be directly executed by the computer
4. **HIGH LEVEL LANAUGES**

* All the instructions are written in High Level languages are in the form of general English. These languages are very easy to learn and easy to implement. But it also requires the translator to translate these programs into machine understandable form.
* To overcome the low level language difficulty of machine dependency, high level problem-oriented languages were developed. Such programming languages, with an extensive vocabulary of words and symbols are used to instruct a computer to carry out the necessary procedures, regardless of the type of machine being used.
* Interactive languages provide facilities for the programmer to make corrections and for changes to his program during its compilation and execution. The increasing availability of mini computers and micro-computers has speeded up this process of developing languages that enable the user to obtain maximum use of the computer, without undergoing an extended period of special training or incurring the considerable post of employing a computer programmer.

**Advantages:**

1. The code is machine independent
2. It is easy to learn and use the language
3. There are few errors
4. It is easy to maintain the code.
5. It is easy to detect and correct errors

**Disadvantages:**

1. Code may not be optimized
2. The code is less efficient
3. It is difficult to write a code that controls the CPU, memory and registers.
4. **VERY HIGH LEVEL LANGUAGES (4GLs):**

With each generation, programming languages started becoming easier to use and more similar to natural languages. 4GLs are a little different from their prior generation because they are non-procedural. While writing a code using a procedural language, the programmer has to tell the computer how a task is done – add this, compare that, do this if the condition is true, and so on- in a specific step-by-step manner.

**Characteristics of 4GLs:**

1. The instructions of the code are written in English like sentences
2. They are non-procedural, so users concentrate on the ‘what’ instead of the ‘how’ aspect of the task.
3. The code written in a 4GL is easy to maintain.
4. The code written in a 4GL enhances the productivity of programmers, as they have to type fewer lines of code to get something done.

A typical example of a 4GL is the query language, which allows a user to request information from a database with precisely worded English-like sentences. A query language is used as a database user interface and hides the specific details of the database from the user.

1. **FIFTH GENERATION PROGRAMMING LANGUAGE(5GL):**

Fifth-generation programming languages are centred on solving problems using the constraints given to a program rather than using an algorithm written by a programmer. Most constraint based and logic programming languages and some declarative languages form a part of the 5GLs. **These languages are widely used in artificial intelligence research**. Another aspect of a 5GL is that it contains visual tools to help develop a program.

**Eg:** Prolog, OPS5, Mercury and VB.

5GLs are designed to make the computer solve a given problem without the programmer. While working with a 4GL, programmers have to write a specific code to do a work, but with a 5GL, they only have to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or an algorithm to solve them.

In general, 5GLs were generally built upon LISP, many originating on the LISP machine, such as ICAD. There are also many frame languages, such as KL-ONE.

**Structured Programming:**

The concept of structured programming, also referred to as modular programming, was first suggested by the mathematicians **CORRADO BOHM** and **GIUSEPPE JACOPINI**. It is basically a subset of procedural programming that enforces a logical structure on the program to make it efficient and easy to understand and modify.

Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules. This allows the code to be efficiently loaded into the memory and to be reused in other programs. Modules are coded separately, and once a module is written and tested individually, it is then integrated with the other modules to form the overall program structure.

Large programs are difficult to understand. Hence complex programs are written using three logic structures: **sequence, selection and iteration**.

Structured programming employs standard tools and program constructs. The structured programming contains three standard control structures

1. Sequence
2. Selection
3. Iteration

All of the programmers use standard methods for coding. Coding and debugging are facilitated as the program is properly segmented and errors can be localized. Each module, which performs a given task after debugging, is integrated with other modules to form a full program. Programming work is faster and programmer productivity is higher than unstructured programming. Structured programming thus employs modular program design.

**Advantages of Structured Programming:**

1. Takes less time for programming
2. Program is divided into functional modules: each module performs one simple function and is easy to understand.
3. Each module can be coded and debugged independently
4. Programming work can be shared by a number of programmers, each working independently of others
5. Errors can be localized as the errors in one module can be located and corrected before being integrated with other modules.

**Disadvantages of Structured Programming:**

1. Code duplications and subroutines slowdown execution resulting in lower execution efficiency.
2. Several layers of modules in hierarchy can complicate logic and debugging.

**DESIGN AND IMPLEMENTATIONS OF CORRECT, EFFICIENT AND MAINTAINABLE PROGRAMS:**

The design and development of correct, efficient, and maintainable programs depends on the approach adopted by the programmer to perform the various activities that need to be carried out during the development process. The entire program or software development process is divided into a number of phases where each phase performs a well-defined task. Moreover, the output of one phase provides the input for its subsequent phase. The phases in a software development process can be summarized as follows.

1. **Requirement Analysis:**

In this phase, the user’s expectations are gathered to understand why the program or software has to be developed. Then, all the gathered requirements are analyzed and the scope or objective of the overall software product is penned down. The last activity in this phase involves documenting every identified requirement of the user in order to avoid any doubts or uncertainty regarding the functionality of the program. The functionality, capability, performance and availability of hardware and software companies are all analyzed in this phase.

1. **Design:**

The requirements documented in the previous phase act as the input to the design phase. In this phase, a plan of actions is made before the actual development process starts. This plan will be followed throughout the development process. Moreover, in the design phase, the core structure of the software or program is broken down into modules. The solution of the program is then specified for each module in the form of algorithms or flow charts. The design phase, therefore, specifies how the program or software will be developed.

1. **Implementation**

In this phase, the designed algorithms are converted into program code using any of the high-level languages. The particular choice of language will depend on the type of program such as whether it is a system or an application program. C is preferred for writing system programs whereas VB might be preferred for an application program. The program codes are tested by the programmer to ensure their correctness.

The implementation phase is also called the construction or code generation phase as the code of the software is generated in this phase. While constructing the code, the development team checks whether the software is compatible with the available hardware and other software components mentioned in the requirements specification document created in the first phase.

1. **Testing**

In this phase, all the modules are tested together to ensure that the overall system works well as a whole product. Although individual pieces of codes have already been tested by the programmers in the implementation phase, there is always a chance for bugs to creep in the program when the individual modules are integrated to form the overall program structure. In this phase, the software is tested using a large number of varied inputs, also known as test data, to ensure that the software is working as expected by the user’s requirements identified in the requirements analysis phase.

**Software deployment, training and support** after the code is tested and the software or the program is approved by the users, it is then installed or deployed in the production environment.

Software training and support is a crucial phase but it is often ignored by most of the developers. Program designers and developers and spend a lot of time creating the software, but if nobody in the organization knows how to use it or to fix certain problems, then no one will want to use it. Moreover, people are often resistant to change and avoid venturing into an unfamiliar area. Hence, as part of the deployment phase, it has become very crucial to provide training for the users of the software.

1. **Maintenance:**

Maintenance and enhancements are ongoing activities that are done to cope with newly discovered problems or new requirements. Such activities may take a long time to complete, as the requirement may call for addition of a new code that does not fit the original design or an extra piece of code may be required to fix an unforeseen problem. In general, if the cost of the maintenance phase exceeds 25 per cent of the cost of the prior phases, then it means that the overall quantity of at least one prior phase its poor. In such cases, it is better to rebuild the software before the maintenance cost goes out of control.

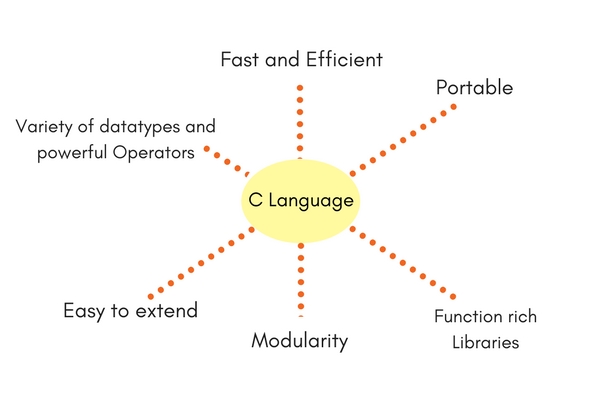
**UNIT-II**

**Introduction of C-language**

* The C programming language is a **structure oriented programming language**, developed at Bell Laboratories in **1972 by Dennis Ritchie**
* C programming language features were derived from an earlier language called **“B”** (Basic Combined Programming Language – BCPL)
* C language was invented for implementing UNIX operating system
* **In 1978, Dennis Ritchie and Brian Kernighan** published the first edition “The C Programming Language” and commonly known as K&R C
* **In 1983, the American National Standards Institute (ANSI)** established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or “ANSI C”, was completed late 1988.

**Features of C-language**

1. It is **structured oriented programming Language**.
2. It is highly **portable** this means that programs once written can be run on another machines with little or no modification.
3. It is a **robust language** with rich set of built-in functions and operators that can be used to write any complex program.
4. The C compiler combines the capabilities of an assembly language with features of a high-level language.
5. Programs Written in C are **efficient and fast**. This is due to its variety of data type and powerful operators.
6. It is many time faster than BASIC.
7. A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.
8. C language is the most widely used language in operating systems and embedded system development today.
9. It is used to perform any kind of calculations.
10. It is used to control hardware components of a system.
11. It has a wide variety of derived data structures like pointers, arrays structures and unions apart from fundamental data types like integers, floating point numbers and characters.
12. It has the ability to deal efficiently with bits, bytes, word, addresses etc.,



**C-Tokens**

A punctuation marks, commas, semi-colons, characters etc., is called as a C-Token. C-Tokens are of 6 types. They are –

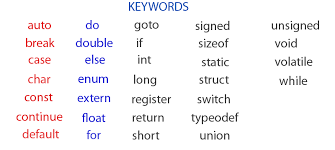
1. Identifiers
2. Keywords
3. Operators
4. Datatypes
5. Strings
6. Special Symbols
7. **IDENTIFIERS**

An identifier is a function name or variable name. To define identifiers the following rules to be followed. They are –

1. The first character of identifier name should always begin with alphabet or underscore.
2. In between identifier name there should not be any special symbol except underscore ("\_").
3. Keywords should not be defined as Identifier.
4. Duplicate identifiers cannot be defined.
5. **KEYWORDS**

The words which are already pre-defined in a c-language is said to be keywords. These are also called as **reserve words**. C-language contains **32 keywords.**

**eg:-**



1. **OPERATORS**

An operator is nothing but a symbol which, is used to operate the operands. The operators are

1. Arithmetic operators
2. Relational / Comparison operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Logical Operators
6. Unary operators
7. Ternary Operators

**Arithmetic Operators**

The arithmetic operators are used to perform arithmetic calculations such as addition, subtraction, multiplication and division. The arithmetic operators are

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Examples** |
| + | Addition | A=10,B=20,C=A+B=>30 |
| - | Subtraction | A=10,B=20,C=A-B=>-10 |
| \* | Multiplication | A=10,B=20,C=A\*B=>200 |
| / | Division | A=10,B=2,C=A/B=>5 |
| % | Modulus | A=10,B=3,C=A%B=>1 |

**Relational / Comparison Operators**

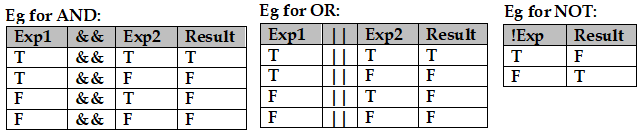
The relational or comparison operators are used to compare different operands. The relational operators are

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Examples** |
| > | Greater than | A=10,b=5,a>b =>0(true) |
| < | Less than | A=10,b=5,A<b =>-1(false) |
| >= | Greater than equals to | A=10,b=5,a>=b =>0(true) |
| <= | Less than equals to | A=10,b=5,A<=b =>-1(false) |
| == | equals to | A=10,b=5,A= =b =>-1(false) |
| != | not equals to | A=10,b=5,A!=b =>0(true) |

**Logical Operators**

The logical operators are used to combine two or more expressions into one. The logical operators are

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| && | And |
| || | Or |
| ! | Not |

****

**Bitwise Logical Operators**

The Bitwise logical operators are used to perform bit calculations. The Bitwise logical operators are

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| >> | Right Shift |
| << | Left Shift |
| ~ | Complement |
| ^ | XOR |

**Assignment Operators**

The assignment operators are used to assign R.H.S value to L.H.S either before calculation or after calculation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Examples** |
| = | Assigns RHS value to LHS | **a=10** |
| += | Assigns RHS value to LHS after addition | **a=10**  **a+=3=>13**  **or**  **a=a+3=>13** |
| -= | Assigns RHS value to LHS after subtraction | **a=10**  **a - =3=>7**  **or**  **a=a - 3=>7** |
| \*= | Assigns RHS value to LHS after multiplication | **a=10**  **a\*=3=>30**  **or**  **a=a\*3=>30** |
| /= | Assigns RHS value to LHS after Division | **a=10**  **a/=2=>5**  **or**  **a=a/2=>5** |
| %= | Assigns RHS value to LHS after modulus | **a=10**  **a%=3=>1**  **or**  **a=a%3=>1** |

**Unary Operator**

The Unary operators are used to perform unary calculations. The unary operators are incrementation and decrementation. The incrementation can be post incrementation or pre-incrementation and the decrementation post decrementation or pre-decrementation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Examples** |
| ++ | Incrementation | A=5  A++ =>6 |
| -- | Decrementation | A=5  A-- =>4 |

**Ternary Operators**

It is also called as conditional operator. The conditional operators are used to execute true statement only when the condition is true otherwise it executes false statement.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| ? | Question mark |
| : | colon |

**4. DATATYPES**

Data types are used to define the variables that the same of type data it can store in memory. The data types determine the type of data to be stored in memory. The data types are used to represent the different values to be stored in the variable. They are

1. Pre-defined data types
2. User-defined data types
3. Derived data types

**Pre-defined data types:**

The data types which are already existed in C they are known as pre-defined data types. The pre-defined data types are

|  |  |  |  |
| --- | --- | --- | --- |
| Name | **Types** | **Range** | **Memory** |
| Integer | Signed Int | -32,678 to +32,767 | 2 bytes |
| Unsigned Int | 0 to 65,535 | 2 bytes |
| Signed Long Int | -2,148,483,648 to +2,148,483,647 | 4 bytes |
| Unsigned Long Int | 0 to 4,296,967,295 | 4 bytes |
| Real | Float | 3.4 \* 10 -38 to 3.4 \* 10 +37 | 4 bytes |
| Double | 1.7 \* 10 -308  to 1.7 \* 10+ 307 | 8 bytes |
| Char | Signed char | -128 to +127 | 1 byte |
| Unsigned char | 0 to 255 | 1 byte |

**User-defined data types:**

The datatypes which creates by the user is called as user-defined data types. The user-defined data types can be created by using keywords like struct, class, enum, etc.

**Derived data types:**

The types which can be created using pre-defined data types, they are known as derived data types. The derived data types that we can create are **arrays, functions, files, pointers etc.**

**String:**

A sequence of characters which are organized under one variable or heading, it is called as a string. A string can be represented within the double quotes.

Eg: s1=”lotus-123”;

s2=”[anil.123@gmail.com](mailto:anil.123@gmail.com)”;

**Special Symbols:**

The symbols which have special functionality are called as special symbols.

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| [ ] | Square Brackets |
| ( ) | Parantheses |
| { } | Braces |
| “ “ | Double Quotes |
| ‘ ‘ | Single Quotes |

**Constant**

Constant is also called as **Literals.** A constant is a fixed value that will not be changed ever during the execution of the program. Once the programmer defines a value to a constant, it remains same through the entire program. C supports several types of constants.

1. Integer constants
2. Real constants
3. Character constants
4. String constants
5. Backslash character constants
6. **Integer constants:**

An integer constant is a sequence of numerical digits. There are three types of integer constants.

* + - 1. **Decimal integer constant** consists of a set of digits 0 to 9. Decimal integers may be either positive or negative.

**Eg: 1234, 3443, -1233, 0 , 92929**

* + - 1. An **octal integer constant** consists of any combination of digits from the set 0 to 7 with a leading 0. Octal values have no sign.

**Eg: 034 , 027, 0736**

* + - 1. **Hexadecimal integer constant** consist a set of digits 0 to 9 and alphabets A to F to represent the values 10 to 15. Each hexadecimal value begins with 0x.

**Eg: 0x20, 0xF5, 0xabf6**

1. **Real Constants:**

The numbers containing fractional parts are called as **real constants**. To represent fixed floating point values, we use real constants. These numbers are represented with a decimal value containing a decimal point. A real constant may be either positive or negative

**Eg: 3.1428, 0.000455, -1.346**

A real number may also be expressed in exponential notation.

**Eg: 2.1565e2**

E2 means multiply the number 102

Then the above value becomes 215.65

1. **Character constants:**

A character constant contains single character enclosed within a pair of single quotation marks. The character may be an alphabet, digit or any symbol.

**Eg: ‘C’, ‘8’,’$’**

1. **String constants:**

A string constant is a sequence of characters enclosed between a pair of double quotation marks. A string constant may contain alphabets, digits, any symbols and white spaces.

**Eg: “hello world” “1234” “@#$&^”**

1. **Backslash Character constants:**

C supports backslash character constants that are used to format the output presented to the user. These characters are also called as **escape sequence characters**.

|  |  |
| --- | --- |
| **Code** | **Meaning** |
| \n | New line |
| \a | Beep sound |
| \t | Horizontal Tab space |
| \’ | Single quotes |
| \” | Double Quotes |
| \\ | Backslash |
| \0 | Null value |
| \r | Carriage return |

**Variable**

A variable is nothing but a space in a memory where values can be constantly changed during run time of a program. To define variables the following rules to be followed. They are

* 1. Variable name should always begin with alphabet
  2. In between variable name there should not be any special symbol except underscore.
  3. Duplicate variable names cannot be defined.
  4. Keywords should not be defined a variables

**Types of Variables:**

**Local variable**

A local variable will be visible to the function in which it is declared, but not to other. It is declared inside the function. A local variable is also called automatic variable, because it is automatically created and destroyed.

**Global variable**

A global variable is visible to all the functions and it is declared outside the function. The proper place of declaration of global variable is at the beginning of the main program or before the function sub program. In other word before the declaration of all the function sub programs.

**Variable Declaration:**

Datatype var1,var2,…;

**Eg:**int a,b;

Float m,n;

Char c;

Char s[15];

**Initialization of Variable:**

**Eg:**int a=10,b=20;

float x=1.2;

char c=’A’;

char s[15]={“Muthahar”};

**Structure of C-Program**

Structure of C program contains 6 sections. They are –

1. **Documentation Section**

The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage. The comment about a program can be written within the following symbols

**/\* Paragraph Comment \*/**

1. **Link Section**

The link section usually contains header files include statement. It provides instructions to the compiler to link functions from the C standard library.

#include<headerfile.h>

1. **Definition section**

The definition section defines all symbolic constants. So it contains preprocessor commands which always start with # as first character.

#define cons\_name value

1. **Global Declaration section**

All global variables are declared in the global declaration section. Variables declared outside all functions are called global variables and they may be accessed by any function within our program. Global variables exist the entire time your program is executing.

1. **Main function section**

This section is a part of every c program. Any c program is a set of functions which must be main() function. The empty parentheses after main() function is necessary and indicate that this main() function doesn’t include any arguments.

This section contains two parts, **local variable declaration part** and **executable part**

**The declaration part** declares all local variables used in the executable part. Every function contains one or more instructions which are enclosed within the pair of curly braces. Each statement should end with a semi-colon.

**Variables declared inside a function are called local variables**. A local variable is known to and may be accessed by only the function in which it is declared.

1. **Sub program section**

It contains all the user defined functions that are called in the main() function. User defined functions are placed immediately after the main() function.

**Input and Output Functions**

1. **printf()**

It is a standard formatted output function, which is used to display an output during run time of a program.

***Syn:***printf(“<Message><FormatControls><EscapeSequences>”, var1,var2);

**Format Controls**

The format controls are used to display the output as per the format specified by the user. The Format controls are –

|  |  |
| --- | --- |
| **Format** | **Meaning** |
| %d (or) %i | Print an integer |
| %u | Print an Unsigned integer |
| %ld | Print a long integer |
| %f | Print float value |
| %e | Print float value in exponential form |
| %c | Print a character |
| %s | Print a string |
| %x | Print hexadecimal integer using lower case a-f |
| %X | Print hexadecimal integer using lower case A-F |
| %o | Print octal value |
| %lo | Print octal long |
| %p | Print a pointer value |

**Escape Sequences:**

These are non-printable characters but its functionality can be observed on the output screen.

|  |  |
| --- | --- |
| **Code** | **Meaning** |
| \n | New line |
| \a | Beep sound |
| \t | Horizontal Tab space |
| \’ | Single quotes |
| \” | Double Quotes |
| \\ | Backslash |
| \0 | Null value |
| \r | Carriage return |

1. **scanf():**

It is a standard formatted input function. It is used to input values during run time of a program.

***Syn:***Scanf(“format control”,&var1,&var2);

**Note:** While input data during runtime of a program, in front of a variable ampersand (&) should be defined. But while accepting string “&” operator should not be defined in front of a variable

1. **getchar():**

It is a character input function. It is used to input/read number of characters during run time of a program and it can store only the starting character. This function is available in a header file called stdio.h.

**Syn:**var=getchar()

1. **getche():**

It is a character input function. It is used to input/read character during run time of a program and it can store in the memory. This function is available in a header file called stdio.h.

**Syn:**var=getche()

1. **getch():**

It is a character input function. It is used to input/read character during run time of a program and it will never echoes onto the screen but it can store in the memory. This function is available in a header file called stdio.h.

**Syn:**var=getch()

1. **putchar():**

It is a character output function which is used to display character as output during run time of a program.

**Syn:** putchar(var)

**Types of files used in a C program**

**Ans:** Every C program has four kinds of files associated with it. They are

1. **Source code file**

The source code file contains the source code of the program. The file extension of any C source code file is “.c”. This file contains C source code that defines the main function and may be other functions. The main() is the starting point of execution when we successfully compile and run the program.

1. **Header files**

In C, we were using many functions that were not written by us. But we don’t know the details of how these functions work. Such functions that are provided by all c compilers are included in standard header files. Such as stdio.h, conio.h, string.h, math.h, etc.

1. **Object files**

Object files are generated by the compiler as a result of processing the source code file. Object files contain compact binary code of the function definitions. Linker uses this object file to produce an executable file by combining the object files together.

1. **Binary executed files**

The binary executed file is generated by the linker. The linker links various object files to produce a binary file that can be directly executed. On windows operating system, the executable files have “.exe” extension.

**Differences between compiler and interpreter**

**Ans:**

|  |  |
| --- | --- |
| **Compiler** | **Interpreter** |
| 1. Translates the entire program | 1. Translates the program line by line |
| 2. Requires more main memory | 2. Requires less main memory |
| 3. Converts the entire program to machine code, when all the syntax errors are removed and executes the object code directly | 3. Each time the program is executed every line is checked for syntax and then converted to equivalent machine code. |
| 4. Neither the source nor the compiler are required for execution | 4. Source program and the interpreter are required for execution. |
| 5. Slow for debugging and testing | 5. Good for fast debugging and testing |
| 6. Execution time is less | 6. Execution time is more |
| 7. Security for source code | 7. No security for source code. |

**Type casting**

Type conversion refers to changing a value from one data type to another data type. It is also called as type casting. C supports two kinds of type casting in expressions. They are

1. **Implicit type casting:**

Automatic casting can be done by C compiler is called implicit type casting. It is done to convert a lower data type into a higher data type.

1. **Explicit type casting:**

The casting is done by the programmer is called explicit casting**.** Explicit casting is compulsory while converting from a higher data type to a lower data type.

**Syn:** (Data type) variable/expression

**Control Structures / Control Statements**

1. Conditional Statements
2. Iterative Statements
3. Jumping Statements / Unconditional Statements
4. **Conditional Statements**

The conditional statements are used to control the flow of execution of statements of a program. The conditional statements are

1. Simple if
2. if .. else
3. if .. else if
4. Nested if
5. Switch
6. Ternary Operator

**Simple if:**

This conditional statement is used to execute true statements only when the condition is true, otherwise it skips the if statement.

**Syntax**

**:**if (expression )

{

Statement1;

Statement2;

}

**if ..else:**

This conditional statement executes true statements only when the condition is true otherwise it executes false statements.

**Syntax:**

if (expression )

{

statement1;

statement2;

}

else

{

statement1;

statement2;

}

**if ..else if:**

It is also called as a branching statement or Ladder statement. This conditional statement executes the statements based on its respective conditional statement.

**Syntax:**

if ( expression )

{

statement1;

statement2;

}

else

if(expression2)

{

statement1;

statement2;

}

else

{

Statement1;

Statement2;

}

**Nested if:**

A if statement which can execute within an if statement itself, it is called as nested if conditional statement. It is also called as a multi-level branching statement.

**Syntax:** if(exp1)

{

if(exp1.1)

{ statement1;

statement2;

}

else

if(exp1.2)

{ statement1;

statement2;

}

}

else

if(exp2)

{

if(exp2.1)

{ statement1;

statement2;

}

else

if(exp2.2)

{ statement1;

statement2;

}

}

**Switch:**

It is also called as branching statement. This conditional statement is very similar to if ..else if conditional statement but the execution of switch statement is very faster than if..else if conditional statement.

**Syntax:** switch(exp)

{

case constant1 : statement1;

statement2;

break;

case constant2: statement1;

statement2;

break;

case constant3: statement1;

statement2;

break;

default : statement;

}

**Ternary operator**

The ternary operators are also called as conditional operators. The ternary operators are used to control the flow of execution of statements of a program. The conditional operators execute a true statement only when the condition is true otherwise it executes a false statement.

**Syntax:** Expression? True statement : false statement;

**Loops**

Executing a statement or group of statements for a repeated number of times, it is called as a loop. C-language contains different types of loops. They are –

1. for .. loop
2. while .. loop
3. do .. while loop

**For ..loop:**

The for loop iterates the elements for the fixed number of times. It should be used if number of iterations is known. The for ..loop can be construct by using the following 3 steps.

1. **Initialization**

Here, we initialize the variable in use, it marks the start of a for..loop. An already declared variable can be used or a variable can be declared local to loop only.

1. **Testing Condition:**

It is used for testing the exit condition for a loop. It must return a boolean value. It is also an entry control loop as the as the condition is checked prior to the execution the loop statements.

1. **Changing value:**

It is used for updating the variable for next iteration.

|  |  |
| --- | --- |
| **Syn-1:**  for(initialvalue; conditionalvalue;changingvalue)  {  block of statements;  }  statement -x;  **Syn-2:**  initialvalue;  for( ;conditionvalvalue; )  {  block of statements;  changingvalue;  } | for loop in c language flowchart |

**Nested for..loop:**

A for..loop which executes within another for..loop itself, such loop can be called as nested for..loop. In nested for..loops, the execution of loops always depends on the conditional values of the loops. In nested for..loop the inner loop will execute first and then the outer loop will get executed.

**Syntax:**for(initialvalue;conditionalvalue;changingvalue)

{

for(initialvalue;conditionalvalue;changingvalue)

{

block of statements;

}

}

**While ..loop:**

* It is also called as a conditional loop.
* While loop is also known as a pre-tested loop
* In this loop a statement or group of statements can be executed for a repeated number of times only when the condition is true otherwise the loop will be terminated.
* The while loop is mostly used in the case where the number of iterations is not known in advance.
* The minimum iterations in this loop is zero
* It is also called as Entry..Exit loop

|  |  |
| --- | --- |
| **Syn:**  initial value;  while(expression)  {  statement1;  statement2;  .  .  changing value;  } | flowchart of c while loop |

**do ..while loop:**

* The do while loop is a post tested loop.
* In this loop the statements of a loop will execute first and then the condition will be verified. Even though the condition is false the statements of loop will execute at least once.
* This loop will be executed for repeated number of times only when the condition is true.
* The do-while loop is mainly used in the case where we need to execute the loop at least once.
* The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.
* It is also called as Exit .. Entry loop.

|  |  |
| --- | --- |
| **Syntax:**  initial value;  do  {  statement1;  statement2;  changing value;  } while (expression); | flowchart of do while loop in c language |

**JUMPING STATEMENTS**

There are three different controls used to jump from one c program statement to another and make the execution of the programming procedure fast. These three jumping controls are

1. **goto statement**
2. **break statement**
3. **continue statement**

**goto statement**

The powerful jumping statement in the C language is goto statement. It is sometimes also called part of branching statement. The goto moves the controls on a specified address called label or label name. The goto is mainly of two types. One is conditional and the other is unconditional. Also jump can be either in forward direction or in backward direction. The different types of goto statement is

1. **forward goto**
2. **backward goto**

**Forward goto:**

In this the control moves forward at a specified label either according to a condition or without condition.

**Syntax for unconditional forward goto:**

Statement1;

Statement2;

**Goto label;**

Statement3;

Statement4;

**Label:**

Statement5;

Statement6;

**Syntax for conditional forward goto:**

Statement1;

Statement2;

**If(condition)**

**Goto label;**

Statement3;

Statement4;

**Label:**

Statement5;

Statement6;

**Backward goto:**

The backward goto or backward jump moves the control back to the specified addresses and so creates a loop. In the case of conditional backward statement, it creates finite looping. But in the case of unconditional backward goto or jump, it creates infinite looping.

**Syntax for unconditional backward goto statement:**

Statement1;

**Label:**

Statement2;

Statement3;

**Goto label;**

Statement4;

**Syntax for conditional backward goto statement:**

Statement1;

**Label:**

Statement2;

Statement3;

**if(condition)**

Goto label;

Statement4;

**Break**

* A break statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop i.e, the break statement is used to terminate loops or to exit from a switch
* It can be used within a for, while, do-while or switch statement.
* The break statement is written simply as **break;**

**Syntax:**

while(condition)

{

statement1;

statement2;

if(condition)

{

break;

}

statement-x;

}

**Continue statement:**

* The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered
* Instead the remaining statements are skipped and the computation proceeds directly to the next pass through the loop.
* The continue statement can be included within a while, a do-while, a for statement
* It is simply written as “continue”. The continue statement tells the compiler “skip the following statements and continue with the next iteration.
* In “while” and “do-while” loops continue causes the control to go directly to the test condition and then to continue the iteration process

**Syntax:**

while(condition)

{

Statement1;

Statement2;

if(condition)

{

Continue;

}

Statement3;

Statement4;

}

Statement-x;

**// Program to calculate the sum of a maximum of 10 numbers, If a negative number // is entered, the loop terminates**

# include <stdio.h>

# include <conio.h>

main()

{

int i,a,s;

clrscr();

for(i=1; i <= 10; ++i)

{

printf("Enter a n %d: ",i);

scanf("%d",&a);

if(a < 0)

break;

s += a;

}

printf("Sum = %d",s);

getch();

}

**Output**

**Enter a n1: 2**

**Enter a n2: 4**

**Enter a n3: 3**

**Enter a n4: -3**

**Sum = 9**

**// Program to calculate the sum of a maximum of 10 numbers, Negative numbers**

**// are skipped from the calculation**

# include <stdio.h>

# include <conio.h>

main()

{

int i,a,s;

clrscr();

for(i=1; i <= 10; ++i)

{

printf("Enter a n %d: ",i);

scanf("%d",&a);

if(a < 0)

continue;

s += a;

}

printf("Sum = %d",s);

}

**Output**

**Enter a n1: 1**

**Enter a n2: 2**

**Enter a n3: 5**

**Enter a n4: 4**

**Enter a n5: -3**

**Enter a n6: -45**

**Enter a n7: 34**

**Enter a n8: -42**

**Enter a n9: -1000**

**Enter a n10: 12**

**Sum = 58**

**UNIT – III**

**Array:**

* An array is defined as the collection of similar type of data items/values stored at contiguous memory locations. It is also called as **homogenous type**.
* Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
* It also has the capability to store the collection of derived data types, such as pointers, structure, etc.
* The array is the simplest data structure where each data element can be randomly accessed by using its index number.
* By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array

**Advantages of Array**

1. It is better and convenient way of storing the data of same datatype with same size.
2. It allows us to store known number of elements in it.
3. It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
4. Iterating the arrays using their index is faster compared to any other methods like linked list etc.
5. It allows to stores the elements in any dimensional array - supports multidimensional array.

**Disadvantages of Array:**

1. Array is **Static data** Structure
2. Memory Allocated during **Compile time**.
3. It allows us to enter only fixed number of elements into it.
4. We cannot alter the size of the array once array is declared.
5. Inserting and deleting the records from the array would be costly since we add / delete the elements from the array, we need to manage memory space too.

**Types of Arrays**

Arrays are of different types. They are –

1. Single Dimensional Array
2. Double Dimensional Array
3. Multi-Dimensional Array

**SINGLE DIMENSIONAL ARRAY:**

An array which contains only one row and it can have set of columns. It is called as a single dimension array. It is also called as **one dimensional array** or **single subscripted value array**. Array position always starts from **“zero”**. Each section of an array is called as an **“element”**.

## Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

## Syn: datatype array-name[size];

## Eg: int a[5];

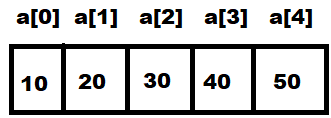
## 

## Initializing Arrays

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

**Initialization of values in an array** –

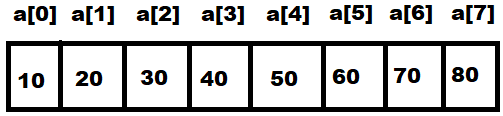
Eg: - 1. int a[5]= { 10,20,30,40,50 };



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 −

2. int a[] ={10,20,30,40,50,60,70,80};

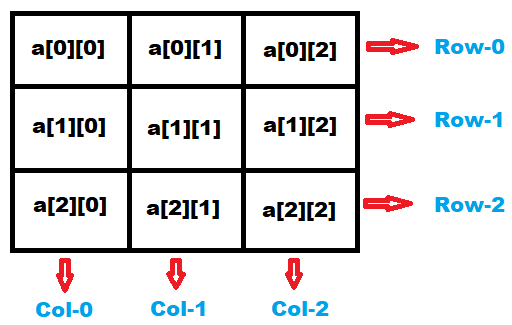


**DOUBLE DIMENSIONAL ARRAY:**

An array which contains number of rows and number of columns, it is called as a double dimension array. It is also called as Double subscripted value array.

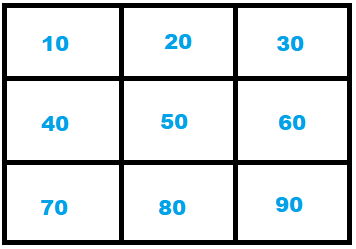
***Syn:*** type array\_name[rs][cs];

**Eg:- 1**. int a[3][3];



**Initialization of values in array –**

int a[][]= {{10,20,30}, {40,50,60}, {70,80,90}}



## Multi-Dimensional Array

## C allows arrays of three or more dimensions The exact limit is determined by the compiler.

## Syn: datatype array\_name [size1] [size2] …..[sizeN];

## Where, array\_name is the name of the array. Size1, size2,…..sizeN refers size of each array locations

## Three Dimensional Array

## A 3D array is a multi-dimensional array. A 3D arrays is a collection of 2D arrays. It is specified by using three subscripts. Block size , row size and column size.

## Syn: datatype array\_name [Block-size] [row-size][column-size];

## Eg: int a[3][3][3];

## Initialization of 3D Array

## You can initialize a three-dimensional array in a similar way like a two-dimensional array

## Eg: int a[2][3][4]={ { {3,4,2,3}, {1,2,3,4}, {10,20,30,40}}, { {6,7,8,9},{4,5,6,7},{6,7,8,9}}}

## Operations on Arrays

Following are the basic operations supported by an array.

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.

## Insertion Operation

## Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the *beginning*, *end*, *or any given index of array.*

### Example

**#include<stdio.h>**

**#include<conio.h>**

main()

{

int a[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

clrscr();

printf("The original array elements are :\n");

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

printf("a[%d] = %d \n", i, a[i]);

n = n + 1;

while( j >= k)

{

a[j+1] = a[j];

j = j - 1;

}

a[k] = item;

printf("The array elements after insertion :\n");

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

printf("a[%d] = %d \n", i, a[i]);

}

**Output**

**The original array elements are:**

a[0] = 1

a[1] = 3

a[2] = 5

a[3] = 7

a[4] = 8

**The array elements after insertion:**

a[0] = 1

a[1] = 3

a[2] = 5

a[3] = 10

a[4] = 7

a[5] = 8

1. **Deletion Operation**

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

**Example**

**#include <stdio.h>**

**#include <conio.h>**

main()

{

int LA[] = {1,3,5,7,8};

int k = 3, n = 5;

int i, j;

clrscr();

printf("The original array elements are :\n");

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

{

printf("LA[%d] = %d \n", i, LA[i]);

}

j = k;

while( j < n)

{

LA[j-1] = LA[j];

j = j + 1;

}

n = n -1;

printf("The array elements after deletion :\n");

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

printf("LA[%d] = %d \n", i, LA[i]);

}

**Output**

The original array elements are:

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after deletion:

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

**Character array**

A collection of characters which are organized under one variable, it is called as a character arrays. In a character array when we initialize any string by default it contains NULL value at the last of character array.

**Syn: datatype var[size];**

**Eg:-** char s[10];

**Initializationof string in Character array**

char s[10]={"hello"};

s[0]=’h’

s[1]=’e’

s[2]=’l’

s[3]=’l’

s[4]=’o’

s[4]='\0'

**Character I/O Functions:** The character i/o functions are -

* + getchar()
  + getche()
  + getch()
  + putchar()

1. **getchar()**

It is a character input function. This function can able to read number of characters but it can able to store only the starting character.

**Syntax:** var=getchar()

1. **getche()**:

It is a character input function. This function is used input only one character.

**Syntax:** var=getche()

1. **getch()**

It is a character input function. This function is used to input only one character but it will never echoes on to a screen.

**Syntax:** var=getch()

1. **putchar()** – This is output function. This function is used to display a character.

**Syntax:**putchar(var);

**String handling functions:**

String handling functions are available in a header file called string.h. The string handling functions are

1. **Strcmp():**

This function is used to compare two different strings and find whether they are exact or not. If they are exact it returns 0 otherwise it returns -1.

**Syn:**strcmp(string1,string2)

**Eg:**strcmp(“hello”,”Hello”); => -1 (false)

Strcmp(“hello”,”hello”) => 0 (true)

1. **Strcat():**This function is used to combine two strings into one.

**Syn:**strcat(string1,string2)

**Eg:**strcat(“hello”,”world”) => helloworld

1. **strrev():** This function is used to reverse the given string.

**Syn:**strrev(string)

**Eg:**strrev(“hello”) =>olleh

1. **Strlen():** This function is used to find the length of a given string

**Syn:**strlen(string)

**Eg:**strlen(“hello”) => 5

1. **Strupr():** This function is used to convert a given string into upper case.

**Syn:**strupr(string)

**Eg:**strupr(“hello”) =>HELLO

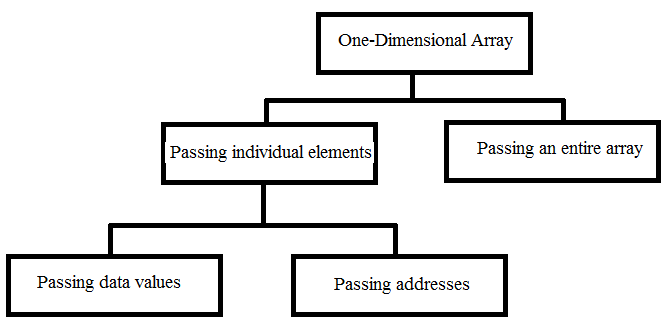
1. **Strlwr():** This function is used to convert a given string into lower case.

**Syn:**strlwr(“string”)

**Eg:**Strlwr(“HELLO”) =>hello

**One dimensional array for inter-function communication:**

Like variables of other data types, we can also pass an array to a function. While in some situations, we may want to pass individual elements of the array, and in other situations we may want to pass the entire array.



**Passing individual elements:**

The individual elements of an array can be passed to a function either by passing their addresses or their data values.

**Passing data values:**

The individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match the type of the function parameter.

**Passing addresses**

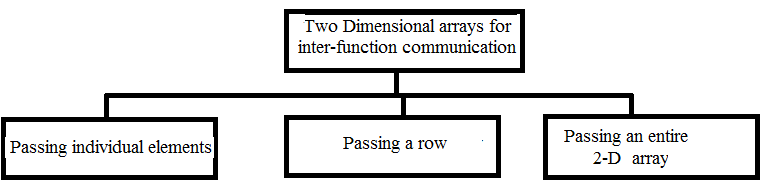
Now, we can pass the address of an individual array element by preceding the address operator to the element’s indexed reference

**Passing the entire array**

We know that the array name refers to the first byte of the array in memory. The address of rest of the elements in the array can be calculated using the array name and the index value of the element. Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array.

**Two dimensional array for inter-function communication**

There are three ways of passing parts of the two-dimensional array to a function.



**Pass individual elements of the array.** This is exactly same as we passed elements of a one-dimensional array.

**Pass a single row of the two-dimensional array.** A row of a two-dimensional array can be passed by indexing the array name with the row number. When we send a single row of a two-dimensional array, then the called function receives a one-dimensional array.

**Pass the entire two-dimensional array to the function:**

To pass a two-dimensional array to the parameter in the called function must indicate that the array has two dimensions

**UNIT – IV**

**Structures**

A collection of different types of values which can be stored under, one variable or heading, it is called as a structure. It is also called as heterogeneous type or user-defined type.

A structure can be created by using a keyword called **struct**, followed by structure name and the members of a structure should be enclosed within the pair of braces. A structure can be terminated by using semi-colon immediately after a closing brace.

The members of a structure can be accessed by defining structure variable and along with the help of period operator.

**Syntax** - struct struct\_name

{

member1;

member2;

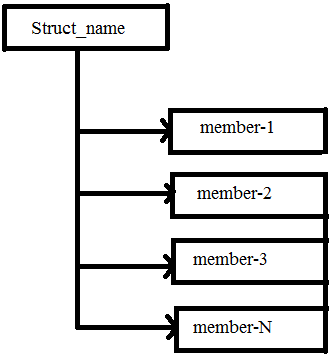
member3;

member n;

};

struct struct\_name var1,var2....;

**Graphical representation of structure:**



**eg:**- struct student

{

int rno,fees;

char nm[15],co[15];

};

struct student x;

x.rno=100

x.nm="anil";

x.co="bsc";

x.fees=5000

**Array of structures:**

A collection of values of different types which can be stored under each element of a array is called as structure using arrays.

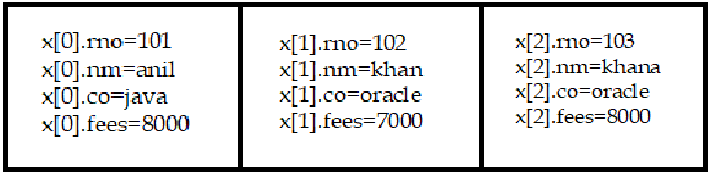
**Eg:-** struct student

{

int rno,fees;

char nm[15],co[15];

}x[3];



Here x is an array of 3 student structures. Each element of the array x will contain the structure of type student. The student structure consists of four individual members: rno, nm, co and fees.

**Structure using pointer:**

A collection of values of different types which can store under one variable or heading, it is called as a structure. The members of structure can be accessed by defining normal structure variable and pointer structure variable. If we want to access the members of structure using pointer structure variable then we have to make use of “->” symbol.

**Eg:** struct stud

{

int rno,fees;

char nm[15],co[10];

float height;

}\*x;

x->rno=1001 (\*x).rno=1001

x->nm="kiran" (\*x).nm="kiran"

x->co="btech" (or) (\*x).co="btech"

x->height=5.6 (\*x).height=5.6

x->fees=20000 (\*x).fees=20000

**Nested Structures:**

A structure can be nested inside another structure. In other words, the members of a structure can be of any other type including structure. **There are two ways to define nested structure in c language:**

1. By separate structure
2. By Embedded structure
   1. **Separate structure**

We can create 2 structures, but dependent structure should be used inside the main structure as a member.

**Eg:** struct Date

{

int dd, mm,yy;

};

struct Employee

{ int id;

char name[20];

struct Date doj;

}emp1;

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

* 1. **Embedded structure**

We can define structure within the structure also. It requires less code than previous way. But it can't be used in many structures.

**Syn:** struct tagname\_1

{

member1;

member2;

...

member-n;

struct tagname\_2

{

member\_1;

member\_2;

...

member\_n;

}var1;

} var2;

**Eg:**

struct Employee

{ int id;

char name[20];

struct Date

{

int dd,mm,yyyy;

}doj;

}e1;

**Self-referential structure**

A self-referential structure contains pointer to a data that is of the same type as that of the structure. It means a structure can have data part as well as address part. A structure can be referred as a node.

**Eg:** struct data

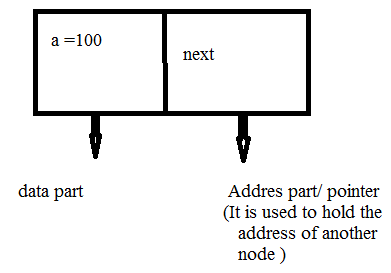
{

int a;

struct data \*next;

};

struct data \*s;



**Union**

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program.

**Syn:** union [union tag]

{

member definition;

member definition;

...

member definition;

} [one or more union variables];

**The union tag is optional** and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

**Eg:** union Data

{

int i;

float f;

char str[20];

} data;

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. **For example,** in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

**Eg:**

#include <stdio.h>

#include <string.h>

union Data {

int i;

float f;

char str[20];

};

main( )

{

union Data data;

printf( "Memory size occupied by data : %d\n", sizeof(data));

getch();

}

**Output:**

Memory size occupied by data : 20

## Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type.

**Eg:**

#include <stdio.h>

union Data

{ int i;

float f;

char str[20];

};

main( )

{union Data data;

data.i = 10;

data.f = 220.5;

strcpy( data.str, "C Programming");

printf( "data.i : %d\n", data.i);

printf( "data.f : %f\n", data.f);

printf( "data.str : %s\n", data.str);

getch();

}

**Output:**

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

**Storage classes**

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

**C language uses 4 storage classes**, namely:

* + - 1. Automatic Variables
      2. Register Variables
      3. Static Variables
      4. External Variables

**Automatic Variables**

* Automatic variables are allocated memory automatically at runtime.
* The visibility of the automatic variables is limited to the block in which they are defined.
* The scope of the automatic variables is limited to the block in which they are defined.
* The automatic variables are initialized to garbage by default.
* The memory assigned to automatic variables gets freed upon exiting from the block.
* The keyword used for defining automatic variables is auto.
* Every local variable is automatic in C by default.

**Syntax: auto datatype var1,var2..;**

**Eg:**

#include <stdio.h>

main()

{ int a;

char b;

float c;

clrscr();

printf("%d %c %f",a,b,c);

getch();

}

**2. Register Variables**

* The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
* We cannot dereference the register variables, i.e., we cannot use & operator for the register variable.
* The access time of the register variables is faster than the automatic variables.
* The initial default value of the register local variables is 0.
* The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler’s choice whether or not; the variables can be stored in the register.
* We can store pointers into the register, i.e., a register can store the address of a variable.
* Static variables cannot be stored into the register since we cannot use more than one storage specifier for the same variable.

**Syntax**- register int c;

**Eg-1:**

#include <stdio.h>

#include <conio.h>

main()

{

register int a;

clrscr();

printf("%d",a);

getch();

}

**Output:**

0

**3. Static Variables**

* The variables defined as static specifier can hold their value between the multiple function calls.
* Static local variables are visible only to the function or the block in which they are defined.
* A same static variable can be declared many times but can be assigned at only one time.
* Default initial value of the static integral variable is 0 otherwise null.
* The visibility of the static global variable is limited to the file in which it has declared.
* The keyword used to define static variable is static.

**Syntax** - static int a;

**Eg-1**

#include<stdio.h>

#include<conio.h>

main()

{

int i;

for(i = 0; i< 3; i++)

sum();

}

void sum()

{

static int a = 10;

static int b = 24;

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

a++;

b++;

}

**Output:**

10 24

1. 25
2. 26

**4. External Variable**

* The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
* The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
* The default initial value of external integral type is 0 otherwise null.
* We can only initialize the extern variable globally, i.e., we cannot initialize the external variable within any block or method.
* An external variable can be declared many times but can be initialized at only once.
* If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

**Syntax** - extern n=100;

**Eg-1:**

#include <stdio.h>

int a;

main()

{

extern int a;

printf("%d",a);

getch();

}

**Output**

0

**Enumerated datatype:**

The enumeration data types are a recent addition to C. Similar concepts are available in other higher level programming languages such as Pascal and Ada. The enumeration data type is not supported by all C compilers. An enumeration type in C is a set of values represented by identifiers called enumeration constants. The enumeration constants are specified when the type is defined.

**Syntax:** enum enum\_name

{

member1,member2,..;

};

Where **enum** is a keyword for defining the enumeration data type and the braces are required. The members of the enumeration data type such as member1, member2, member N are the individual identifiers. Once the enumeration data type is defined, it can be declared in the following ways –

**Eg:-** enum sample

{

mon,tue,wed,thu,fri,sat,sun

} d1,d2,d3,d4,d5,d6,d7;

**Eg;-1**

#include <stdio.h>

main()

{

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

clrscr();

printf("Sun = %d", Sun);

printf("\nMon = %d", Mon);

printf("\nTue = %d", Tue);

printf("\nWed = %d", Wed);

printf("\nThu = %d", Thu);

printf("\nFri = %d", Fri);

printf("\nSat = %d", Sat);

getch();

}

**Output:**

Sun = 0

Mon = 1

Tue = 2

Wed = 3

Thu = 4

Fri = 5

Sat = 6

**Function**

A block of statements which performs a particular task is called as a function. Sub program is the independent and complete program. It is also called function program. A sub-program is independent because it can be called by the main program or other sub-program.

**Syntax: return\_type function\_name(arg1,arg2…)**

**{**

**Statement1;**

**Statement2;**

**}**

**Function declaration/prototype:**

A function declaration provides the following information to the compiler

The name of the function

The type of the value returned

The number of arguments, type of arguments and order of arguments that must be supplied in a call to the function.

When a function call is encountered, the compiler verifies the function call with its declaration, so correct argument types are used.

**Advantages of Function**

The length of source program can be reduced by using function by using it at different places in the program according to the user’s requirement.

By using function, memory space can be properly utilized. Also less memory is required to run program if function is used.

A function can be used by many programs.

Functions increases the execution speed of the program and makes the programming simple.

By using the function, portability of the program is very easy.

It removes the redundancy i.e, avoids the repetition and saves the time and space.

Debugging becomes very easier and fast using the function sub-programming.

Function are more flexible than library functions

Testing is very easy by using functions.

Reliability is high in the function programming

**Types of functions**

Functions are of different types. They are –

* 1. Library Functions
  2. User-defined Functions

**Library Functions**

The functions which are already existed in c-language is known as library functions. The Library functions are

**Eg**:- Printf( ), scanf( ), strlen( ), strcat( ), sqrt( )., etc.

**User-defined functions**

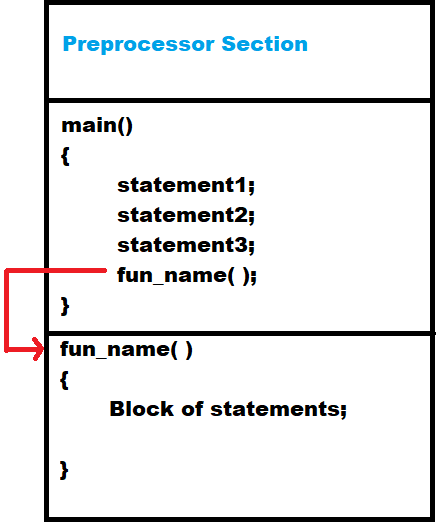
The function which creates by the user is called as user-defined functions. The user-defined functions are –

* 1. Function with no arguments no return values.
  2. Function with arguments but no return values.
  3. Function with arguments and return value
  4. Function with no arguments but return value.

1. **Function with no arguments no return value**

A function doesn’t contains any arguments, while calling a called function the called function will perform a particular task and it will never return a value to a calling function.

**Syn:**

****

1. **Function with arguments no return value**

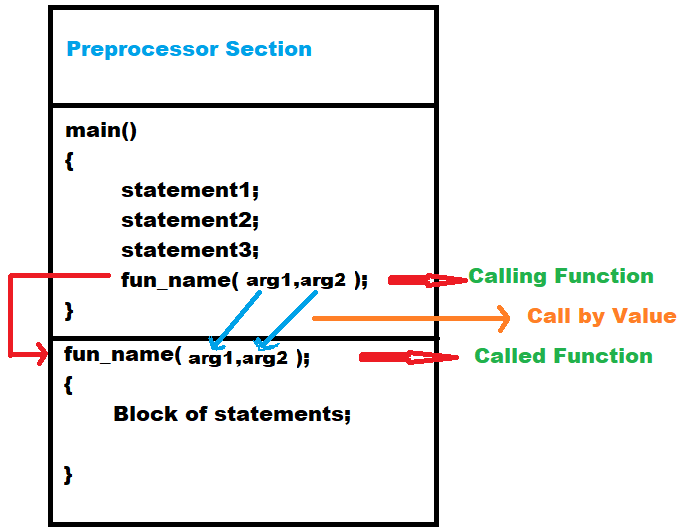
A function contains arguments and while calling a called function the arguments can be passed from calling function to called function, this passing of arguments from calling function to called function is called as **call by value.**

A calling function and a called function should have the same types of arguments but the argument names of calling function and called function may be same or not similar. If the names of the arguments in both calling and called function are same then such arguments can be called as **actual arguments.**

If the names of the arguments in both calling and called function are not same then such arguments can be called as **formal arguments or dummy arguments or parameterized arguments.**

The arguments passed to a called function can be make use to perform required task and the resultant value will not be returned to a calling function.

**Syntax:**

****

1. **Function with arguments and return value**

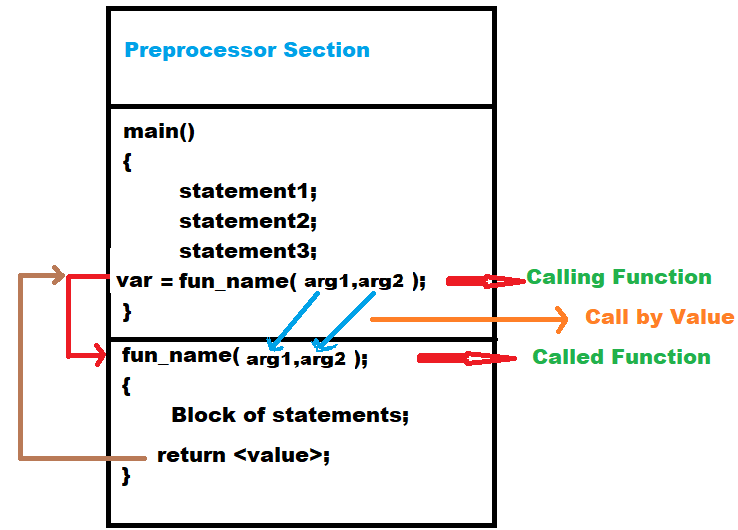
A function contains arguments and while calling a called function the arguments can be passed from calling function to called function, this passing of arguments from calling function to called function is called as **"call by value".**

A calling function and a called function should have the same types of arguments but the argument names of calling function and called function may be same or not similar. If the names of the arguments in both calling and called function are same then such arguments can be called as **actual arguments.**

If the names of the arguments in both calling and called function are not same then such arguments can be called as **formal arguments or dummy arguments or parameterized arguments.**

The arguments passed to a called function can be make use to perform required task and the resultant value can be returned to a calling function by using the statement **"return"**.

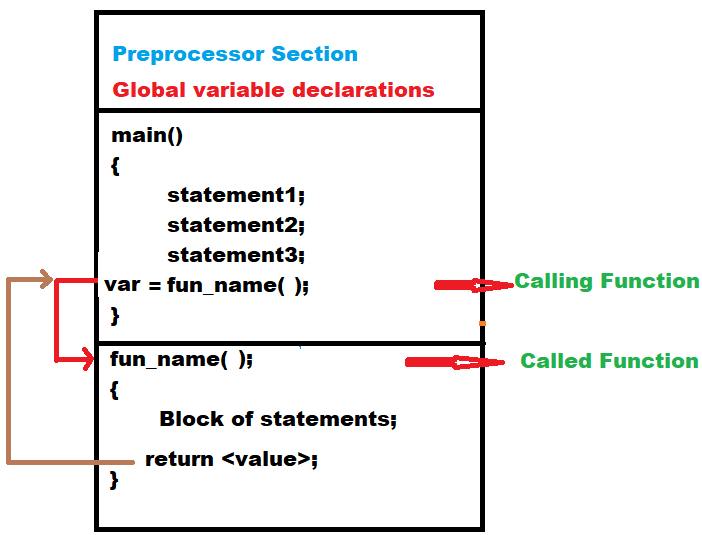
**Syntax:**

****

1. **Function with no arguments and no return value**

A function contains arguments, while calling a called function, while executing the called function the called function will look after any global variable declarations exist or not, if exist then those argument values can be utilized within the function and performs a particular task, the resultant value will return to a calling function

**Syntax:**



**Passing array to function**

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

**Methods to declare a function that receives an array as an argument**

**There are 3 ways to declare the function** which is intended to receive an array as an argument.

* 1. **Declaring blank subscript notation [] is the widely used technique.**

return\_type function(type arrayname[])

**Eg:**

#include<stdio.h>

main()

{ int i=0,min=0;

int a[]={4,5,7,3,8,9};

min=minarray(a,6);

printf("minimum number is %d \n",min);

getch();

}

intminarray(int arr[],int size)

{ int min=arr[0];

int i=0;

for(i=1;i<size;i++)

{

if(min>arr[i])

min=arr[i];

}

return min;

}

**Output**

minimum number is 3

* 1. **Optionally, we can define size in subscript notation [].**

return\_type function(type arrayname[SIZE])

**Eg:**

#include<stdio.h>

main()

{

int n,a[100],i;

printf("\n Enter N value");

scanf("%d",&n);

printf("\n Enter %d values in array",n);

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

scanf("%d",&a[i]);

sort(a,n);

printf("\n After sorting");

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

printf("\n a[%d]=%d",i,a[i]);

getch();

}

**sort(int a[100],int n)**

{

int i,j,t;

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

{

for(j=i+1;j<n;j++)

{

if(a[i]>a[j])

{ t=a[i];

a[i]=a[j];

a[j]=t;

}

}

}

* 1. **return\_type function(type \*arrayname)**

You can also use the concept of a pointer.

**Eg:** #include<stdio.h>

main ()

{ int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

int \*p = Bubble\_Sort(arr), i;

printf("printing sorted elements ...\n");

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

printf("%d\n",\*(p+i));

}

int\* Bubble\_Sort(int a[])

{

int i, j,temp;

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

{

for(j = i+1; j<10; j++)

{

if(a[j] < a[i])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

return a;

}

**Passing structure to function**

* A structure can be passed to any function from main function or from any sub function.
* Structure definition will be available within the function only.
* It won’t be available to other functions unless it is passed to those functions by value or by address(reference).
* Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program
* Passing structure to function can be done in 3 ways

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global
   * 1. **PASSING STRUCTURE TO FUNCTION IN C BY VALUE:**

It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

**Eg:**

#include <stdio.h>

#include <string.h>

struct student

{

int id;

char name[20];

float percentage;

};

main()

{ struct student record;

record.id=1;

strcpy(record.name, "Raju");

record.percentage = 86.5;

display\_studentinfo(record);

getch();

}

void display\_studentinfo (struct student record)

{

printf(" Id is: %d \n", record.id);

printf(" Name is: %s \n", record.name);

printf(" Percentage is: %f \n", record.percentage);

}

**OUTPUT:**

Id is: 1

Name is: Raju

Percentage is: 86.500000

* + 1. **PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:**

It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

**Eg:**

#include <stdio.h>

#include <string.h>

struct student

{

int id;

char name[20];

float percentage;

};

main()

{

struct student record;

record.id=1;

strcpy(record.name, "Raju");

record.percentage = 86.5;

display\_studentinfo (&record);

getch();

}

display\_studentinfo (struct student \*record)

{

printf(" Id is: %d \n", record->id);

printf(" Name is: %s \n", record->name);

printf(" Percentage is: %f \n", record->percentage);

}

**OUTPUT:**

Id is: 1

Name is: Raju

Percentage is: 86.500000

* + 1. **TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:**

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don’t need to pass the structure to any function separately.

**Eg:**

#include <stdio.h>

#include <string.h>

struct student

{

int id;

char name[20];

float percentage;

};

struct student record; // Global declaration of structure

main()

{

record.id=1;

strcpy(record.name, "Raju");

record.percentage = 86.5;

structure\_demo();

getch();

}

void structure\_demo()

{

printf(" Id is: %d \n", record.id);

printf(" Name is: %s \n", record.name);

printf(" Percentage is: %f \n", record.percentage);

}

**OUTPUT:**

Id is: 1

Name is: Raju

Percentage is: 86.500000

**Recursive function:**

**A function which can be called by itself for repeated number of times, it is called as a recursive function.** Recursive functions are used to create data structures like lists, stacks, queues, etc. **Recursion is also called self-reference loop, then it is called recursion.**

**Advantages:**

1. It is a simple, easily understandable, concise, compact and transparent to view a c program.
2. Lesser number of programming statements required with the use of recursion.
3. It is useful in solving mathematical, trigonometric, logical games and algebraic problems.
4. It is eminently useful if the solution to problem is in recursive terms.
5. It is more useful in the multiprocessing and multitasking environment.
6. It is very useful in solving the data structure problems like linked list, queries, stack, tree, quick sort and merge sort etc.
7. Recursion saves the memory. In other words it utilizes the memory well.

**Disadvantages:**

1. It requires more memory to store the automatic variable to solve the problems like stack and so waste the memory space. i.e. to consume more storage space.
2. If properly recursion procedure is not checked, then it will create a problem for the processing and procedure run out of memory.
3. In some problems, it is a time consuming process and is not efficient.
4. It will create indefinite looping process, if condition not be applied at the proper place.

**Types of Recursion**

A function is recursive if it makes a call to itself directly or indirectly. If a function calls itself within from its own body, it is called recursive. Secondly, if a function f() calls another function g() that ultimately calls back to f(), then it can also be considered as recursion function. Following variants of recursion tell us making recursive calls in different ways depending upon the problem.

1. Linear Recursion
2. Tail Recursion
3. Binary Recursion
4. Multiple Recursion
5. **Linear Recursion:**

In linear recursion a function calls exactly once to itself each time the function is invoked, and grows linearly in proportion to the size of the problem. Finding maximum among an array of integers could be a good example of linear recursion.

1. **Tail Recursion:**

Tail recursion is another form of linear recursion, where the function makes a recursive call as its very last operation. Note that, there is a difference between last operation and last statement. If you see at recursiveMax, then you will find that recursive call to function recursiveMax is last statement but not last operation, code arr[n-1] yet to be processed to complete the execution of last statement.

1. **Binary Recursion:**

As name suggests, in binary recursion a function makes two recursive calls to itself when invoked, it uses binary recursion. Fibonacci series is a very nice example to demonstrate binary recursion.

1. **Multiple Recursion**

Multiple recursion can be treated a generalized form of binary recursion. When a function makes multiple recursive calls possibly more than two, it is called multiple recursion

**/\* Eg: recursive function to find factorial of given number \*/**

main()

{ int n,x;

clrscr();

printf("\n Enter N value");

scanf("%d",&n);

x=factorial(n);

printf("\n Factorial = %d",x);

}

factorial(int n)

{ int s=1;

if(n==0)

s=1;

else

s=n\*factorial(n-1);

return s; }

**UNIT – V**

**Pointer**

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte

**Advantages of Pointers:**

1. Pointers increase the execution speed of the c-program and are more efficient.
2. Pointer reduces the length and complexity of the program
3. Pointer accesses the memory elements very easily.
4. Pointers are more efficient in handling the data table i.e. two dimensional way. In other words, multidimensional arrays are easier to handle and represent with the pointers.
5. Use of the pointer to the character array to save the storage space in the memory.
6. Pointers have direct link with structure and union. By linking with structure these can increase the execution speed of structured programming.
7. Pointer helps to return more than one value from the function.
8. By using pointers, functions can be passed to the functions.
9. Pointers are also very useful to handle files.
10. Pointers are very useful in data structure manipulation.
11. Pointers can also be used to allocate and deallocate dynamic variables and arrays in memory.

**Disadvantages of Pointers:**

1. **Memory leak:**

Memory leakage occurs when memory is allocated but not released when it is no longer required. Memory leakage may result in slowing down the application or crashing the application when the computer memory resource limits are reached.

1. **Dangling pointer:**

Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer. As a result, the pointer still points to the memory location of the de-allocated memory.

**Eg: char \*p1;**

**Char \*p2=(\*char)malloc(sizeof(char));**

Now p1 becomes a dangling pointer. A solution to the above is to assign 0 (NULL) to p1 immediately before exiting the block in which it is declared.

1. **Using un-initialized memory:** An un-initialized memory contains garbage value.
2. **Using un-owned memory:**

A common programming mistake is to use pointers for accessing and modifying memory that is not owned by the program.

1. **Faulty de-allocation of memory:**

Memory leaks and freeing un-allocated memory can also result in memory corruption.

**Pointer operators**

1. **\* : Indirection Operator / Value at address operator**
2. **&: Address operator**

**Declaring a pointer**

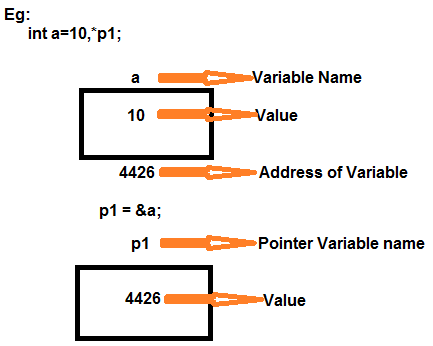
The pointer in c language can be declared using \* (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

**Syn: datatype \*var\_name;**

**Eg: int \*p1;**

**float \*p2;**

**char \*p3;**

****

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

**Exercise**

**/\* Program to display the addresses of variables \*/**

#include<stdio.h>

#include<conio.h>

main()

{ int a,\*p1;

char b,\*p2;

float c,\*p3;

clrscr();

p1=&a;

p2=&b;

p3=&c;

printf("\n Address of A = %u",p1);

printf("\n Address of B = %u",p2);

printf("\n Address of C = %u",p3);

getch();

}

**/\* Program to display the addresses of variables and their contents \*/**

**#include<stdio.h>**

**#include<conio.h>**

**main()**

**{** int a=10,\*p1;

char b='A',\*p2;

float c=12.455,\*p3;

clrscr();

p1=&a;

p2=&b;

p3=&c;

printf("\n Address of A = %u",p1);

printf("\n Address of B = %u",p2);

printf("\n Address of C = %u",p3);

printf("\n value of A =%d",\*p1);

printf("\n value of B =%c",\*p2);

printf("\n Value of C =%f",\*p3);

getch();

}

# Pointer Arithmetic’s:

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

1. Increment
2. Decrement
3. Addition
4. Subtraction

## eg-1:

## int a,\*p1;

## p1 = &a

## = 100 + 1 => 102

## = 100 + 2 => 104

## = 100 - 1 => 98

## eg-2: float x,\*p2;

## p2= &x;

## = 200 + 1 => 204

## = 200 + 2 => 208

## = 200 - 1 => 196

## eg -3: char c,\*p3;

## p3= &c;

## = 300 + 1=>301

## = 300 + 2=>302

## = 300 - 1=>299

## Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

**The Rule to increment the pointer is given below:**

**new\_address= current\_address + i \* size\_of(data type)**

**Where i is the number by which the pointer get increased.**

### 32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

### 64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

**/\***

**Eg program to display the address of variable before incrementation and after incrementation** \*/

#include<stdio.h>

main()

{ int n=50,\*p;

clrscr();

p=&n;

printf("Address of p variable is %u \n",p);

p=p+1;

printf("After increment: Address of p variable is %u \n",p);

getch();

}

**Output**

Address of p variable is 3214864300

After increment: Address of p variable is 3214864302

### Traversing an array by using pointer

#include<stdio.h>

#include<conio.h>

main()

{

int a[100],n,i,\*p,j,t;

clrscr();

printf("\n Enter N value");

scanf("%d",&n);

printf("\n Enter %d values in array",n);

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

scanf("%d",&a[i]);

p=&a[0];

printf("\n Postion \t Value \t Address");

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

{

printf("\n a[%d] \t\t %d \t %u",i,\*p,p);

p++;

}

getch();

}

**Output:**

Enter N value

5

Enter 5 values in array

10

20

30

40

50

**position value address**

a[0] 10 100

a[1] 20 102

a[2] 30 104

a[3] 40 106

a[4] 50 108

**Pointer using structure**

A collection of values of different types which can store under one variable or heading, it is called as a structure. The members of structure can be accessed by defining either simple structure variable or pointer structure variable. If we want to access the members of structure with pointer structure variable we should also make use of "->" symbol

**syn:**

struct struct\_name

{

member1;

member2;

member3;

member-n;

};

struct struct\_name \*var;

var->member1=value (\*var).member1=value

var->member2=value (\*var).member2=value

var->member3=value or (\*var).member3=value

var->member-n=value (\*var).member-n=value

**5Q) Write about Call by reference?**

**Ans:** The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types

**Eg:**

#include<stdio.h>

#include<conio.h>

main()

{

int a = 100,b = 200;

clrscr();

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

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

swap(&a, &b);

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

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

getch();

}

swap(int \*x, int \*y)

{

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

**6Q) Write about Dynamic memory allocation?**

**Ans:**

As pointer variable are used to hold the address of a variable, which is declared with the pointer variable during the variable declaration statement. So assigning of address of variables to a pointer variable is one of the legal way of the initialization of pointer. This process is always dynamic.

In the dynamic memory allocation of the memory, the declaration of variable is no longer required. The job of memory allocation can be done by using the three functions. These are **malloc(), calloc() and realloc().**

1. **malloc():**

* The malloc() function allocates single block of requested memory.
* It doesn't initialize memory at execution time, so it has garbage value initially.
* It returns NULL if memory is not sufficient.

**Syn:** void \*malloc(unsigned size)

**Eg:**

#include<stdio.h>

#include<stdlib.h>

main()

{

int n,i,\*ptr,sum=0;

clrscr();

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)malloc(n\*sizeof(int));

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

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

free(ptr);

}

**Output:**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

1. **Calloc():**

* The calloc() function allocates multiple block of requested memory.
* It initially initialize all bytes to zero.
* It returns NULL if memory is not sufficient.

**Syn:** void \*calloc(xyz, unsigned size)

**Eg:**

#include<stdio.h>

#include<stdlib.h>

main()

{

int n,i,\*ptr,sum=0;

clrscr();

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int\*)calloc(n,sizeof(int)); //memory allocated using calloc

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=\*(ptr+i);

}

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

free(ptr);

getch();

}

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

1. **Realloc():**

It adjusts the previously allocated block to the size bytes. It will allocate the increasing or decreasing the block to the new size. Here block refers to the original block already allocated by malloc() or calloc() function. It returns the address of the new reallocated block. Note that if it will unable to reallocate the block, then it will display and return the NULL value.

**Syn:** void \*recalloc(xyz, unsigned size)

**FILES**

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level calls to handle file on your storage devices. There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are

1. **Text Files.**

A Text file contains only the text information like alphabets, digits and special symbols. The ASCII code of these characters are stored in these files. It uses only 7 bits allowing 8 bit to be zero.

1. **Binary Files**

A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can’t be recognized by a word processor.

Hence depending up on **the method of accessing the data stored**, there are two types of files.

1. Sequential file
2. Random access file

**Sequential File:**

In this type of files data is kept in sequential order. In these files the data can read or write only from the beginning of a file.

**Random access Files:**

In this type of files data can be read and modified randomly. If we want to read the last record we can read it directly. It takes less time when compared to sequential file.

**File Operation Functions:**

To perform all the different file operation the following functions should be used. They are -

1. Charater i/o
2. Formatted i/o
3. Unformatted i/o

**Character i/o functions**

The character i/o functions are used to write characters in to a file as well as they can read from the file. The character i/o functions are

1. **fputc():** This function is used to write characters in to a file.

**Syn:**fputc(var,fptr);

1. **fgetc()**: This function is used to read characters from a file.

**Syn:** var=fgetc(fptr);

**Formatted i/o functions**

The formatted i/o functions are used to write data elements into a file as well as they can read. The formatted i/o functions are –

* + 1. **fprintf()** - It is used to write data elements into a file in a specified format.

**Syn:** fprintf(fprt,"format controls",var1,var2,...);

* + 1. **fscanf()** – It is used to read data elements from a file.

**Syn:** fscanf(fptr,"format controls",&var1,var2,....)

**Unformatted I/O Functions**

* + - 1. **fwrite() -** It is a unformatted function. It is used to write structure of data into a file.

**Syn:** fwrite(&struct\_var,sizeof(struct\_var),1,fptr)

* + - 1. **fread() -** It is a unformatted function. It is used read data structure of data from a file.

**Syn:** fread(&struct\_var,sizeof(struct\_var),1,fptr)

**Steps to create a file**

1. Define a file pointer

FILE \*fptr;

2. Open a file by giving file name and specify the purpose of a file.

fptr=fopen("filename","modes")

|  |  |
| --- | --- |
| **Text modes** | **Description** |
| **W** | used to create a new file for writing purpose |
| **R** | used to open a existed file for reading purpose. |
| **A** | Used to open a existed file and appends the data into a file. |
| **W+** | Used for both writing and reading |
| **R+** | Used for both reading and writing |
| **A+** | Used to open an already existed file for appending purpose. If the file is not existed it creates a new file and writes the data from the beginning of file. |

3. Peform the required operations in a file.

4. Close a file

fclose(fptr);

**Random access files**

In these files the data can read or write either from the beginning of a file or from the middle of a file. To perform Random access file operations the following functions should be used.

1. rewind()
2. ftell()
3. fseek()

* **rewind():** This function is used to move the record pointer to the beginning of a file

**Syn:** rewind(fptr)

* **ftell():** This function tells the position of a record pointer in terms of long int.

**Syn:** Var=ftell(fptr)

* **fseek():** This function is used to move a record pointer to a desired location in a file.

**Syn: fseek(fptr,offset,whence)**

Offset means moving a record pointer to specified memory location and whence indicates from where the data should read or write.

**WHENCE OPTIONS:**

**Keyword value description**

SEEK\_SET 0 Used to read/write from the beginning of a file

SEEK\_CUR 1 Used to read/write from the current position of a File

SEEK\_END 2 Used to read/write from the end of a file.

**Error Handling During I/O Operation**

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow means no space in the disk for storing data.
3. Trying to use a file that has not been opened.
4. Trying to perform operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Try to read a file with no data in it.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status library functions;

1. **feof**
2. **ferror**

### feof

The **feof**function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all the data from the specified file has been read, and returns zero otherwise. If **fp**is a pointer to file that has just been opened for reading, then the statement

**if ( feof ( fp ) )**

**printf ( “End of data.\n” ) ;**

would display the message “End of data.” on reaching the end of file condition.

### ferror

The **ferror**function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

**if ( ferror ( fp ) ! = 0 )**

**print ( “An error has occurred.\n” ) ;**

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen**function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. **Example:**

**if ( fp == NULL )**

**printf (“ File could not be opened.\n”);**

**Command line arguments**

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

To support command line argument, you need to change the structure of main() function as given below

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

Here, **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

* **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name).The value of argc should be non negative.
* **argv(ARGument Vector)** is array of character pointers listing all the arguments.
* If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
* Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

**Example**

#include <stdio.h>

void main(int argc, char \*argv[] )

{

clrscr();

printf("Program name is: %s\n", argv[0]);

if(argc < 2)

{

printf("No argument passed through command line.\n");

}

else

{

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

}

}

Run this program as follows: program.exe hello

**Output:**

Program name is: program

First argument is: hello