SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF MECHANICAL ENGINEERING

DEPARTMENT OF AUTOMOBILE, AERONUTICAL, MECHATRONICS AND MECHANICAL ENGINEERING

UNIT - I

Programming in C - SCSA1103

INTRODUCTION OF C

Content

Introduction: Generation and Classification of Computers- Basic Organization of a Computer Algorithms & flowcharts - Overview of C - Features of C - Structure of C program - Compilation & execution of C program - Identifiers, variables, expression, keywords, data types, constants, scope and life of variables, and local and global variables – Operators: arithmetic, logical, relational, conditional and bitwise operators– Special operators: size of () & comma (,) operator – Precedence and associativity of operators & Type conversion in expressions – Input and output statements- solving simple scientific and statistical problems

HISTORY OF COMPUTER

- Until the development of the first generation computers based on vacuum tubes, there had been several developments in the computing technology related to the mechanical computing devices. The key developments that took place till the first computer was developed are as follows— Calculating Machines ABACUS was the first mechanical calculating device for counting of large numbers. The word ABACUS means calculating board. It consists of bars in horizontal positions on which sets of beads are inserted. The horizontal bars have 10 beads each, representing units, tens, hundreds, etc. An abacus is shown in Figure 1.1

Fig. 1.1 Abacus

- Napier’s Bones was a mechanical device built for the purpose of multiplication in 1617 ad. by an English mathematician John Napier.
- Slide Rule was developed by an English mathematician Edmund Gunter in the 16th century. Using the slide rule, one could perform operations like addition, subtraction, multiplication and division. It was used extensively till late 1970s. Figure 1.2 shows a slide rule.

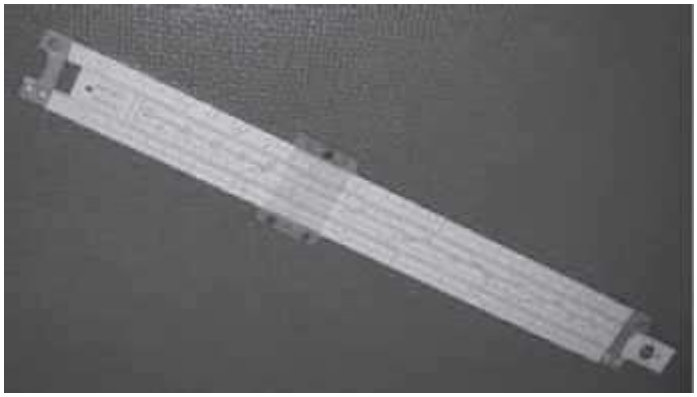


Fig. 1.2 Slide rule

- Pascal's Adding and Subtraction Machine was developed by Blaise Pascal. It could add and subtract. The machine consisted of wheels, gears and cylinders.
- Leibniz's Multiplication and Dividing Machine was a mechanical device that could both multiply and divide. The German philosopher and mathematician Gottfried Leibniz built it around 1673.
- Punch Card System was developed by Jacquard to control the power loom in 1801. He invented the punched card reader that could recognize the presence of hole in the punched card as binary one and the absence of the hole as binary zero. The Os and 1s are the basis of the modern digital computer. A punched card is shown in Figure 1.3.

Fig. 1.3 Punched card

- Babbage's Analytical Engine An English man Charles Babbage built a mechanical machine to do complex mathematical calculations, in the year 1823. The machine was called as difference engine. Later, Charles Babbage and Lady Ada Lovelace developed a general-purpose calculating machine, the analytical engine. Charles Babbage is also called the father of computer.
- Hollerith's Punched Card Tabulating Machine was invented by Herman Hollerith. The machine could read the information from a punched card and process it electronically.

The developments discussed above and several others not discussed here, resulted in the development of the first computer in the 1940s.

GENERATIONS OF COMPUTER

The computer has evolved from a large—sized simple calculating machine to a smaller but much more powerful machine. The evolution of computer to the current state is defined in terms of the generations of computer. Each generation of computer is designed based on a new technological development, resulting in better, cheaper and smaller computers that are more powerful, faster and efficient than their predecessors. Currently, there are five generations of computer. In the following subsections, we will discuss the generations of computer in terms of—

1. the technology used by them (hardware and software),
2. computing characteristics (speed, i.e., number of instructions executed per second),
3. physical appearance, and
4. their applications.

First Generation (1940 to 1956): Using Vacuum Tubes

- Hardware Technology The first generation of computers used vacuum tubes (Figure 1.4) for circuitry and magnetic drums for memory. The input to the computer was through punched cards and paper tapes. The output was displayed as printouts.

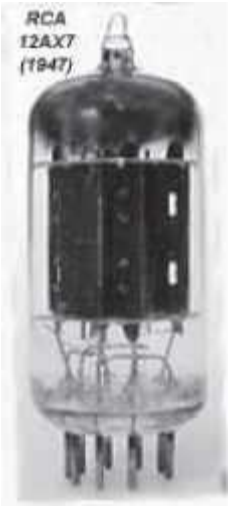


Fig. 1.4 Vacuum tube

- Software Technology The instructions were written in machine language. Machine language uses 0s and 1s for coding of the instructions. The first generation computers could solve one problem at a time.
- Computing Characteristics The computation time was in milliseconds.
- Physical Appearance These computers were enormous in size and required a large room for installation.
- Application They were used for scientific applications as they were the fastest computing device of their time.
- Examples UNIVersal Automatic Computer (UNIVAC), Electronic Numerical Integrator And Calculator (ENIAC), and Electronic Discrete Variable Automatic Computer (EDVAC).

  The first generation computers used a large number of vacuum tubes and thus generated a lot of heat. They consumed a great deal of electricity and were expensive to operate. The machines were prone to frequent malfunctioning and required constant maintenance. Since first generation computers used machine language, they were difficult to program.

Second Generation (1956 to 1963): Using Transistors

Fig. 1.5 Transistors

- Hardware Technology Transistors (Figure 1.5) replaced the vacuum tubes of the first generation of computers. Transistors allowed computers to become smaller, faster, cheaper, energy efficient and reliable. The second generation computers used magnetic core technology for primary memory. They used magnetic tapes and magnetic disks for secondary storage. The input was still through punched cards and the output using printouts. They used the concept of a stored program, where instructions were stored in the memory of computer.

- Software Technology The instructions were written using the assembly language. Assembly language uses mnemonics like ADD for addition and SUB for subtraction for coding of the instructions. It is easier to write instructions in assembly language, as compared to writing instructions in machine language. High-level programming languages, such as early versions of COBOL and FORTRAN were also developed during this period.
- Computing Characteristics The computation time was in microseconds.
- Physical Appearance Transistors are smaller in size compared to vacuum tubes, thus, the size of the computer was also reduced.
- Application The cost of commercial production of these computers was very high, though less than the first generation computers. The transistors had to be assembled manually in second generation computers.
- Examples PDP-8, IBM 1401 and CDC 1604.

  Second generation computers generated a lot of heat but much less than the first generation computers. They required less maintenance than the first generation computers.

Third Generation (1964 to 1971): Using Integrated Circuits

- Hardware Technology The third generation computers used the Integrated Circuit (IC) chips. Figure 1.6 shows IC chips. In an IC chip, multiple transistors are placed on a silicon chip. Silicon is a type of semiconductor. The use of IC chip increased the speed and the efficiency of computer, manifold. The keyboard and monitor were used to interact with the third generation computer, instead of the punched card and printouts.

Fig. 1.6 IC chips

- Software Technology The keyboard and the monitor were interfaced through the operating system. Operating system allowed different applications to run at the same time. High-level languages were used extensively for programming, instead of machine language and assembly language.

- Computing Characteristics The computation time was in nanoseconds.
- Physical Appearance The size of these computers was quite small compared to the second generation computers.
- Application Computers became accessible to mass audience. Computers were produced commercially, and were smaller and cheaper than their predecessors.
- Examples IBM 370, PDP 11.

  The third generation computers used less power and generated less heat than the second generation computers. The cost of the computer reduced significantly, as individual components of the computer were not required to be assembled manually. The maintenance cost of the computers was also less compared to their predecessors.

Fourth Generation (1971 to present): Using Microprocessors

- Hardware Technology They use the Large Scale Integration (LSI) and the Very Large Scale Integration (VLSI) technology. Thousands of transistors are integrated on a small silicon chip using LSI technology. VLSI allows hundreds of thousands of components to be integrated in a small chip. This era is marked by the development of microprocessor. Microprocessor is a chip containing millions of transistors and components, and, designed using LSI and VLSI technology. A microprocessor chip is shown in Figure 1.7. This generation of computers gave rise to Personal Computer (PC). Semiconductor memory replaced the earlier magnetic core memory, resulting in fast random access to memory. Secondary storage device like magnetic disks became smaller in physical size and larger in capacity. The linking of computers is another key development of this era. The computers were linked to form networks that led to the emergence of the Internet.

  This generation also saw the development of pointing devices like mouse, and handheld devices.

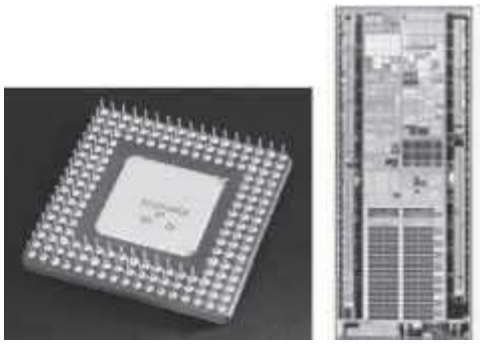


Fig. 1.7 Microprocessors

- Software Technology Several new operating systems like the MS-DOS and MS- Windows developed during this time. This generation of computers supported Graphical User Interface (GUI). GUI is a user-friendly interface that allows user to interact with the computer via menus and icons. High-level programming languages are used for the writing of programs.
- Computing Characteristics The computation time is in picoseconds.
- Physical Appearance They are smaller than the computers of the previous generation. Some can even fit into the palm of the hand.
- Application They became widely available for commercial purposes. Personalcomputers became available to the home user.
- Examples The Intel 4004 chip was the first microprocessor. The components of the computer like Central Processing Unit (CPU) and memory were located on a single chip. In 1981, IBM introduced the first computer for home use. In 1984, Apple introduced the Macintosh.

  The microprocessor has resulted in the fourth generation computers being smaller and cheaper than their predecessors. The fourth generation computers are also portable and more reliable. They generate much lesser heat and require less maintenance compared to their predecessors. GUI and pointing devices facilitate easy use and learning on the computer. Networking has resulted in resource sharing and communication among different computers.

Fifth Generation (Present and Next): Using Artificial Intelligence

The goal of fifth generation computing is to develop computers that are capable of learning and self-organization. The fifth generation computers use Super Large Scale Integrated (SLSI) chips that are able to store millions of components on a single chip. These computers have large memory requirements. This generation of computers uses parallel processing that allows several instructions to be executed in parallel, instead of serial execution. Parallel processing results in faster processing speed. The Intel dualcore microprocessor uses parallel processing.

The fifth generation computers are based on Artificial Intelligence (AI). They try to simulate the human way of thinking and reasoning. Artificial Intelligence includes areas like Expert System (ES), Natural Language Processing (NLP), speech recognition, voice recognition, robotics, etc.

CLASSIFICATION OF COMPUTER

The digital computers that are available nowadays vary in their sizes and types. The computers are broadly classified into four categories (Figure 1.8) based on their size and type—(1) Microcomputers, (2) Minicomputers, (3) Mainframe computers, and (4) Supercomputer.

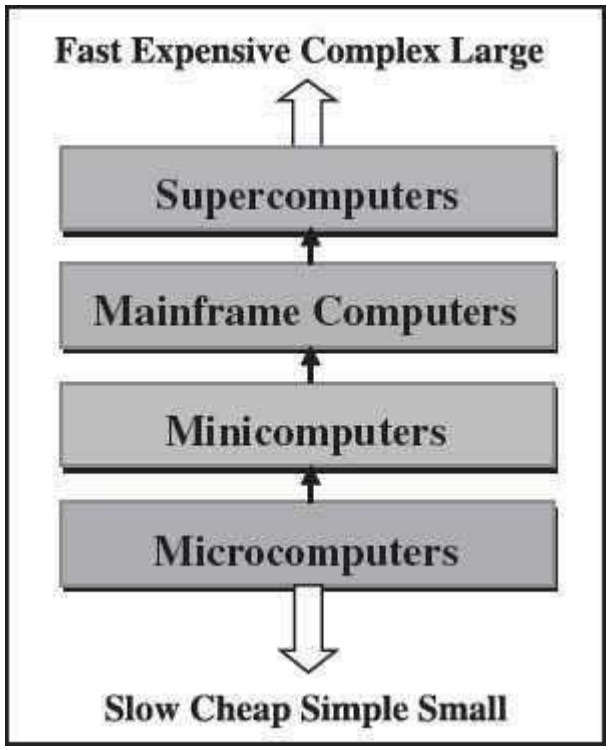


Fig.1.8 Classification of computers based on size and type

Microcomputers

Microcomputers are small, low-cost and single-user digital computer. They consist of CPU, input unit, output unit, storage unit and the software. Although microcomputers are stand-alone machines, they can be connected together to create a network of computers that can serve more than one user. IBM PC based on Pentium microprocessor and Apple Macintosh are some

examples of microcomputers. Microcomputers include desktop computers, notebook computers or laptop, tablet computer, handheld computer, smart phones and netbook, as shown in Figure 1.9.

Fig. 1.9 Microcomputers

- Desktop Computer or Personal Computer (PC) is the most common type of microcomputer. It is a stand-alone machine that can be placed on the desk. Externally, it consists of three units—keyboard, monitor, and a system unit containing the CPU, memory, hard disk drive, etc. It is not very expensive and is suited to the needs of a single user at home, small business units, and organizations. Apple, Microsoft, HP, Dell and Lenovo are some of the PC manufacturers.
- Notebook Computers or Laptop resemble a notebook. They are portable and have all the features of a desktop computer. The advantage of the laptop is that it is small in size (can be put inside a briefcase), can be carried anywhere, has a battery backup and has all the functionality of the desktop. Laptops can be placed on the lap while working (hence the name). Laptops are costlier than the desktop machines.
- Netbook These are smaller notebooks optimized for low weight and low cost, and are designed for accessing web-based applications. Starting with the earliest netbook in late 2007, they have gained significant popularity now. Netbooks deliver the

performance needed to enjoy popular activities like streaming videos or music, emailing, Web surfing or instant messaging. The word netbook was created as a blend of Internet and notebook.

- Tablet Computer has features of the notebook computer but it can accept input from a stylus or a pen instead of the keyboard or mouse. It is a portable computer. Tablet computer are the new kind of PCs.
- Handheld Computer or Personal Digital Assistant (PDA) is a small computer that can be held on the top of the palm. It is small in size. Instead of the keyboard, PDA uses a pen or a stylus for input. PDAs do not have a disk drive. They have a limited memory and are less powerful. PDAs can be connected to the Internet via a wireless connection. Casio and Apple are some of the manufacturers of PDA. Over the last few years, PDAs have merged into mobile phones to create smart phones.
- Smart Phones are cellular phones that function both as a phone and as a small PC. They may use a stylus or a pen, or may have a small keyboard. They can be connected to the Internet wirelessly. They are used to access the electronic-mail, download music, play games, etc. Blackberry, Apple, HTC, Nokia and LG are some of the manufacturers of smart phones.

Minicomputers

Minicomputers (Figure 1.10) are digital computers, generally used in multi-user systems. They have high processing speed and high storage capacity than the microcomputers. Minicomputers can support 4–200 users simultaneously. The users can access the minicomputer through their PCs or terminal. They are used for real-time applications in industries, research centers, etc. PDP 11, IBM (8000 series) are some of the widely used minicomputers.

Fig. 1.10 Minicomputer

Mainframe Computers

Mainframe computers (Figure 1.11) are multi-user, multi-programming and high performance computers. They operate at a very high speed, have very large storage capacity and can handle the workload of many users. Mainframe computers are large and powerful systems generally used in centralized databases. The user accesses the mainframe computer via a terminal that may be a dumb terminal, an intelligent terminal or a PC. A dumb terminal cannot store data or do processing of its own. It has the input and output device only. An intelligent terminal has the input and output device, can do processing, but, cannot store data of its own. The dumb and the intelligent terminal use the processing power and the storage facility of the mainframe computer. Mainframe computers are used in organizations like banks or companies, where many people require frequent access to the same data. Some examples of mainframes are CDC 6600 and IBM ES000 series.

Fig. 1.11 Mainframe computer

Supercomputers

Supercomputers (Figure 1.12) are the fastest and the most expensive machines. They have high processing speed compared to other computers. The speed of a supercomputer is generally measured in FLOPS (Floating point Operations Per Second). Some of the faster supercomputers can perform trillions of calculations per second. Supercomputers are built by interconnecting thousands of processors that can work in parallel.

Supercomputers are used for highly calculation-intensive tasks, such as, weather forecasting, climate research (global warming), molecular research, biological research, nuclear research and aircraft design. They are also used in major universities, military agencies and scientific research laboratories. Some examples of supercomputers are IBM Roadrunner, IBM Blue gene and Intel ASCI red. PARAM is a series of supercomputer assembled in India by C-DAC (Center for Development of Advanced Computing), in Pune. PARAM Padma is the latest machine in this series. The peak computing power of PARAM Padma is 1 Tera FLOP (TFLOP).

Fig. 1.12 Supercomputer

THE COMPUTER SYSTEM

Computer is an electronic device that accepts data as input, processes the input data by performing mathematical and logical operations on it, and gives the desired output. The computer system consists of four parts•(1) Hardware, (2) Software, (3) Data, and (4) Users. The parts of computer system are shown in Figure 1.13.

Hardware consists of the mechanical parts that make up the computer as a machine. The hardware consists of physical devices of the computer. The devices are required for input, output, storage and processing of the data. Keyboard, monitor, hard disk drive, floppy disk drive, printer, processor and motherboard are some of the hardware devices.

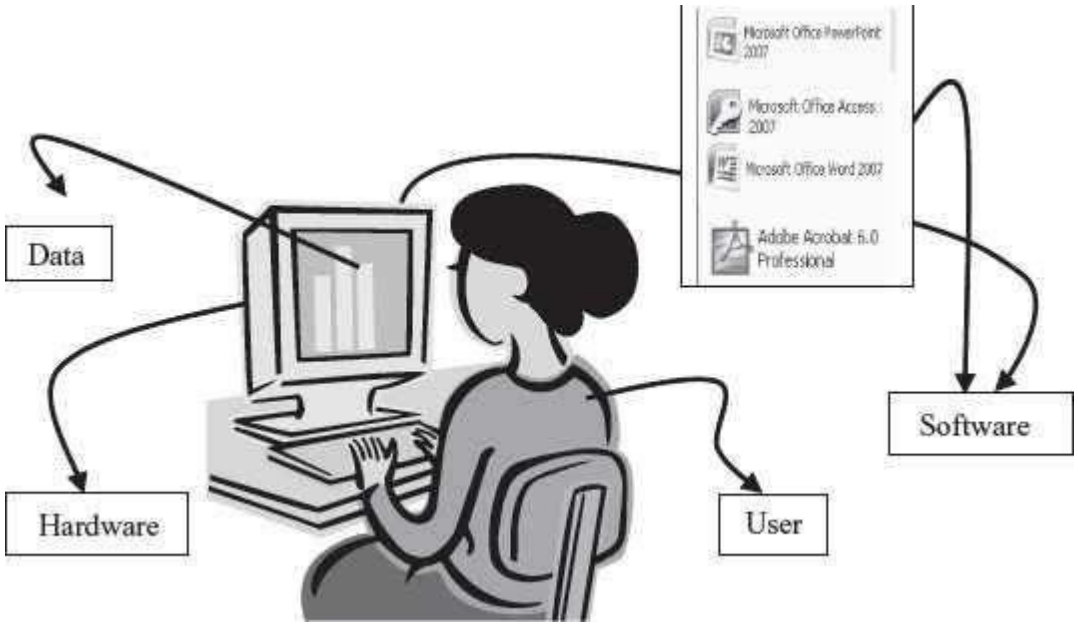


Fig. 1.13 Parts of computer system

Software is a set of instructions that tells the computer about the tasks to be performed and how these tasks are to be performed. Program is a set of instructions, written in a language understood by the computer, to perform a specific task. A set of programs and documents are collectively called software. The hardware of the computer system cannot perform any task on its own. The hardware needs to be instructed about the task to be performed. Software instructs the computer about the task to be performed. The hardware carries out these tasks. Different software can be loaded on the same hardware to perform different kinds of tasks.

Data are isolated values or raw facts, which by themselves have no much significance. For example, the data like 29, January, and 1994 just represent values. The data is provided as input to the computer, which is processed to generate some meaningful information. For example, 29, January and 1994 are processed by the computer to give the date of birth of a person.

Users are people who write computer programs or interact with the computer. They are also known as skinware, liveware, humanware or peopleware. Programmers, data entry operators, system analyst and computer hardware engineers fall into this category.

The Input-Process-Output Concept

A computer is an electronic device that (1) accepts data, (2) processes data, (3) generates output, and (4) stores data. The concept of generating output information from the input 4 data is also referred to as input-process-output concept.

The input-process-output concept of the computer is explained as follows—

- Input The computer accepts input data from the user via an input device like keyboard. The input data can be characters, word, text, sound, images, document, etc.
- Process The computer processes the input data. For this, it performs some actions on the data by using the instructions or program given by the user of the data. The action could be an arithmetic or logic calculation, editing, modifying a document, etc. During processing, the data, instructions and the output are stored temporarily in the computer's main memory.
- Output The output is the result generated after the processing of data. The output may be in the form of text, sound, image, document, etc. The computer may display the output on a monitor, send output to the printer for printing, play the output, etc.
- Storage The input data, instructions and output are stored permanently in the secondary storage devices like disk or tape. The stored data can be retrieved later, whenever needed.

Components of Computer Hardware

The computer system hardware comprises of three main components —

1. Input/Output (I/O) Unit,
2. Central Processing Unit (CPU), and
3. Memory Unit.

The I/O unit consists of the input unit and the output unit. CPU performs calculations and processing on the input data, to generate the output. The memory unit is used to store the data, the instructions and the output information. Figure 1.14 illustrates the typical interaction among the different components of the computer.

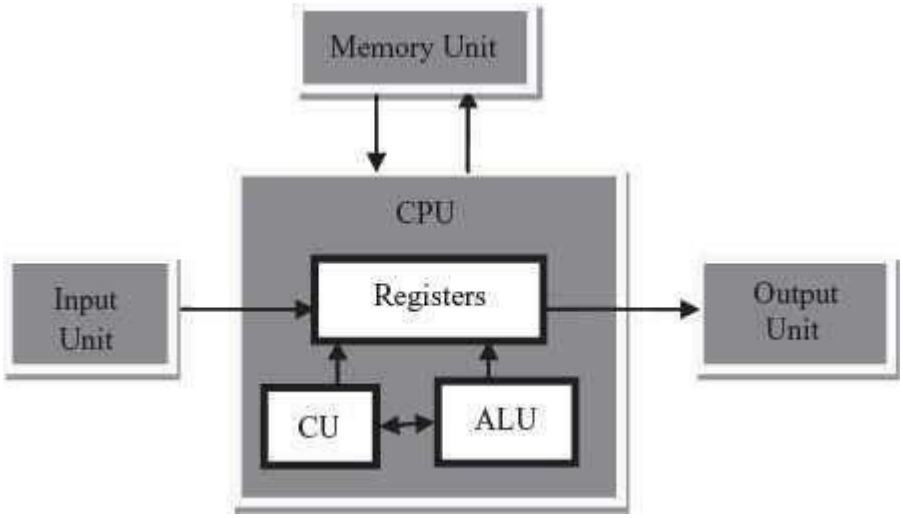


Fig. 1.14 The computer system interaction

- Input/Output Unit The user interacts with the computer via the I/O unit. The Input unit accepts data from the user and the Output unit provides the processed data i.e. the information to the user. The Input unit converts the data that it accepts from the user, into a form that is understandable by the computer. Similarly, the Output unit provides the output in a form that is understandable by the user. The input is provided to the computer using input devices like keyboard, trackball and mouse. Some of the commonly used output devices are monitor and printer.
- Central Processing Unit CPU controls, coordinates and supervises the operations ofthe computer. It is responsible for processing of the input data. CPU consists of Arithmetic Logic Unit (ALU) and Control Unit (CU).

- ALU performs all the arithmetic and logic operations on the input data.
- CU controls the overall operations of the computer i.e. it checks the sequence of execution of instructions, and, controls and coordinates the overall functioning of the units of computer.

Additionally, CPU also has a set of registers for temporary storage of data, instructions, addresses and intermediate results of calculation.

- Memory Unit Memory unit stores the data, instructions, intermediate results and output, temporarily, during the processing of data. This memory is also called the main memory or primary memory of the computer. The input data that is to be processed is brought into the main memory before processing. The instructions required for processing of data and any intermediate results are also stored in the main memory. The output is stored in memory before being transferred to the output device. CPU can work with the information stored in the main memory. Another kind of storage unit is also referred toas the secondary memory of the computer. The data, the programs and the output are stored permanently in the storage unit of the computer. Magnetic disks, optical disks and magnetic tapes are examples of secondary memory.

APPLICATION OF COMPUTERS

Computers have proliferated into various areas of our lives. For a user, computer is a tool that provides the desired information, whenever needed. You may use computer to get information about the reservation of tickets (railways, airplanes and cinema halls), books in a library, medical history of a person, a place in a map, or the dictionary meaning of a word. The information may be presented to you in the form of text, images, video clips, etc.

Figure 1.15 shows some of the applications of computer. Some of the application areas of the computer are listed below—

- Education Computers are extensively used, as a tool and as an aid, for imparting education. Educators use computers to prepare notes and presentations of their lectures. Computers are used to develop computer-based training packages, to provide distance education using the e-learning software, and to conduct online examinations. Researchers use computers to get easy access to conference and journal details and to get global access to the research material.

- Entertainment Computers have had a major impact on the entertainment industry. The user can download and view movies, play games, chat, book tickets for cinema halls, use multimedia for making movies, incorporate visual and sound effects using computers, etc. The users can also listen to music, download and share music, create music using computers, etc.
- Sports A computer can be used to watch a game, view the scores, improve the game, play games (like chess, etc.) and create games. They are also used for the purposes of training players.
- Advertising Computer is a powerful advertising media. Advertisement can be displayed on different websites, electronic-mails can be sent and reviews of a product by different customers can be posted. Computers are also used to create an advertisement using the visual and the sound effects. For the advertisers, computer is a medium via which the advertisements can be viewed globally. Web advertising has become a significant factor in the marketing plans of almost all companies. In fact, the business model of Google is mainly dependent on web advertising for generating revenues.

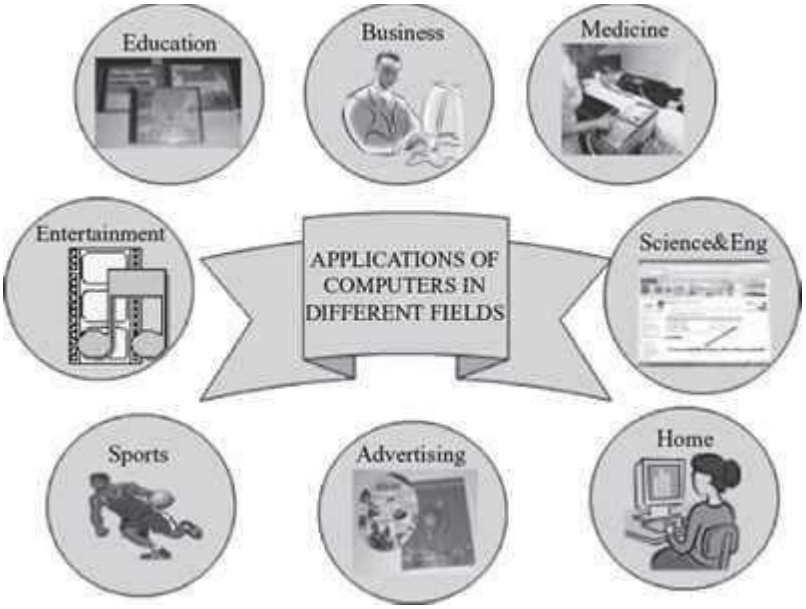


Fig.1.15 Applications of computer

- Medicine Medical researchers and practitioners use computers to access information about the advances in medical research or to take opinion of doctors globally. The medical history of patients is stored in the computers. Computers are also an integralpart of various kinds of sophisticated medical equipments like ultrasound machine, CAT scan machine, MRI scan machine, etc. Computers also provide assistance to the medical surgeons during critical surgery operations like laparoscopic operations, etc.

- Science and Engineering Scientists and engineers use computers for performing complex scientific calculations, for designing and making drawings (CAD/CAM applications) and also for simulating and testing the designs. Computers are used for storing the complex data, performing complex calculations and for visualizing 3–dimensional objects. Complex scientific applications like the launch of the rockets, space exploration, etc., are not possible without the computers.
- Government The government uses computers to manage its own operations and also for e-governance. The websites of the different government departments provide information to the users. Computers are used for the filing of income tax return, paying taxes, online submission of water and electricity bills, for the access of land record details, etc. The police department uses computers to search for criminals using fingerprint matching, etc.
- Home Computers have now become an integral part of home equipment. At home, people use computers to play games, to maintain the home accounts, for communicating with friends and relatives via Internet, for paying bills, for education and learning, etc. Microprocessors are embedded in house hold utilities like, washing machines, TVs, food processors, home theatres, security devices, etc.

The list of applications of computers is so long that it is not possible to discuss all of them here. In addition to the applications of the computers discussed above, computers have also proliferated into areas like banks, investments, stock trading, accounting, ticket reservation, military operations, meteorological predictions, social networking, business organizations, police department, video conferencing, telepresence, book publishing, web newspapers, and information sharing.

SUMMARY

- Computer is an electronic device which accepts data as input, performs processing on the data, and gives the desired output. A computer may be analog or digital computer.
- Speed, accuracy, diligence, storage capability and versatility are the main characteristics of computer.
- The computing devices have evolved from simple mechanical machines, like ABACUS, Napier’s bones, Slide Rule, Pascal’s Adding and Subtraction Machine, Leibniz’s Multiplication and Dividing Machine, Jacquard Punched Card System, Babbage’s Analytical Engine and Hollerith’s Tabulating Machine, to the first electronic computer.
- Charles Babbage is called the father of computer.

- The evolution of computers to their present state is divided into five generations of computers, based on the hardware and software they use, their physical appearance and their computing characteristics.
- First generation computers were vacuum tubes based machines. These were large in size, expensive to operate and instructions were written in machine language. Their computation time was in milliseconds.
- Second generation computers were transistor based machines. They used the stored program concept. Programs were written in assembly language. They were smaller in size, less expensive and required less maintenace than the first generation computers. The computation time was in microseconds.
- Third generation computers were characterized by the use of IC. They consumed less power and required low maintenance compared to their predecessors. High-level languages were used for programming. The computation time was in nanoseconds. These computers were produced commercially.
- Fourth generation computers used microprocessors which were designed using the LSI and VLSI technology. The computers became small, portable, reliable and cheap. The

  computation time is in picoseconds. They became available both to the home user and for commercial use.
- Fifth generation computers are capable of learning and self organization. These computers use SLSI chips and have large memory requirements. They use parallel processing and are based on AI. The fifth generation computers are still being developed.
- Computers are broadly classified as microcomputers, minicomputers, mainframe computers, and supercomputers, based on their sizes and types.
- Microcomputers are small, low-cost standalone machines. Microcomputers include desktop computers, notebook computers or laptop, netbooks, tablet computer, handheld computer and smart phones.
- Minicomputers are high processing speed machines having more storage capacity than the microcomputers. Minicomputers can support 4–200 users simultaneously.
- Mainframe computers are multi-user, multiprogramming and high performance computers. They have very high speed, very large storage capacity and can handle large workloads. Mainframe computers are generally used in centralized databases.
- Supercomputers are the most expensive machines, having high processing speed capable of performing trillions of calculations per second. The speed of a supercomputer is measured in FLOPS. Supercomputers find applications in computing-intensive tasks.
- Computer is an electronic device based on the input-process-output concept. Input/Output Unit, CPU and Memory unit are the three main components of computer.

- Input/Output Unit consists of the Input unit which accepts data from the user and the Output unit that provides the processed data. CPU processes the input data, and, controls, coordinates and supervises the operations of the computer. CPU consists of ALU, CU and Registers. The memory unit stores programs, data and output, temporarily, during the processing. Additionally, storage unit or secondary memory is used for the storing of programs, data and output permanently.
- Computers are used in various areas of our life. Education, entertainment, sports, advertising, medicine, science and engineering, government, office and home are some of the application areas of the computers.

  Algorithms & Flowcharts

  The sequence of steps to be performed in order to solve a problem by the computer is known as an Algorithm. The Algorithm often refers to the logic of a program. Algorithms can be expressed in many different notations, including natural languages, pseudocode, flowcharts and programming languages.

  Flowchart is a graphical or symbolic representation of an algorithm. It is the diagrammatic representation of the step-by-step solution to a given problem.

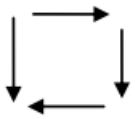
Algorithms & Flowcharts

The sequence of steps to be performed in order to solve a problem by the computer isknown as an Algorithm. The Algorithm often refers to the logic of a program. Algorithms can beexpressed in many different notations, including natural languages, pseudocode, flowcharts andprogramming languages.

Flowchart is a graphical or symbolic representation of an algorithm. It is thediagrammatic representation of the step-by-step solution to a given problem.

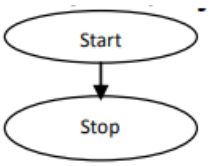
Flow Chart Symbols:

Flow Line Symbol:



- These are the left, right, top &bottom line connection symbols.
- These lines show the flow of control through the program.

Terminal Symbol:



- The oval shape symbol always begins and ends the flowchart.

- The Start symbol have only one flow line but not entering flow line.
- The stop symbol have an entering flow line but not exit flow line.

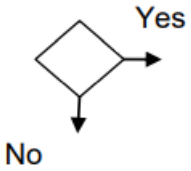
Input / Output Symbol:

- The parallelogram is used for both Input(read) and output(write) operations.

Process symbol:

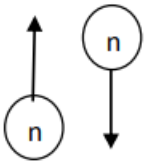
- The rectangle symbol is used primarily for calculations and initialization of memory location, all the arithmetic operations, data movements and initializations.

Decision Symbol:



- The diamond symbol is used in a flowchart to indicate the point at which a decision has to be made and a branch of two or more alternatives are possible
- There are always two exits from a decision symbol - one is labeled Yes or True and other labeled No or False.

Connector Symbol:



- A connector symbol is represented by a circle with a letter or digit inside to specify the link.
- Used if the flowchart is big and needs continuation in next page.

Let us take a small problem and see how can we write an algorithm using natural language and draw flowchart for the same.
Illustration Consider the problem of finding the largest number in a given set of three numbers.

Algorithm:

1. Get three numbers from the user
2. Compare the first two numbers
3. The larger of the first two numbers is compared with the third number
4. The larger number obtained as a result of the above execution is the largest number
5. Print the that number as output

Flowchart:

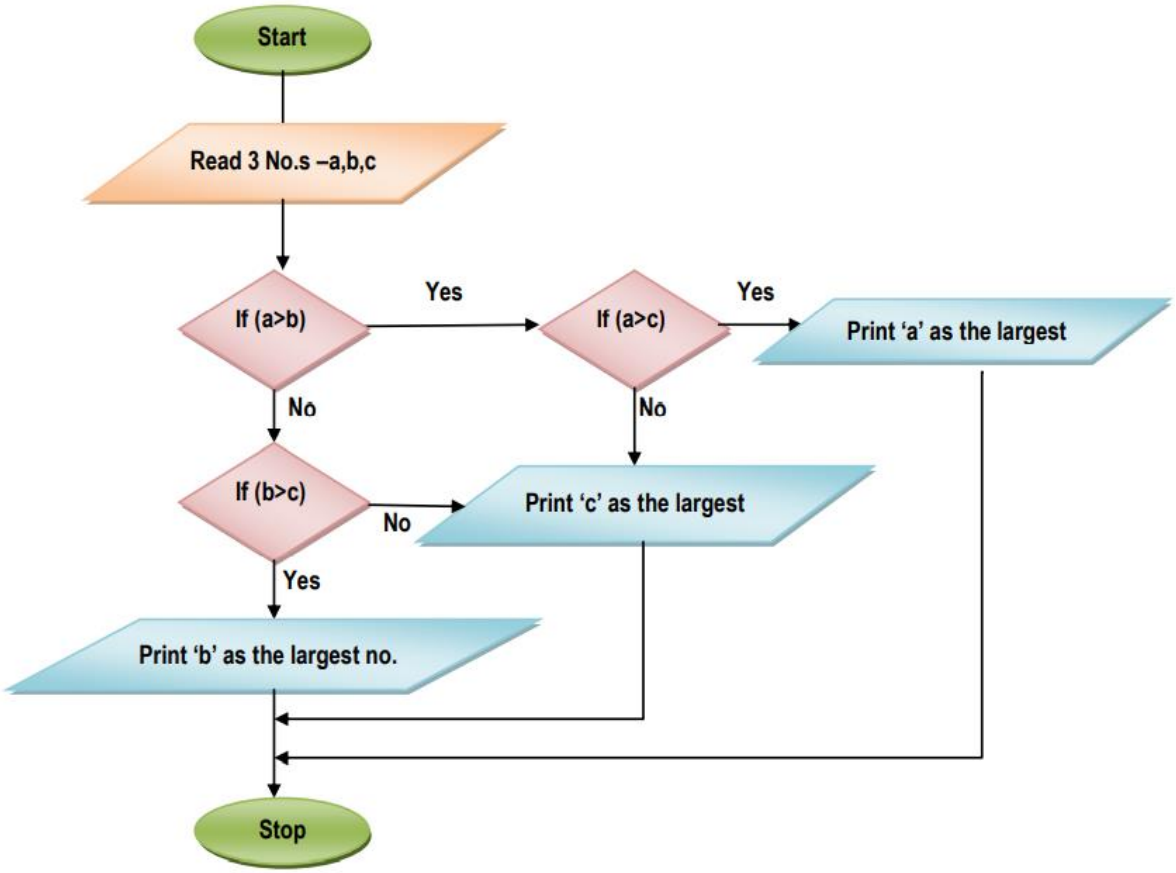


Fig.1.16. Flow chart for the program-finding the largest of 3 given nos

Overview of C

- C is a procedural programming language as well as a general-purpose programming language that was developed by Dennis Ritchie at AT&T's Bell laboratories in 1972.
- It is an amazing and simple language that helps us to develop complex software applications with ease.
- It is considered as the mother of all languages.

- C is a high-level programming language that provides support to a low-level programming language as well.

A brief history

- C is a programming language developed at "AT & T's Bell Laboratories" of USA in 1972.
- It was written by Dennis Ritchie.

Fig .1.17. Dennis Ritchie

- The programming language C was first given by Kernighan and Ritchie, in a classic book called "The C Programming Language, 1st edition".
- For several years the book "The C Programming Language, 1st edition" was the standard on the C programming.
- In 1983 a committee was formed by the American National Standards Institute (ANSI) to develop a modern definition for the programming language C .
- In 1988 they delivered the final standard definition ANSI C.

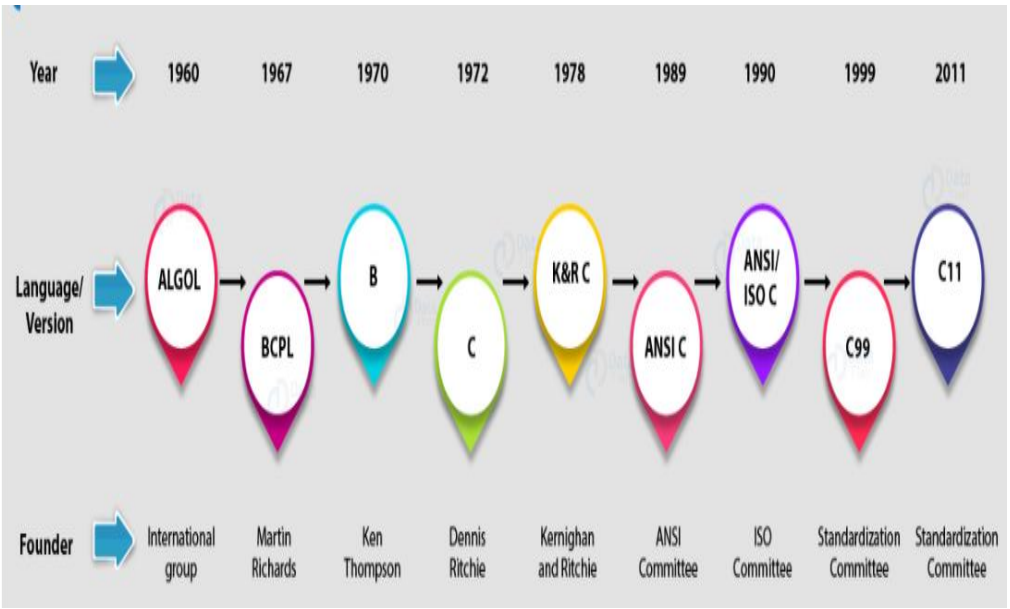


Fig.1.18. History of C

ALGOL – ALGORITHMIC LANGUAGE
BCPL – BASIC COMBINED PROGRAMMING LANGUAGE

WHY C?

- The C compiler supports both assembly language features and high-level language.
- It is best suitable for writing both system applications and most of the business packages.

- It is a portable language and hence, once the code is written, it can run on any computer system.
- C is basically used for developing Operating Systems.
- The first Operating System developed using C was Unix.

Features of C

1. **Simple and efficient** – The syntax style is easy to comprehend. We can use C to design applications that were previously designed by assembly language.
2. **Memory Management** – It allows you to allocate memory at the runtime, that is, it supports the concept of dynamic memory allocation.

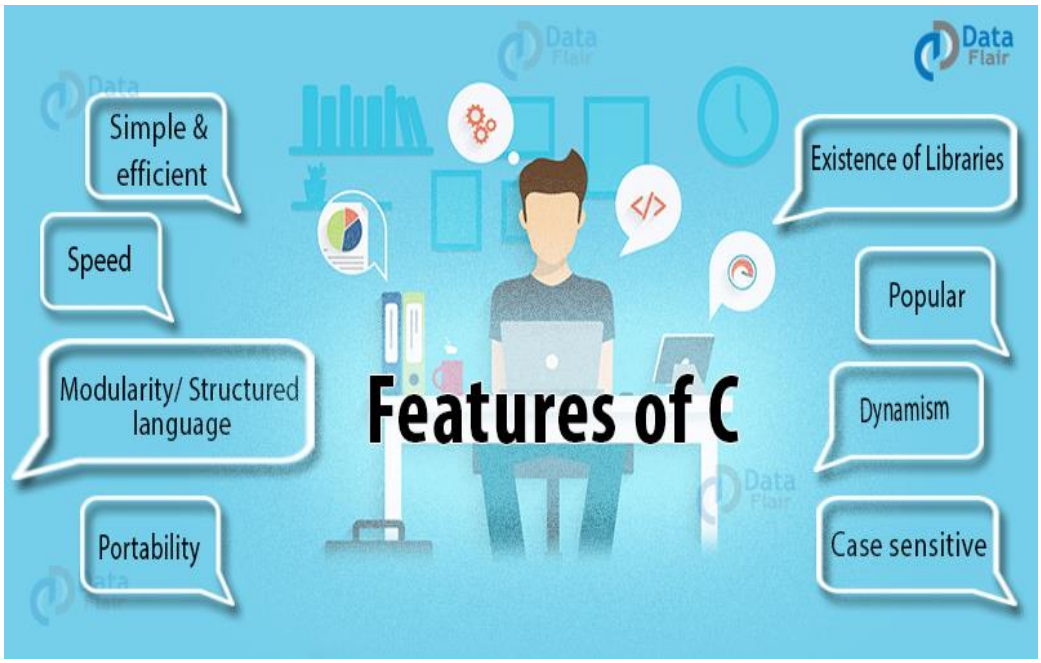


Fig.1.19. Features of C Programming Language

3. **Dynamic Memory Allocation-** When you are not sure about the memory requirements in your program and want to specify it at the run time, that is, when you run your program, you can do it manually.
4. **Pointers –** C language provides a pointer that stores the memory address as its value. Pointers are useful in storing and accessing data from memory. We will study this in detail in our upcoming unit.
5. **Case Sensitive –** It is pretty clear that lowercase and uppercase characters are treated differently in C. It means that if you write “program” and “Program”, both of them would connote different meanings in C. The ‘p’ in “program” is in lowercase format whereas, the ‘P’ in Program is in uppercase format.
6. **Compiler Based –** C is a compiler-based language, that is, to execute a code we first need to compile it.
7. **Structure Oriented/Modular –** C is a structured programming language. This means you can divide your code and task within a function to make it interactive. These functions also help in code reusability.

Applications of C Language

- It is used in the development of Operating Systems and Embedded Softwares.(Unix Kernal)
- It comes in handy when designing a compiler for other programming languages.
- Data structures and algorithms are implemented in C
- It acts as a base language to develop new languages. For instance, C++ was developed from C.
- Computer applications can be developed using C.
- Firmware is designed for electrical, industrial and communication appliances using C.

Advantages of C Programming Language

1. **Portable –** It is easy to install and operate and the result file is a .exe file that is easy to execute on any computer without any framework.
2. **Compiles faster –** C has a faster compiler that can compile 1000 lines of code in seconds and optimize the code to give speedy execution.
3. **User-defined functions –** C has many header files that define a lot of functions, making it easier for you to code. You can also create your functions; these are called user-defined functions (UDFs).
4. **C has a lower level of abstraction –** C is a very clear and descriptive language. You can, in a way, directly see into the machine without any conceptual hiding and so learning C first makes the concepts very clear for you to proceed.

Structure of C Program:

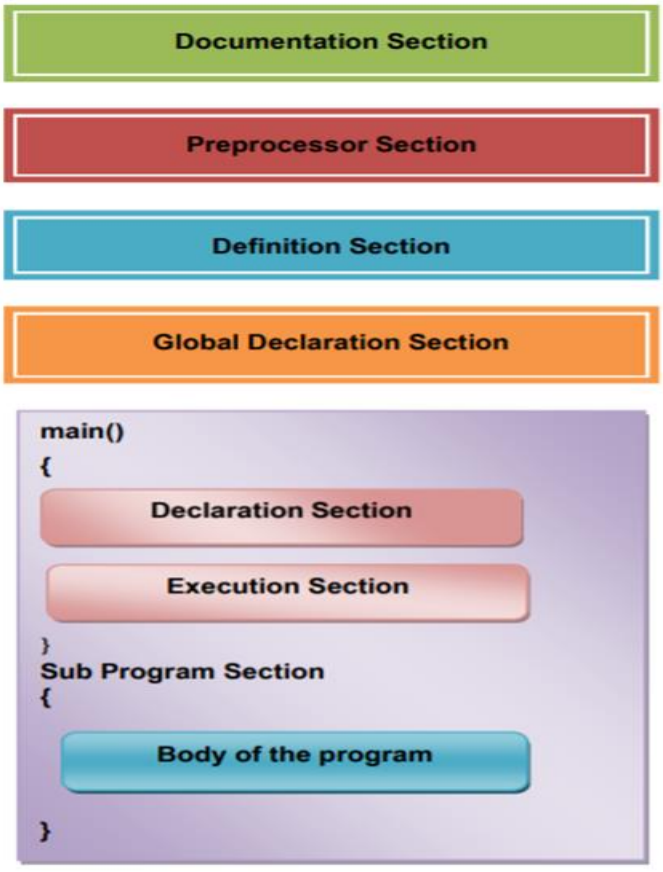


Fig.1.20. Structure of C Program

Documentation Section:

It consists of a set of comment lines

The comment lines begin with /* and ends with */ or a single set of // in the beginning of the line.

These lines are not executable

Comments are very helpful in identifying the program features.

Preprocessor Section:

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

E.g.: #include, #include, #define MAX 100, etc.,

Global Declaration Section:

The variables that are used in more than one function throughout the program are called global variables

Should be declared outside of all the functions i.e., before main ().

main ():

Every 'C' program must have one main() function, which specifies the starting of a 'C' program.

It contains the following two parts

Declaration Part: This part is used to declare the entire variables that are used in the executable part of the program and these are called local variables

Execution Part: It contains at least one valid C Statement.

The Execution of a program begins with opening brace "{and ends with closing brace}"

The closing brace of the main function is the logical end of the program.

Sub Program section:

Sub programs are basically functions are written by the user (user defined functions)

They may be written before or after a main () function and called within main () function.

This is optional to the programmer.

Points to be considered while writing a C program:

- All statements in 'C' program should be written in lower case letters. Uppercase letters are only used for symbolic constants.
- Blank space may be inserted between the words. Should not be used while declaring a variable, keyword, constant and function
- The program statements can be written anywhere between the two braces following the declaration part.
- All the statements should end with a semicolon (;)

Compilation and Execution of C program

1. Creating the program
2. Compiling the Program
3. Linking the Program with system library
4. Executing the program

Creating the program:

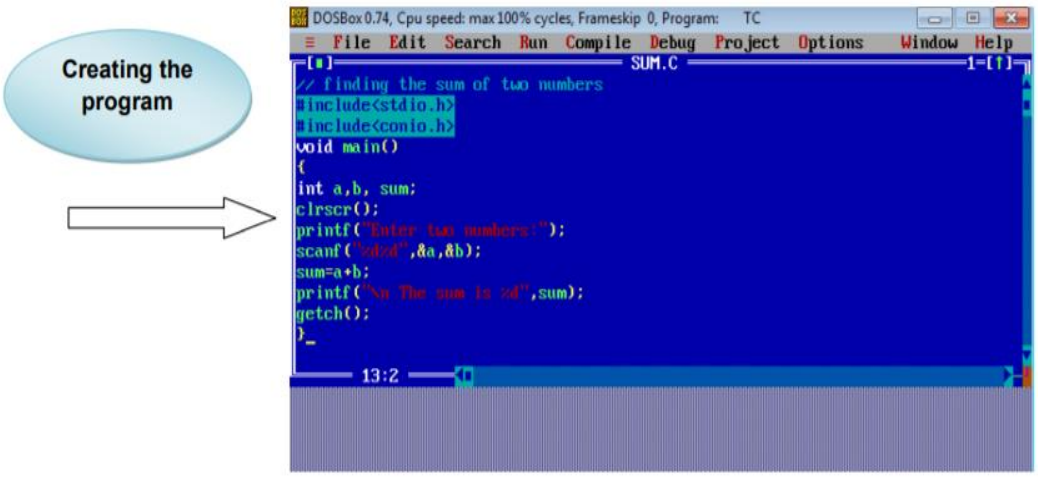


Fig.1.21. Writing A Program in TurboC

- Type the program and edit it in standard 'C' editor and save the program with .c as an extension.
- This is the source program

Compiling (Alt + F9) the Program:

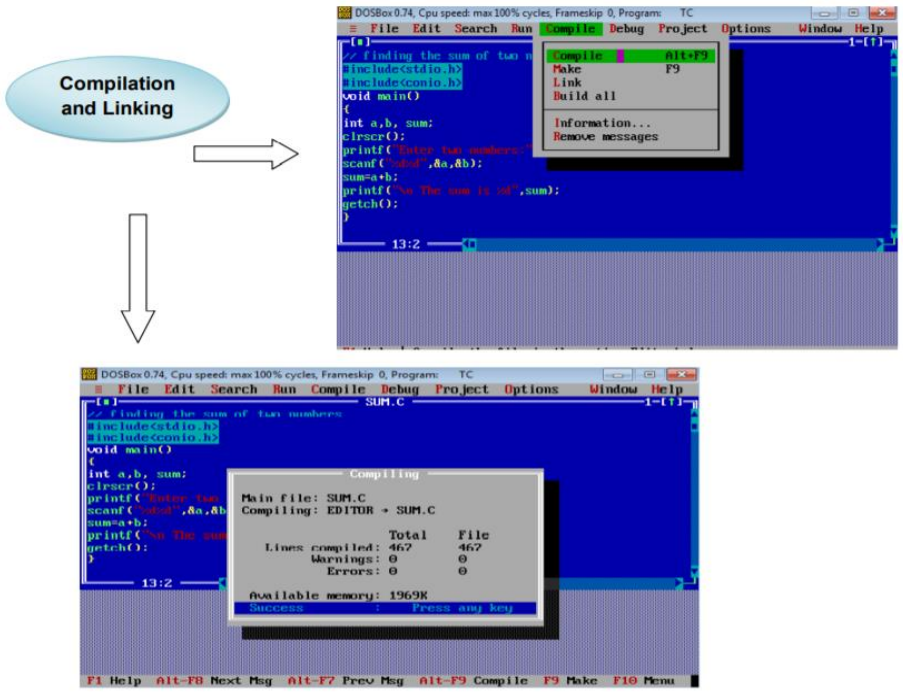


Fig.1.7. Compiling and Linking (With Pre-processor Header Files) the Program

- This is the process of converting the high-level language program to Machine level
- Language (Equivalent machine instruction) -> Compiler does it!
- Errors will be reported if there is any, after the compilation
- Otherwise the program will be converted into an object file (.obj file) as a result of the
- compilation
- After error correction the program has to be compiled again

Linking the program with system Library:

- Before executing a c program, it has to be linked with the included header files and other system libraries -> Done by the Linker

Executing the Program:

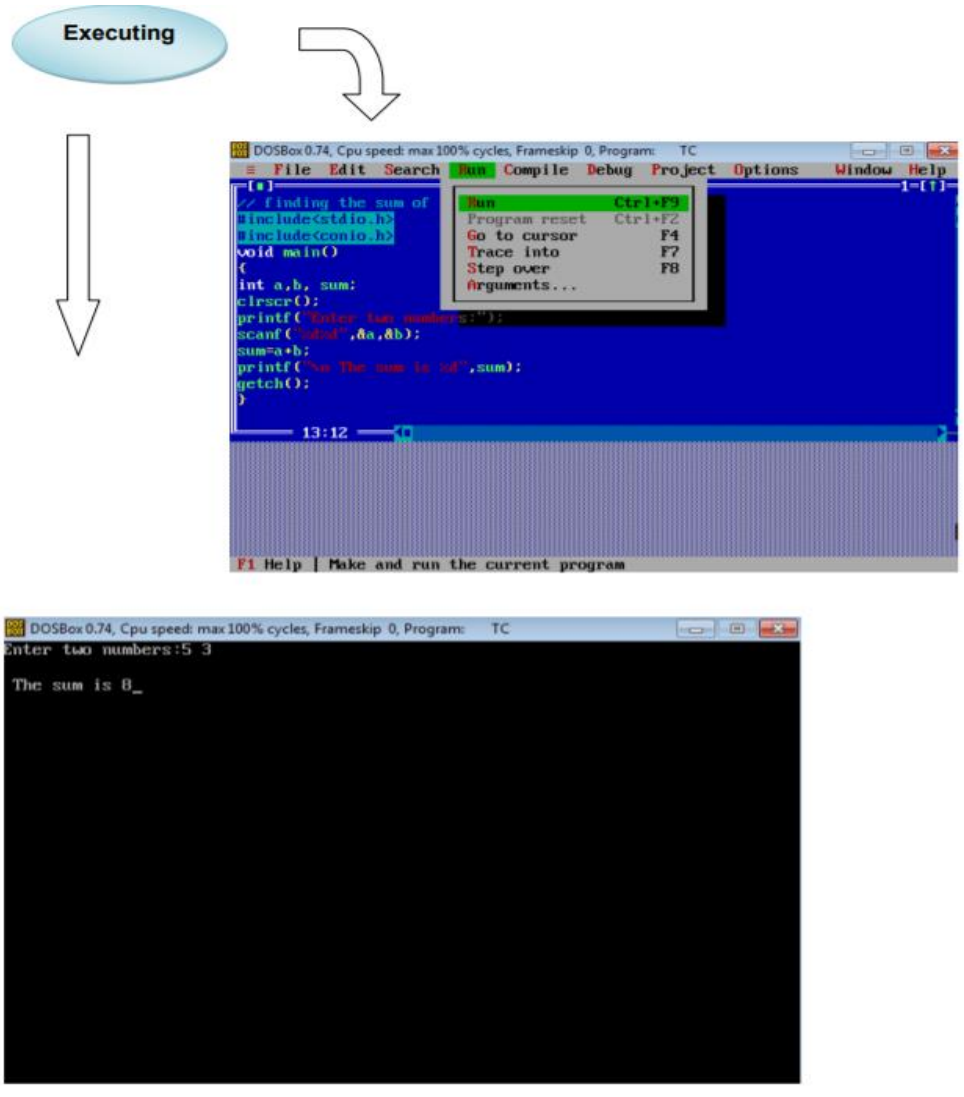


Fig.1.22. Executing the Program

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

main () is a special function in C programming language.

Reasons that make it special are -

- It defines starting point of the program.
- main is the first executed function.
- It controls all other child functions.
- Behaves as both user-defined and pre-defined function.
- Every software written in C must have a main function.

Various main () function declarations:

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

Example Program

```
/* addition.c – To find the average of two numbers and print them out together with their average */

#include <stdio.h>

void main( )

{

 int first, second;

float avg;

printf("Enter two numbers: ");

scanf("%d %d", &first, &second);

printf("The two numbers are: %d, %d", first, second);

avg = (first + second)/2;

printf("Their average is %f", avg);

}
```

Data Types in C

C has a concept of 'data types' which are used to define a variable before its use. Thedefinition of a variable will assign storage for the variable and define the type of data that will beheld in the location.The value of a variable can be changed any time.

C has the following basic built-in datatypes.

- int
- float
- double
- char

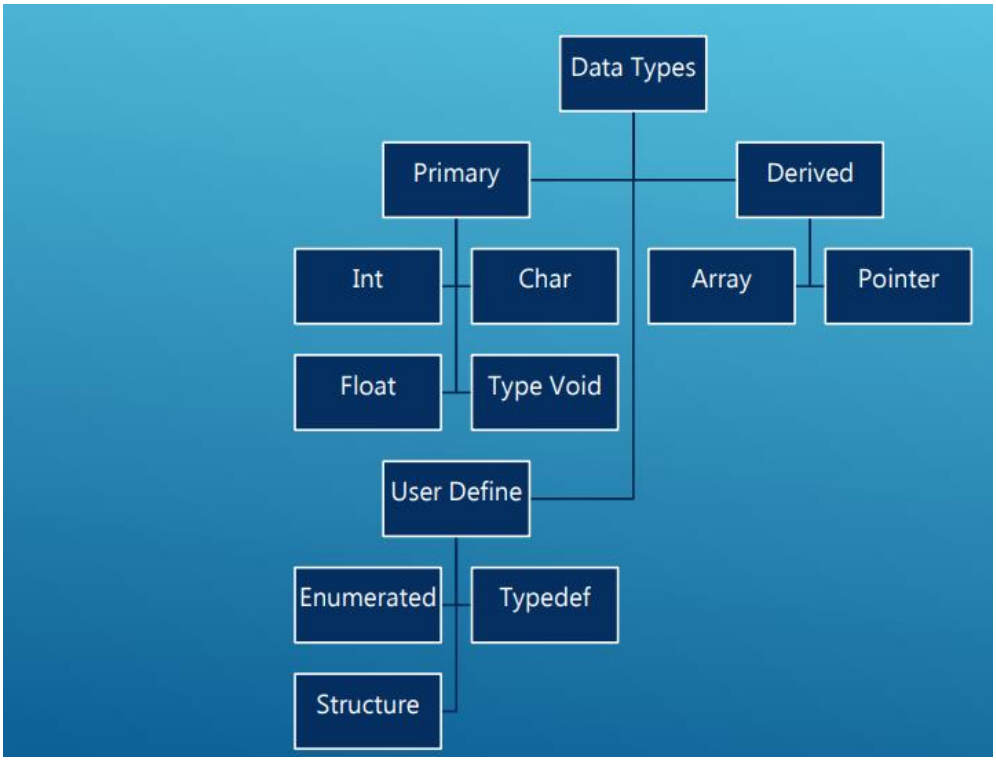


Fig.1.23. Data Types in C

The bytes occupied by each of the primary data types are

Table.1.1.Different Data Types in C

Data type	Description	Memory bytes	Control String	Example
Int	Integer Quantity	2 bytes	%d or %i	int a=12;
Char	Single Character	1 bytes	%C	char s='n';
float	Floating Point	4 bytes	%f	float f=29.777
Double	Double precision floating pointing no's	8 bytes	%lf	double d= 5843214

Integer Data Type:

Integers are whole numbers with a range of values, range of values are machine dependent. Generally, an integer occupies 2 bytes memory space and its value range limited to -32768 to +32767.

Table.1.2.Integer Data Type

Data Type	Control String	Size	Range
int	%d	2 bytes	-32768 to +32767
long int	%ld	4 bytes	-2^{31} to $+2^{31}$
unsigned int	%u	2 bytes	0 to 65535

CHAR DATA TYPE

- Character type variable can hold a single character.
- As there are singed and unsigned int (either short or long), in the same way there are signed and unsigned chars; both occupy 1 byte each, but having different ranges.
- Unsigned characters have values between 0 and 255; signed characters have values from –128 to 127.

FLOAT DATA TYPE

- The float data type is used to store fractional numbers (real numbers) with 6 digits of precision.
- Floating point numbers are denoted by the keyword float. When the accuracy of the floating-point number is insufficient, we can use the double to define the number.
- The double is same as float but with longer precision and takes double space (8 bytes) than float.
- To extend the precision further we can use long double which occupies 10 bytes of memory space.

Table.1.3. Float Data Type

Data Type	Size	Value Range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 Decimal Places
double	8 byte	2.3E-308 to 1.7E+308	15 Decimal Places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 Decimal Places

VOID DATA TYPE

- The void type has no values therefore we cannot declare it as variable as we did in case of integer and float.
- The void data type is usually used with function to specify its type.

- C program we declared "main ()" as void type because it does not return any value.

Array:

- An array in C language is a collection of similar data-type, means an array can hold value of a particular data type for which it has been declared.
- Arrays can be created from any of the C data-types int.

Pointer:

C Pointer is a special variable that can be used to store address of another variable.

ENUMERATED DATA TYPE (ENUM)

- Enumerated data type is a user defined data type having finite set of enumeration constants. The keyword 'enum' is used to create enumerated data type.
- Enumeration data type consists of named integer constants as a list.
- It start with 0 (zero) by default and value is incremented by 1 for the sequential identifiers in the list.

Syntax: Enum [data _ type] {const1, const2… constn};

Enum example in C:enum month { Jan, Feb, Mar }; or /* Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default */

enum month { Jan = 1, Feb, Mar };

/* Feb and Mar variables will be assigned to 2 and 3 respectively by default */ enum month { Jan = 20, Feb, Mar };

/* Jan is assigned to 20. Feb and Mar variables will be assigned to 21 and 22 respectively by default */

C Tokens

C tokens, Identifiers and Keywords are the basic elements of a C program.
C tokens are thebasic buildings blocks in C.
Smallest individual units in a C program are the C tokens.
C tokensare of six types. They are,
1. Keywords (e.g.: int, while),
2. Identifiers (e.g.: main, total),
3. Constants (e.g.: 10, 20),
4. Strings (e.g.: “total”, “hello”),
5. Special symbols (e.g.: (), {}),
6. Operators (e.g.: +, /, -, *)

Keywords

Keywords are those words whose meaning is already defined by Compiler.
They cannot beused as Variable Names.
There are 32 Keywords in C.
C Keywords are also calledas Reserved words.
There are 32 keywords in C.

They are given below:

Table.1.4. Keywords in C

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

Identifiers

Identifiers are the names given to various program elements such as variables, arrays &functions. Basically, identifiers are the sequences of alphabets or digits.

Rules for forming identifier name

- The first character must be an alphabet (uppercase or lowercase) or an underscore.
- All succeeding characters must be letters or digits.
- No space and special symbols are allowed between the identifiers.
- No two successive underscores are allowed.
- Keywords shouldn't be used as identifiers.

Constants

The constants refer to fixed values that the program may not change or modify duringits execution. Constants can be of any of the basic data types like an integer constant, afloating constant and a character constant. There is also a special type of constant calledenumeration constant.
E.g.:

Integer Constants- 45, 215u
Floating Constants- 3.14, 4513E-5L
Character Constants- \t, \n

Strings
A string in C is actually a one-dimensional array of characters which is terminated by a null character '\0'.

E.g.:
char str = {‘S’, ’A’, ’T’, ’H’, ’Y’, ’A’, ’B’, ’A’, ’M’, ’A’}

Special Symbols

The symbols other than alphabets, digits and white spaces for example - [] () {},; : * … = # are the special symbols.

Operators

An Operator is a symbol that specifies an operation to be performed on the operands. Thedata items that operators act upon are called operands. Operators which require two operandsare called Binary operators. Operators which require one operand are called Unary Operators.

Types of Operators

Depending upon their operation they are classified as
1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Sizeof() Operators

Arithmetic Operators

Arithmetic Operators are used to perform mathematical calculations like addition,subtraction, multiplication, division and modulus.

Table.1.5. Arithmetic Operators in C

S.NO	Operators	Operation	Example
1	+	Addition	A+B
2	-	Subtraction	A-B
3	*	multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

Rules for Arithmetic Operators:

1. C allows only one variable on left hand side of = eg. c=a*b is legal, but a*b=c is not legal.
2. Arithmetic operations are performed on the ASCII values of the characters and not on characters themselves
3. Operators must be explicitly written.
4. Operation between same type of data yields same type of data, but operation between

integer and float yield a float result.

Example Program

```
#include <stdio.h>
int main()
{
int m=40,n=20, add,sub,mul,div,mod;
add = m+n;
sub = m-n;
mul = m*n;
div = m/n;
mod = m%n;
printf("Addition of m, n is : %d\n", add);
printf("Subtraction of m, n is : %d\n", sub);
printf("Multiplication of m, n is : %d\n", mul);
printf("Division of m, n is : %d\n", div);
printf("Modulus of m, n is : %d\n", mod);
}
```

Output

Addition of m, n is: 60
Subtraction of m, n is: 20
Multiplication of m, n is: 800
Division of m, n is: 2
Modulus of m, n is: 0

Relational Operators:

Relational Operators are used to compare two or more operands. Operands maybe variables, constants or expression.

Table.1.6. Relational Operators in C

S.NO	Operators	Operation	Example
1	>	is greater than	m > n
2	<	is less than	m <n
3	>=	is greater than or equal to	m >= n
4	<=	is less than or equal to	m <= n
5	==	is equal to	m == n
6	!=	is not equal to	m!=n

Example Program

```
#include <stdio.h>
int main()
{
int m=40,n=20;
if (m == n)
{
printf("m and n are equal");
}
else
{
printf("m and n are not equal");
}
}
```

Output

m and n are not equal

Logical Operators:

Logical Operators are used to combine the results of two or more conditions. It isalso used to test more than one condition and make decision.

Table.1.7. Logical Operators in C

S.NO	Operators	Operation	Example	Description
1	&&	logical AND	(m>5)&&(n<5)	It returns true when both conditions are true
2	||	logical OR	(m>=10)||(n>=10)	It returns true when at-least one of the condition is true
3	!	logical NOT	!((m>5)&&(n<5))	It reverses the state of the operand "((m>5) && (n<5))" If "((m>5) && (n<5))" is true, logical NOT operator makes it false

Example Program

```
#include <stdio.h>
int main()
{
int a=40,b=20,c=30;
if ((a>b )&& (a >c))
{
printf(" a is greater than b and c");
```

```
}
else
if(b>c)
printf("b is greater than a and c");
else
prinf("c is greater than a and b");
}
```

Output

a is greater than b and c.

Conditional Operator

It itself checks the condition and executed the statement depending on the condition.

Syntax:

Condition? Exp1:Exp2

Example:

X=(a>b)?a:b

The '?:' operator acts as ternary operator. It first evaluates the condition, if it is true thenexp1 is evaluated, if condition is false then exp2 is evaluated. The drawback ofAssignment operator is that after the? or: only one statement can occur.

Example Program

```
#include <stdio.h>
int main()
{
int x,a=5,b=3;
x = (a>b) ? a : b ;
printf("x value is %d\n", x);
}
```

Output

x value is 5

Bitwise Operators:

Bitwise Operators are used for manipulation of data at bit level.

It operates on integeronly.

Table.1.8. Bitwise Operators in C

S.NO	Operators	Operation	Example	Description
1	&	Bitwise AND	X & Y	Will give 1 only when both inputs are 1
2	|	Bitwise OR	X | Y	Will give 1 when either of input is 1
3	^	Bitwise XOR	X ^ Y	Will give 1 when one input is 1 and other is 0
4	~	1's Complement	~X	Change all 1 to 0 and all 0 to 1
5	<<	Shift left	X<<Y	X gets multiplied by 2^Ynumber of times
6	>>	Shift right	X>>Y	X gets divided by 2^Y number of times

Example Program

```
#include <stdio.h>
main()
{
int c1=1,c2;
c2=c1<<2;
printf("Left shift by 2 bits c1<<2=%d",c2);
}
```

Output

Left shift by 2 bits c1<<2=4

Special operators:

sizeof () operator:

1. Sizeof operator is used to calcualte the size of data type or variables.
2. Sizeof operator will return the size in integer format.
3. Sizeof operator syntax looks more like a function but it is considered as an operator in c programming

Example of Size of Variables

```
#include<stdio.h>
int main()
{
 int ivar = 100;
 char cvar = 'a';
 float fvar = 10.10;
printf("%d", sizeof(ivar));
printf("%d", sizeof(cvar));
printf("%d", sizeof(fvar));
 return 0;
}
```

Output:

2 1 4

In the above example we have passed a variable to size of operator. It will print the value of variable using sizeof() operator.

Example of Sizeof Data Type

```
#include<stdio.h>
int main()
{
printf("%d", sizeof(int));
printf("%d", sizeof(char));
printf("%d", sizeof(float));
 return 0;
}
```

Output:

2 1 4

In this case we have directly passed an data type to an sizeof.

Example of Size of constant

```
include<stdio.h>
int main()
{
printf("%d", sizeof(10));
printf("%d", sizeof('A'));
printf("%d", sizeof(10.10));
 return 0;
}
```

Output:

2 1 4

In this example we have passed the constant value to a sizeof operator. In this case sizeof will print the size required by variable used to store the passed value.

Example of Nested sizeof operator

```
#include<stdio.h>
int main()
{
 int num = 10;
printf("%d", sizeof(sizeof(num)));
 return 0;
}
```

Output:

2

We can use nested sizeof in c programming. Inner sizeof will be executed in normal fashion and the result of inner sizeof will be passed as input to outer sizeof operator.
Innermost Sizeof operator will evaluate size of Variable “num” i.e 2 bytes Outer Sizeof will evaluate Size of constant “2” .i.e 2 bytes.

Comma(,) Operator:

1. Comma Operator has Lowest Precedence i.e it is having lowest priority so it is evaluated at last.
2. Comma operator returns the value of the rightmost operand when multiple commaoperators are used inside an expression.
3. Comma Operator Can acts as –

 Operator: In the Expression

 Separator: Function calls, Function definitions, Variable declarations and Enumdeclarations

Example:

```
#include<stdio.h>
void main()
{
 int num1 = 1, num2 = 2;
 int res;
 res = (num1, num2);
printf("%d", res);
}
```

Output

2

Consider above example

int num1 = 1, num2 = 2;// In variable Declaration as separator
res = (num1, num2);// In the Expression as operator
In this case value of rightmost operator will be assigned to the variable. In this case value of num2 will be assigned to variable res.

Examples of comma operator:

Type 1: Using Comma Operator along with Assignment

```
#include<stdio.h>
int main()
{
 int i;
i = 1,2,3;
printf("i:%d\n",i);
 return 0;
}
```

Output:

i:1

Explanation:

i = 1,2,3;

1. Above Expression contain 3 comma operator and 1 assignment operator.
2. If we check precedence table then we can say that “Comma” operator has lowest precedence than assignment operator
3. So Assignment statement will be executed first .
4. 1 is assigned to variable “i”.

Type 2 : Using Comma Operator with Round Braces

```
#include<stdio.h>
int main()
{
 int i;
i = (1,2,3);
printf("i:%d\n",i);
 return 0;
}
```

Output:

i:3

Explanation:

i = (1,2,3);

1. Bracket has highest priority than any operator.
2. Inside bracket we have 2 comma operators.
3. Comma operator has associativity from Left to Right.
4. Comma Operator will return rightmost operand

i = (1,2,3) Assign 3 to variable i.

Type 3: Using Comma Operator inside printf statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("Computer","Programming");
getch();
```

```
}
```
Output:
Computer
You might feel that answer of this statement should be “Programming” because commaoperator always returns rightmost operator, in case of printf statement once comma is readthen it will consider preceding things as variable or values for format specifier.
Type 4: Using Comma Operator inside Switch cases.
```
#include<stdio.h>
#include<conio.h>
void main()
{
int choice = 2 ;
switch(choice)
{
 case 1,2,1:
printf("\nAllas");
 break;
 case 1,3,2:
printf("\nBabo");
 break;
 case 4,5,3:
printf("\nHurray");
 break;
}
}
```
Output:
Babo
Type 5: Using Comma Operator inside For Loop
```
#include<stdio.h>
int main()
{
int i,j;
for(i=0,j=0;i<5;i++)
 {
printf("\nValue of J : %d",j);
j++;
 }
return(0);
}
```
Output:
Value of J : 0
Value of J : 1
Value of J : 2
Value of J : 3
Value of J : 4
Type 6: Using Comma Operator for multiple Declaration
```
#include<stdio.h>
int main()
{
int num1,num2;
```

```
int a=10,b=20;
return(0);
}
```
Note: Use of comma operator for multiple declaration in same statement.

Variable:

 A variable is an identifier that is used to represent some specified type of information within a designated portion of the program.

 A variable may take different values at different times during the execution

Rules for naming the variable

 A variable name can be any combination of 1 to 8 alphabets, digit, or underscore

 The first character must be an alphabet or an underscore (_).

 The length of variable should not exceed 8 characters length, and some of the 'C' compiler can be recognizedupto 31 characters.

Scope of a variable

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block is called local variable,
2. Outside of all functions is called global variable.
3. In the definition of function parameters which is called formal parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main() function.

```
#include <stdio.h>
main ()
{
 /* local variable declaration */
 int a, b, c;
 /* actual initialization */
 a = 10;
 b = 20;
 c = a + b;
printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <stdio.h>
/* global variable declaration */
int g;
main ()
```

```
{
 /* local variable declaration */
 int a, b;
 /* actual initialization */
 a = 10;
 b = 20;
 g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
}
```

PRECEDENCE AND ASSOCIATIVELY OF OPERATORS

If an arithmetic expression is given, there are some rules to be followed to evaluate thevalue of it. These rules are called as the priority rules. They are also called as the hierarchyrules. According to these rules, the expression is evaluated as follows;

Rule 1: - If an expression contains parentheses, the expression within the parentheses will be performed first. Within the parentheses, the priority is to be followed.

Rule 2: - If it has more than parentheses, the inner parenthesis is performed first.

Rule 3: - If more than one symbols of same priority, it will be executed from left to right.C operators in order of precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied

Table.1.9. Precedence and Associativity of Operators in C

Operator	Operation	Associativity	Priority
() [] . -> ++ --	Parentheses Brackets (array subscript) Dot operator Structure operator Postfix increment/decrement	left-to-right	1
++ -- + - ! ~ (type) * & sizeof	Prefix inc/decrement Unary plus/minus Not operator,complement Type cast Pointer operator Address operator Determine size in bytes	right-to-left	2
* / %	Multiplication/division/modulus	left-to-right	3
+ -	Addition/subtraction	left-to-right	4
<< >>	Bitwise shift left Bitwise shift right	left-to-right	5
< <= > >=	Relational less than less than or equal to Relational greater than greater than or equal to	left-to-right	6
== !=	Relational is equal to is not equal to	left-to-right	7
&	Bitwise AND Bitwise exclusive	left-to-right	8
^	Bitwise exclusive OR	left-to-right	9
|	Bitwise inclusive OR	left –to-right	10
&&	Logical AND	left-to-right	11
||	Logical OR	left-to-right	12
?:	Ternary conditiona	right-to-left	13
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left	14
,	Comma	left-to-right	15

Example for evaluating an expression
Let X = 2 , Y =5
then the value of the expression (((Y - 1) / X) * (X + Y)) is calculated as:-
(Y - 1) = (5 - 1) = 4 = T1 (T 1 / X) = (4 / 2) = 2 = T2 (X + Y) = (2 + 5) = 7 = T3 (T2 * T3) = (2 * 7) = 14 The evaluations are made according to the priority rule.

Type conversion in expressions.

Type conversion is the method of converting one type of data into another data type.

There are two types of type conversion.

1. Automatic type conversion

2. Type casting

Automatic type conversion

- This type of conversion is done automatically. The resultant value of an expression depends upon the operand which occupies more space, which means the result value converted into highest data type.
- The compiler converts all operands into the data type of the largest operand.
- This type of type conversion is done implicitly, this method is called as implicit type conversion.

Eg.1

```
float a,b,c;
a=10,b=3;

c=a/b
```

output= > c= 3.3 {4 bytes(float) (All the variables are same datatype}

Eg.2

```
int a,b,c; a=10,b=3;
c=a/b;
```

output= >c=3{2 bytes(int)}

Eg.3

```
int a;
float b,c;
a=10,b=3;
c=a/b;
```

output=> c=3.3 {4 bytes(float) highest data type is float}

Type casting

- This method is used, when user wants to change the type of the data.

General Format for type casting is (datatype) operand
Eg.1

int x=10, y=3; z=(float)x/y;(ie z=10.0/3;) output=>z=3.3(float)
Eg:2

int x=10,y=3; z=x/(float)y;(ie z=10/3.0;) output=>3.3(float)

- The type of the x is not changed, only the type of the value can be changed
- Since the type of conversion is done explicitly, this type conversion is called as explicit type conversion

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

1. All integer types to be converted to float.
2. All float types to be converted to double.
3. All character types to be converted to integer

Input and Output statements
In 'c' language several functions ara available for input/output operations.
Thesefunctions are collectively known as the standard I/O library.
1.Unformatted input /output statements
2. Formatted input /output statements
Unformatted Input /Output statements
These statements are used to input /output a single /group of characters from/to the input/output devices.Here, the user cannot specify the type of data that is going to be input/output.
The following are the Unformatted input /output statements available in 'C'.

Table.1.10. Unformatted Input and output Statements in C

Input	Output
getchar( )	putchar( )
getc( )	putc( )
gets( )	Puts( )

single character input-getchar( ) function:
A getchar( ) function reads only one character through the keyboard.
Syntax: char variable=getchar( );
Example:
char x;
x=getchar( );

single character output-putchar( ) function:

A putchar( ) function is used to display one character at a time on the standard output device.
Syntax:putchar(charvariable);
Example:
char x;
putchar(x);

the getc( ) function
This is used to accept a single character from the standard input to a character variable.
Syntax: character variable=getc( );
Example:
char c;
c=getc( );
the putc( ) function
This is used to display a single character variable to standard output device.
Syntax:putc(character variable);
Example:
char c;
putc(c);
the gets( ) and puts( ) function
The gets( ) function is used to read the string from the standard input device.
Syntax:gets(string variable);
Example:
gets(s);
The puts( ) function is used to display the string to the standard output device.
Syntax:puts(string variable);
Example:
puts(s);
Program using gets and puts function

```
#include<stdio.h>
main()
{
char scientist[40];
puts("Enter name");
gets(scientist);
puts("Print the Name");
puts(scientist);
}
```

output:
Enter Name:Abdul Kalam
Print the Name:Abdul Kalam

Formatted input /output statements
The function which is used to give the value of variable through keyboard is called inputfunction. The function which is used to display or print the value on the screen is called output
function.
Note: - In C language we use two built in functions, one is used for reading and another is used

for displaying the result on the screen. They are scanf() and printf() functions. They are stored inthe header file named stdio.h.

General format for scanf( ) function

scanf("control string", &variable1, &variable2,……)

The control sting specifies the field format in which the data is to be entered.

%d –integer
%f – float
%c- char
%s – string
% ld – long integer
%u – Unsigned Integer

Example:

scanf("%d",&x) – reading an integer value, the value will be stored in x.
scanf("%d%f", &x,&a) - reading a integer and a float value.

Output Function: To print the value on the screen or to store the value on the file, the output functions are used. printf() is the function which is use to display the output on the screen.

The General format of the printf() function is

printf("control string",variable1,variable2,…..);

Example

printf("%d",x) – printing the integer value x.
printf("%d%f", x,a)- printing a integer and float value using a single printf function.

Formatted Output of Integer: Similar to formatted input, there is a formatted output also to have the output in a format manner.

In this control string consists of three types of items.

 Characters that will be printed on the screen as they appear
 Format specification that define the output format for display of each item
 Escape sequence characters such as

\n – new line
\b – back space
\f – form feed
\r – carriage return
\t - horizontal tab
\v – vertical tab

Text / Reference Books:

1. Byron S Gottfried, "Programming with C", Schaum's Outlines, 2 nd Edition, Tata McGrawHill, 2006.
2. Dromey R.G., "How to Solve it by Computer", Pearson Education, 4 th Reprint, 2007.
3. Kernighan, B.W. and Ritchie, D.M., "The C Programming language", 2 nd Edition, Pearson Education, 2006.
4. Balaguruswami. E., "Programming in C", TMH Publications, 2003.
5. Yashavant P. Kanetkar, 'LET US C', 5 th Edition.2005.
6. Stevens, 'Graphics programming in C', BPB Publication, 2006.
7. Subburaj. R , 'Programming in C', Vikas Publishing, 1 st Edition, 2000.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF MECHANICAL ENGINEERING

DEPARTMENT OF AUTOMOBILE, AERONUTICAL, MECHATRONICS AND MECHANICAL ENGINEERING

UNIT - II

Programming in C - SCSA1103

CONTROLS STRUCTURES AND FUNCTIONS

Control structures: Conditional statements – Looping statements – Functions: Library Functions - User Defined– Function Prototype - Function Definition – Types of Functions – Functions with and without Arguments-Functions with no return and with Return Values - solving simple scientific and statistical problems- Nested Functions - Recursion.

CONTROL STATEMENTS IN C

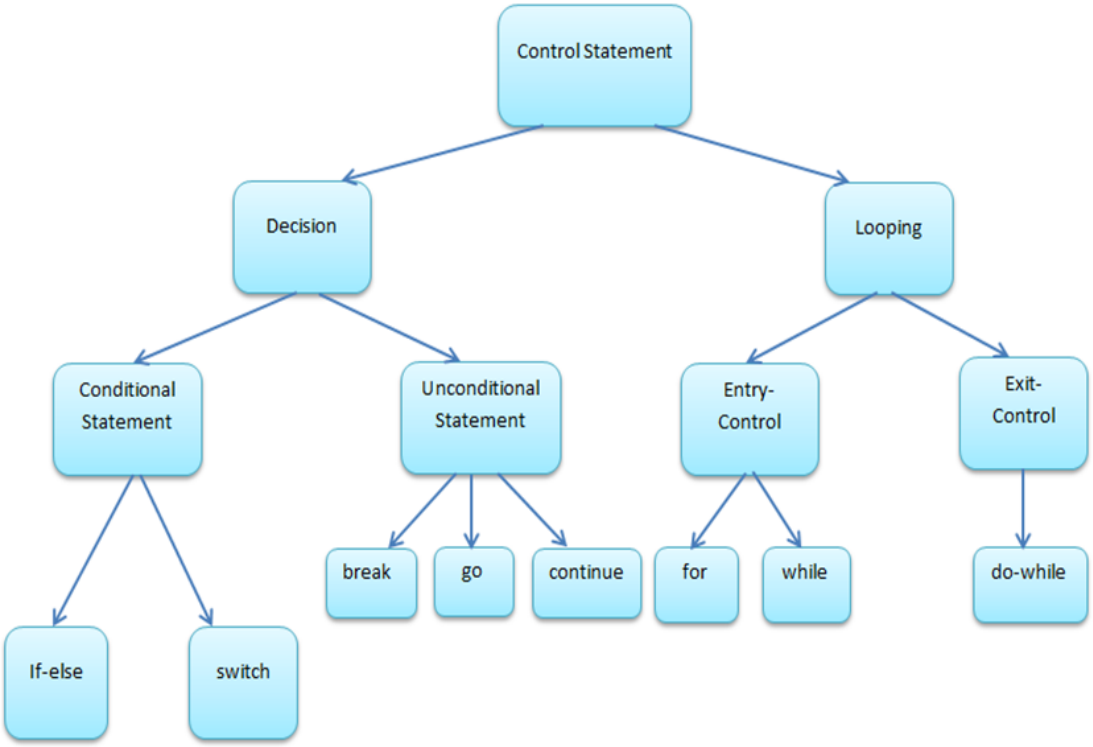


Fig.2.1. Control Statements in C

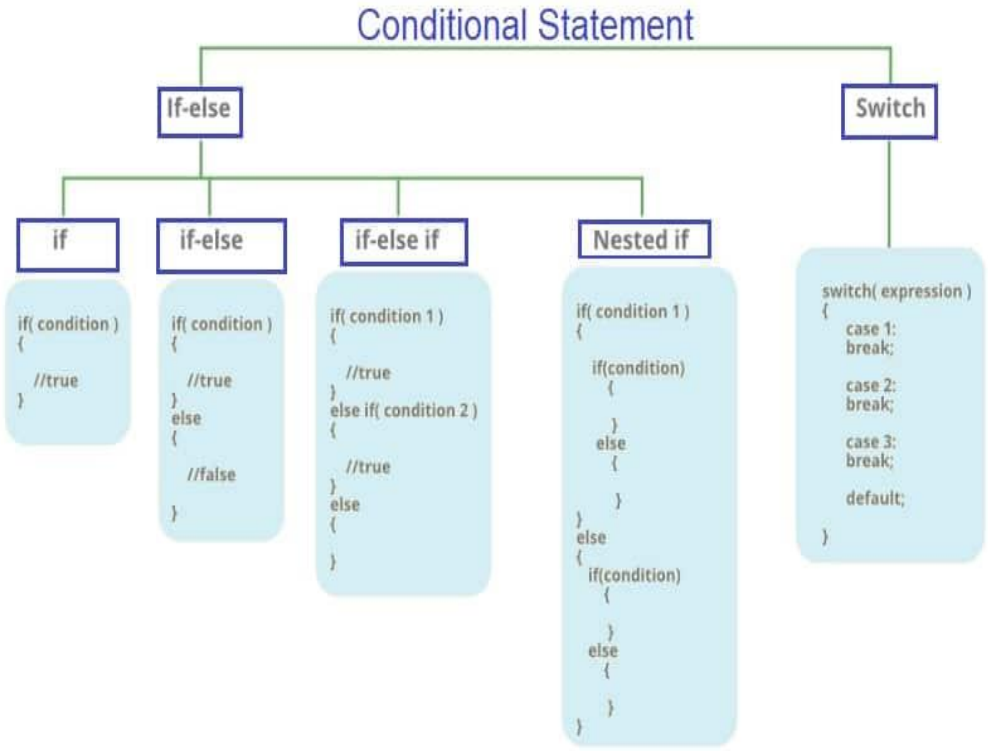


Fig.2.2. Conditional Statements in C

If Statement:

 The if statement is a decision-making statement.

 It is used to control the flow of execution of the statement and also used to the logically whether the condition is true or false

 It is always used in conjunction with condition.

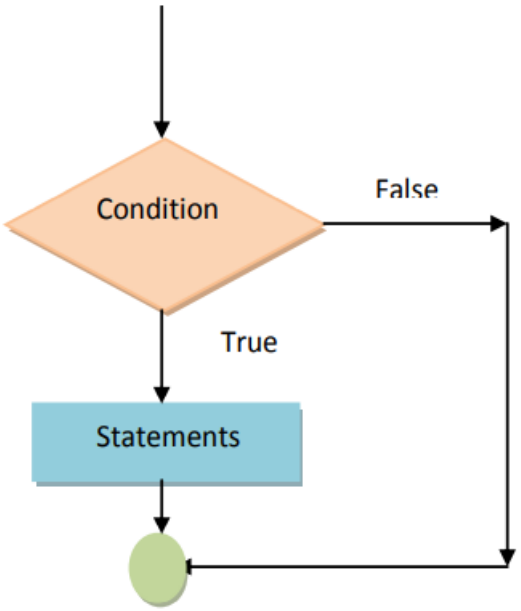


Fig.2.3. IF Statement in C

Syntax:

```
If(condition)
{
True statements;
}
```

 If the condition is true, then the true statements are executed.
 If the condition is false then the true statements are not executed, instead the program skips past them.
 The condition is given by relational operators like ==,<=,>=,!=,etc.

Example 1: //program to check whether the entered number is less than 25

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
printf("Enter one value");
scanf("%d",&i);
if(i<=25)
printf("The entered no %d is < 25",i);
getch();
}
```

Output:

```
Enter one value 5
The entered no 5 is < 25
```

Example 2: //program to calculate the sum and multiplication using if Statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,n;
clrscr();
printf("Enter two values");
n=scanf("%d%d",&a,&b);
if(n==2)
{
printf("the sum of two numbers : %d",a+b);
printf("the product of two numbers:%d",a*b);
}
getch();
}
```

Output:

```
Enter two value 5 10
the sum of two numbers : 15
the product of two numbers : 50
```

if. else statement:

 It is basically two-way decision-making statement and always used in conjunction with condition.

 It is used to control the flow of expression and also used to carry the logical test and then pick up one of the two possible actions depending on the logical test.

 If the condition is true, then the true statements are executed otherwise false statements are executed.

 The true and false statements may be single or group of statements.

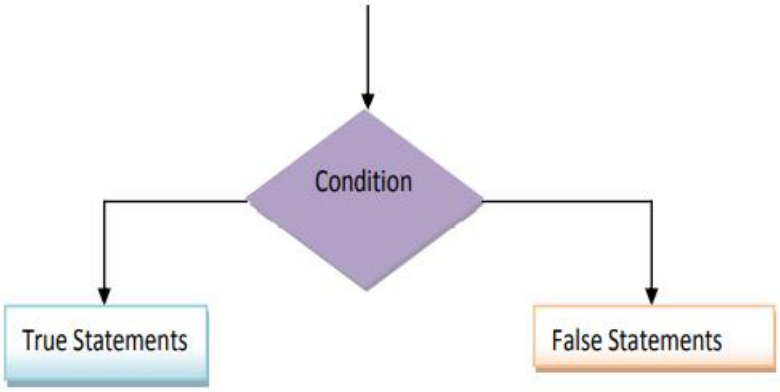


Fig2.4. IF Else Statement in C

Syntax:

If (condition)
True statements;
else
False statements;

Example 1: //program to find the greatest of two number.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("Enter two value");
scanf("%d%d",&a,&b);
if(a>b)
printf("The given no %d is greatest",a);
else
printf("The given no %d is greatest",b);
}
```

Output:

Enter two value 5 10
The given no 10 is greatest

Nested if..else Statement:

When a series of if_else statements are needed in a program, we can write an entire

if_else statement inside another if and it can be further nested. This is called nesting if.

Syntax:

```
if(condition 1)
{
if(condition 2)
{
True statement 2;
else
False statement 2;
}
else
False statement 1;
}
```

Example 1: //program to find the greatest of three numbers.

```
#include <stdio.h>
int main ()
{
 /* local variable definition */
 int a = 100;
 int b = 200;
 /* check the boolean condition */
if( a == 100 )
 {
 /* if condition is true then check the following */
if( b == 200 )
 {
 /* if condition is true then print the following */
printf("Value of a is 100 and b is 200\n" );
 }
 }
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
 return 0;
}
```

Output:

Value of a is 100 and b is 200
Exact value of a is: 100
Exact value of b is: 200

If_else Ladder:

 Nested if statements will become complex, if several conditions have to be checked.
 In such situations we can use the else if ladder.

Syntax:

```
if(condition 1)
{
if(condition 2)
{
True statement 2;
}
elseif(condition 3)
{
True statement 3;
else
False statement 3;
}
else
False statement 1;
}
```

Switch Statement

 The switch statement is used to execute a particular group of statements from several available groups of statements.
 It allows us to make a decision from the number of choices.
 It is a multi-way decision statement.

Rules for writing switch () statement.

 The expression in switch statement must be an integer value or a character constant.
 No real numbers are used in an expression.
 Each case block and default block must be terminated with break statement.
 The default is optional and can be placed anywhere, but usually placed at end.
 The 'case' keyword must terminate with colon(:).
 Cases should not be identical.
 The values of switch expression is compared with the case constant expression in the order specified i.e., from top to bottom.

Syntax:

```
switch(expression)
{
case 1:
statement;
break;
case 2:
statement;
break;
switch
default: statements
break;
```

}

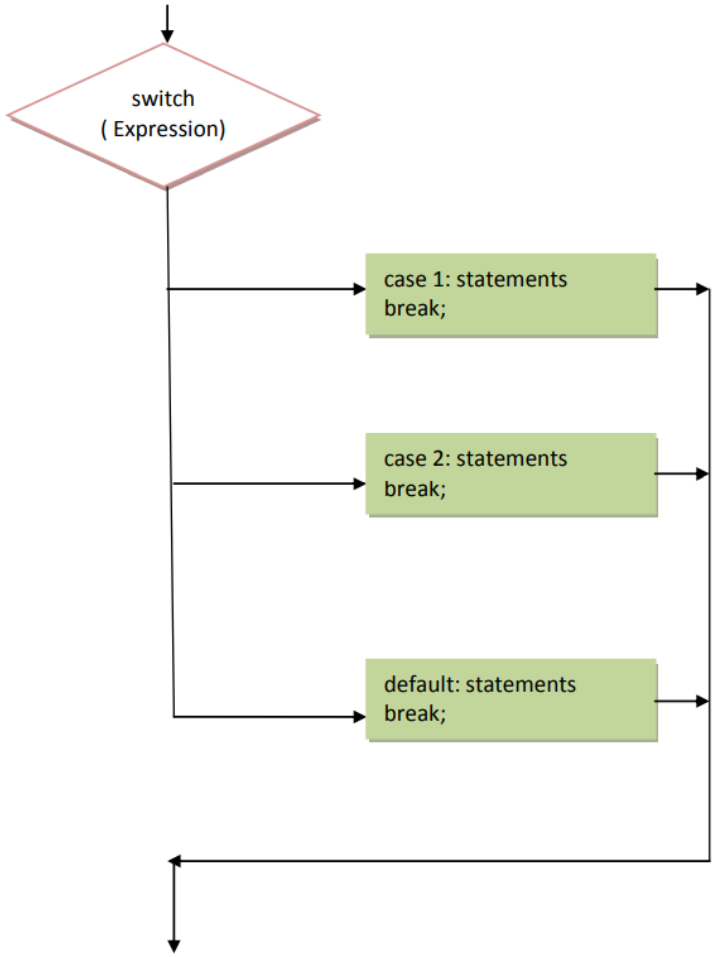


Fig.2.5. Switch Case Statement in C

// program to print the give number is odd / even using switch case statement.

```
#include<stdio.h>
#include<conio.h> void main()
{
int a,b,c;
printf("Enter one value"); scanf("%d",&a);
switch(a%2)
{
case 0:
printf("The given no %d is even", a);
break;
default :
printf("The given no %d is odd", a);
break;
}
}
```

Output:

Enter one value 5

The given no 5 is odd

Unconditional statement

Break statement

 The break statement is used to terminate the loop.

 When the keyword break is used inside any loop, control automatically transferred to the first statement after the loop.

Syntax:break;

//program to print the number upto 5 using break statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
for(i=1;i<=10;i++)
{
if(i==6)
break;
printf("%d",i);
}
}
```

Output:

1 2 3 4 5

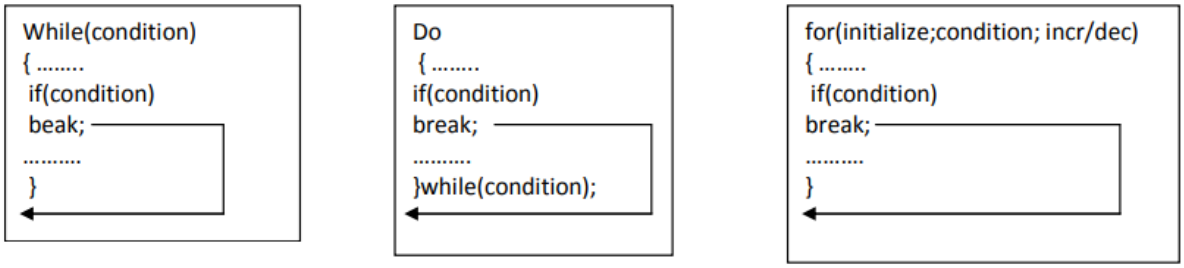


Fig.2.6. Break Statement in C

Continue Statement

 In some situation, we want to take the control to the beginning of the loop, bypassing the statement inside the loop which have not been executed, for this purpose the continue is used.

 When the statement continue is encountered inside any loop, control automatically passes to the beginning of the loop.

Syntax:continue;

Example:

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

}

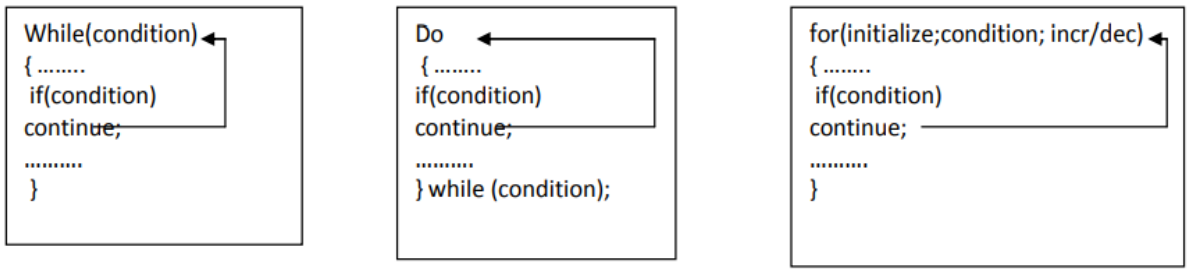


Fig.2.7. Continue Statement in C

Table.2.1.Difference between break and continue

Break	Continue
Break statement takes the control to the outside of the loop	Continue statement takes the control to be beginning of the loop
It is also in switch statement	This can be used only in loop statements
Always associated with if condition in loop	This is also associated with if condition

Goto Statement:

 C provides the goto statement to transfer control unconditionally from one place to another place in the program.

 A goto statement can change the program control to almost anywhere in the program unconditionally.

 The goto statement require a label to identify the place to move the execution.

 The label is a valid variable name and must be ended with colon(:).

Syntax:

1. gotolabel; 2. label:
…… …..……
……. ……..
label: goto label;

*** program to print the given number is equal or not*/**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("Enter the numbers");
scanf("%d%d",&a,&b);
if(a==b)
goto equal;
else
{
printf("%d and %d are not equal",a,b);
```

```
exit(0);
}
equal: printf("%d and %d are equal",a,b);
}
```

Output:

```
Enter the numbers 4 5
4 and 5 are not equal
Enter the numbers 5 5
5 and 5 are equal
```

LOOPING STATEMENTS

A loop statement allows us to execute certain block of code repeatedly until test condition is false.

There are 3 types of loops in C programming:

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

for loop:

Syntax of For Loop:

```
for ( variable initialization; condition; variable update )
{
Code to execute while the condition is true
}
```

The initialization statement is executed only once at the beginning of the for loop.
Then the testexpression is checked by the program.
If the test expression is false, for loop is terminated.
But if test expression is true then the code/s inside body of for loop is executed and then updateexpression is updated.
This process repeats until test expression is false.

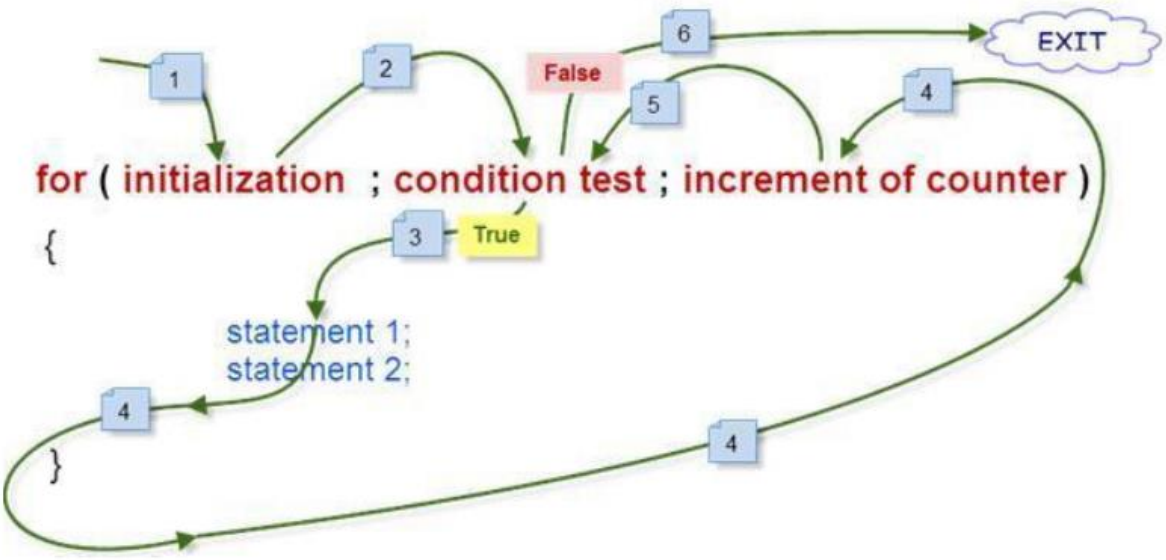


Fig.2.8. For Loop in C

for loop example

Write a program to find the sum of first n natural numbers where n is entered by user.
Note: 1,2,3... are called natural numbers.

```
 #include <stdio.h>
 void main(){
int n, count, sum=0;
printf("Enter the value of n.\n");
scanf("%d",&n);
for(count=1;count<=n;++count) //for loop terminates if count>n
{
sum+=count; /* this statement is equivalent to
 sum=sum+count */
}
printf("Sum=%d",sum);
 }
```

Output

```
Enter the value ofn: 19
Sum=190
```

In this program, the user is asked to enter the value of n. Suppose you entered 19 then, count isinitialized to 1 at first. Then, the test expression in the for loop,i.e., (count<= n) becomes true.So, the code in the body of for loop is executed which makes sum to 1. Then, the expression++count is executed and again the test expression is checked, which becomes true. Again, thebody of for loop is executed which makes sum to 3 and this process continues. When count is20, the test condition becomes false and the for loop is terminated.

/* C program to check whether a number is prime or not. */

```
#include <stdio.h>
int main()
{
int n, i, flag=0;
printf("Enter a positive integer: ");
scanf("%d",&n);
for(i=2;i<=n/2;++i)
{
if(n%i==0)
 {
 flag=1;
 break;
 }
}
if (flag==0)
printf("%d is a prime number.",n);
else
printf("%d is not a prime number.",n);
return 0;
```

}

Output

Enter a positive integer: 29
29 is a prime number.

This program takes a positive integer from user and stores it in variable n. Then, for loop isexecuted which checks whether the number entered by user is perfectly divisible by i or notstarting with initial value of i equals to 2 and increasing the value of i in each iteration. If thenumber entered by user is perfectly divisible by i then, flag is set to 1 and that number will not
be a prime number but, if the number is not perfectly divisible by i until test condition i<=n/2 istrue means, it is only divisible by 1 and that number itself and that number is a prime number.

Different Types of For Loop in C Programming

For loop can be implemented in different ways
1. Single Statement inside For Loop
2. Multiple Statements inside For Loop
3. No Statement inside For Loop
4. Semicolon at the end of For Loop
5. Multiple Initialization Statement inside For
6. Missing Initialization in For Loop
7. Missing Increment/Decrement Statement
8. Infinite For Loop
9. Condition with no Conditional Operator.

Single Statement inside For Loop:

```
for(i=0;i<5;i++)
printf("sathyabama");
```

1. Above code will print sathyabama word 5 times.
2. We have single statement inside for loop body.
3. No need to wrap printf inside opening and closing curly block.
4. Curly Block is Optional.

Multiple Statements inside For Loop

```
for(i=0;i<5;i++)
{
printf("Statement 1"); printf("Statement 2");
printf("Statement 3"); if(condition)
{
 --------
 --------
}
}
```

If we have block of code that is to be executed multiple times then we can use curly braces to wrap multiple statement in for loop

No Statement inside For Loop

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

It is bodyless for loop. It is used to increment value of “i”.This are not used generally. Atthe end ,for loop value of i will be 5.

Semicolon at the end of For Loop:

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

- We will not get compile error if semicolon is at the end of for loop.
- This is perfectly legal statement in C Programming.
- This statement is similar to bodyless for loop.

Multiple Initialization Statement inside For:

```
for(i=0,j=0;i<5;i++)
{
statement1;
statement2;
statement3;
}
```

Multiple initialization statements must be seperated by Comma .

Missing Increment/Decrement Statement:

```
for(i=0;i<5;)
{
statement1;
statement2;
statement3;
i++;
}
```

we have to explicitly alter the value i in the loop body.

Missing Initialization in For Loop:

```
i = 0;
for(;i<5;i++)
{
statement1;
statement2;
statement3;
}
```

we have to set value of ‘i’ before entering in the loop otherwise it will take garbage value of i“.

Infinite For Loop:

```
i = 0;
for( ; ; )
{
statement1;
statement2;
statement3;
if(breaking condition)
break;
i++;
}
```

Infinite for loop must have breaking condition in order to break for loop. otherwise it will cause
overflow of stack.

While Loop

while loop repeatedly executes a target statement as long as a givencondition is true.

Syntax:

Initialization;
while(condition)
{

Increment/decrement;
}

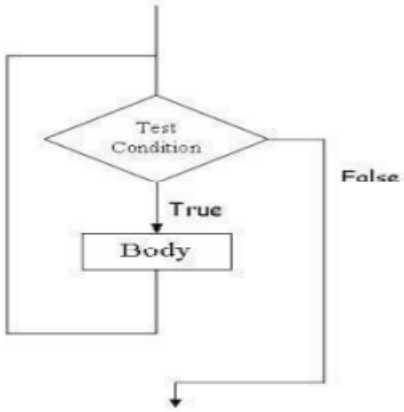


Fig.2.9. While Loop in C

For Single Line of Code – Opening and Closing braces are not needed.
while(1) is used
for Infinite Loop
 Initialization, Increment/Decrement and Condition steps are on different Line.
 While Loop is also Entry Controlled Loop.[i.e conditions are checked if found true then and then only code is executed**.**

Examples:

```
#include <stdio.h>
int main()
{
int y = 0;/* Don't forget to declare variables*/
while ( y< 10 ) {/* While y is less than 10 */
printf( "%d\n", y );
y++; /* Update y so the condition can be met
eventually */
}
getchar();
}
```

C Program to Find Number of Digits in a Number

```
#include <stdio.h>
int main()
{
int n,count=0;
```

```
printf("Enter an integer: ");
scanf("%d", &n);
while(n!=0)
{
n/=10; /* n=n/10 */
++count;
}
printf("Number of digits: %d",count);
}
```

Output:

Enter an integer: 34523 Number of digits: 5

Types of infinite while loop

Semicolon at the end of while loop

```
#include<stdio.h>
void main()
{
int num=300;
while(num>255); //Note it Carefully
printf("Hello");
}
```

Output :

Will not print anything

1. In the above program, Condition is specified in the While Loop
2. Semicolon at the end of while indicated while without body.
3. In the program variable numdoesn‟tget incremented, condition remains true forever.
4. As Above program does not have Loop body, Itwon‟t print anything

Non-Zero Number as a Parameter

```
#include<stdio.h>
void main()
{
while(1)
printf("Hello");
}
```

Output :

Infinite Time "Hello" word

1. We can specify any non-zero positive number inside while loop
2. Non zero number is specified in the while loop which means that while loop will remains true forever.

Subscript variable remains the same

```
#include<stdio.h>
void main()
{
int num=20;
while(num>10) {
printf("Hello");
```

```
}
}
```
Output :
Infinite Time "Hello C" word

Explanation :
1. Condition is specified in while Loop, but terminating condition is not specified and even we haven‟t modified the condition variable.
2. In this case our subscript variable (Variable used to Repeat action) is not either incremented or decremented
3. so while remains true forever.

Character as a Parameter in While Loop
```
#include<stdio.h>
void main()
{
while('A')
printf("Hello");
}
```
Output :
Infinite Time "Hello" word

Explanation :
1. Character is Represented in integer in the form of ASCII internally.
2. Any Character is Converted into Non-zero Integer ASCII value
3. Any Non-zero ASCII value is TRUE condition, that is why Loop executes forever.

DO..WHILE
DO..WHILE loops executes the body of the loop atleast once.
Syntax:
```
initialization;
do
{
--------------
--------------
incrementation;
}while(condition);
```

The condition is tested at the end of the block instead of the beginning, so the block will beexecuted at least once. If the condition is true, it go back to the beginning of the block andexecute it again. A do..while loop is almost same as a while loop except that the loop body isguaranteed to execute at least once.
- It is Exit Controlled Loop.
- Initialization, Incrementation and Condition steps are on different Line.
- It is also called Bottom Tested.
- Semicolon must be added after the while.

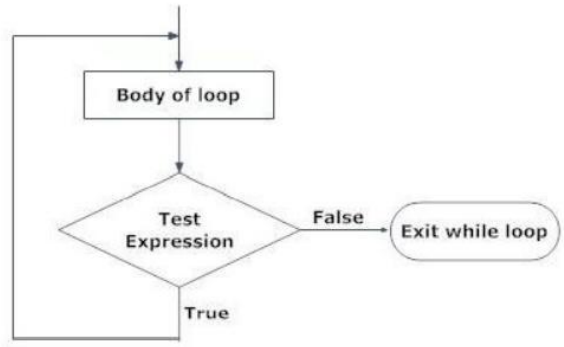


Fig.2.10.Do - While Loop in C

Example:

```
#include <stdio.h>
int main()
{
int z;
z = 0; do {
/* " sathyabama is printed at least one time even though the
condition is false */
printf( "sathyabama\n" ); } while ( z != 0 );
getchar();
}
```

C Program to print first 5 Natural Numbers

Using For Loop

```
#include<stdio.h>
void main()
{ inti = 1;
for (i = 1; i<= 5; i++)
{
printf("%d", i);
}
}
```

```
1
2
3
4
5
```

Using While Loop

```
#include<stdio.h>
void main()
{ inti = 1;
while (i<= 5)
{
printf("%d", i); i++;
}
}
```

Using Do-While Loop

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

```
void main()
{
int i = 1;
do
{
printf("%d", i);
i++;
} while (i<= 5);
}
```

FUNCTIONS
LIBRARY FUNCTIONS

Definition

C Library functions are inbuilt functions in C language which are clustered in a group andstored in a common place called Library. Each and every library functions in C executes explicitfunctions. In order to get the pre- defined output instead of writing our own code, these libraryfunctions will be used. Header file consists of these library functions like Function prototype anddata definitions.

 Every input and output operations (e.g., writing to the terminal) and all mathematicaloperations (e.g., evaluation of sines and cosines) are put into operation by libraryfunctions.
 The C library functions are declared in header files (.h) and it is represented as[file_name].h
 The Syntax of using C library functions in the header file is declared as"#include<file_name.h>". Using this syntax we can make use of those library functions.
 #include<filename.h>" command defines that in C program all the codes are included inthe header files followed by execution using compiler.
 It is required to call the suitable header file at the beginning of the program in terminal inorder to use a library function. A header file is called by means of the pre-processorstatement given below,#include<filename.h>

Whereas the filename represents the header file name and #include is a pre- processor directive.To access a library function the function name must be denoted, followed by a list ofarguments, which denotes the information being passed to the function.

Example

In case if you want to make use of printf() function, the header file <stdio.h> should be included atthe beginning of the C program.

```
#include <stdio.h>
int main()
{
/* NOTE: Error occurs if printf() statement is written without using
the header file */
printf(" Hello World");
}
```

The „main() function" is also a library function which is called at the initial of the program.

Example

To find the square root of a number we use our own part of code to find them but this maynot be most efficient process which is time consuming too. Hence in C programming by declaring

the square root function sqrt() under the library function “math.h” will be used to find them rapidlyand less time consuming too. Square root program using the library functions is given below:

Finding Square root Using Library Function

```
#include <stdio.h>
#include <math.h>
int main(){
float num,root;
printf("Enter a number to find square root.");
scanf("%f",&num);
root=sqrt(num); /* Computes the square root of num and stores in
root. */
printf("Square root of %.2f=%.2f",num,root);
return 0;
}
```

List of Standard Library Functions in C Programming

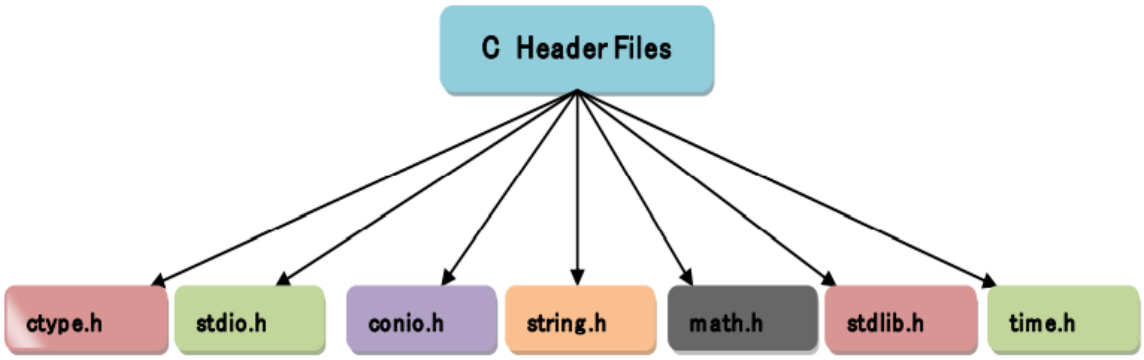


Fig.2.11.Standard Library Functions in C

Adding User Defined functions in C library:

 In C Programming we can declare our own functions in C library which is called as userdefined functions.

 It is possible to include, remove, change and access our own user defined function to orfrom C library functions.

 Once the defined function is added to the library it is merely available for all C programswhich are more beneficial of including user defined function in C library function

 Once it is declared it can be used anywhere in the C program just like using other C libraryfunctions.

 By using these library functions in GCC compilers (latest version), compilation time can beconsumed since these functions are accessible in C library in the compiled form.

 Commonly the header files in C program are saved as ”file_name.h” in which all libraryfunctions are obtainable. These header files include source code and this source code isfurther added in main C program file where we include this header file via “#include<file_name.h>” command.

Steps for adding user defined functions in C library:

Step 1:

For instance, hereby given below is a test function that is going to be included in the C library function. Write and save the below function in a file as "addition.c"

```
addition(int a, int b)
{
int sum;
total =a + b;
return sum;
}
```

Step 2:

Compile "addition.c" file by using Alt + F9 keys (in turbo C).

Step 3:

A compiled form of "addition.c" file would be created as "addition.obj".

Step 4:

To add this function to library, use the command given below (in turbo C).

```
c:\>tlib math.lib + c:\ addition.obj
```

+ represents including c:\addition.obj file in the math library.

We can delete this file using – (minus).

Step 5:

Create a file "addition.h" and declare sample of addition() function like below.

```
int addition (int a, int b);
```

Now "addition.h" file has the prototype of function "addition".

Note: Since directory name changes for each and every IDE, Kindly create, compile and add files in the particular directory.

Step 6:

Here is an example to see how to use our newly added library function in a C program.

```
# include <stdio.h>
// User defined function is included here.
# include "c:\\addition.h"
int main ( )
{
int total;
// calling function from library
total = addition (10, 20);
printf ("Total = %d \n", total);
}
```

Output:

```
Total = 30
```

 Source code checking for all header files can be checked inside "include" directory following C compiler that is installed in system.

 For instance, if you install DevC++ compiler in C directory in our system, "C:\DevCpp\include" is the path where all header files will be readily available.

Mostly used header files in C:

C library functions and header files in which they are declared in conio.h is listed below:

Table.2.2.C Library Functions

S.No	Header file	Description
1	stdio.h	A standard input/output header file where Input/ Output functions are declared
2	conio.h	Console input/output header file
3	string.h	String functions are defined in this header file
4	stdlib.h	The general functions used in the C program is defined in this header file.
5	math.h	Mathematical related functions are defined in this header file.
6	time.h	Time and clock allied functions are defined in this header file.
7	ctype.h	Every character managing functions are declared in this header file
8	errno.h	This header file contains Error handling functions.
9	assert.h	Diagnostics functions are declared in this header file.

C – conio.h library functions

The entire C programming inbuilt functions that are declared in conio.h header file are given below. The source code for conio.h header file is also given below for your reference.

List of inbuilt conio.h file C functions:

Table.2.3. List of conio.h Header Files

S.no	Function	Description
1	clrscr()	This function is used to clear the output screen.
2	getch()	It reads character from keyboard
3	getche()	It reads character from keyboard and echoes to o/p screen
4	textcolor()	This function is used to change the text colour
5	textbackground()	This function is used to change text background

C – stdio.h library functions

Inbuilt functions of C declared in stdio.h header file are given below.

Table.2.4.List of stdio.h Header Files

S.no	Function	Description
1	printf()	This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen
2	scanf()	This function is used to read a character, string, numeric data from keyboard.
3	getc()	It reads character from file
4	gets()	It reads line from keyboard
5	getchar()	It reads character from keyboard
6	puts()	It writes line to o/p screen
7	putchar()	It writes a character to screen
8	clearerr( )	Clears the error indicators
9	f open()	All file handling functions are defined in this header file.
10	f close()	closes an opened file
11	getw()	reads an integer from file
12	putw()	writes an integer to file
13	f getc()	reads a character from file
14	putc()	writes a character to file
15	f putc()	writes a character to file
16	f gets()	reads string from a file, per line at a time
17	f puts()	writes string to a file
18	f eof()	finds end of file
19	f getchar	reads a character from keyboard
20	f getc()	reads a character from file
21	f printf()	writes formatted data to a file
22	f scanf()	reads formatted data from a file
23	f getchar	reads a character from keyboard
24	f putchar	writes a character from keyboard
25	f seek()	moves file pointer position to given location
26	SEEK_SET	moves file pointer position to the beginning of the file
27	SEEK_CUR	moves file pointer position to given location
28	SEEK_END	moves file pointer position to the end of file.
29	f tell()	gives current position of file pointer
30	rewind()	moves file pointer position to the beginning of the file
31	putc()	writes a character to file
32	sprint()	writes formatted output to string
33	sscanf()	Reads formatted input from a string
34	remove()	deletes a file
35	fflush()	flushes a file

Functions

 A function is a group of statement that is used to perform a specified task which repeatedly occurs in the main program. By using function, we can divide the complex problem into a manageable problem.

 A function can help to avoid redundancy.

 Function can be of two types, there are

1. Built-in Function (or) Predefined Function (or) Library Function
2. User defined Function

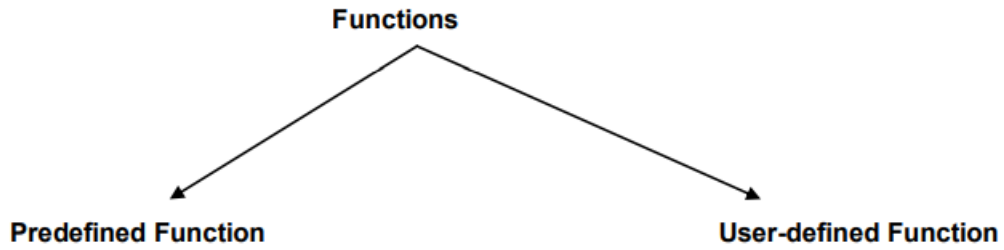


Fig.2.12.Classification of Functions

Table.2.5.Difference between Predefined and User Defined Functions

Predefined Function	User-defined function
Predefined function is a function which is already defined in the header file (Example: math.h, string.h, etc)	User- Defined function is a function which is created by the user as per requirement of its owner
Predefined Function is a part of a header file, which are called at runtime	User- Defined function are part of the program which are compiled at runtime
The Predefined function name is given by the developer	User- Defined function name created by the user
Predefined Function name cannot be changed	User defined Function name can be changed

User Defined Functions

 The function defined by the users according to their context (or) requirements is known as a user defined function.

 The User defined function is written by the programmer to perform specific task (or) operation, which is repeatedly used in the main program.

 These functions are helpful to break down the large program into a number of the smaller function.

 The user can modify the function in order to meet their requirements.

 Every user define function has three parts namely

Function Declaration
Function Calling
Function Definition

Need for user-defined function

 While it is possible to write any complex program under the function, and it leads to a number of problems, such as

 The problem becomes too large and complex.
 The user can‟tgo through at a glance
 The task of debugging, testing and maintenance become difficult.
 If a problem is divided into a number of parts, then each part may be independently coded and later it combined into a single program. These subprograms are called functions, it is much easier to understand, debug and test the program.

Merits of User-Defined Function

 The length of the source program can be reduced by dividing it into smaller functions
 It provides modularity to the program
 It is easy to identify and debug an error
 Once created a user defined function, can be reused in other programs
 Function facilitates top-down programming approach
 The Function enables a programmer to build a customized library of repeatedly used routines
 Function helps to avoid coding of repeated programming of the similar instruction

Elements of User-Defined Function

1. Function Declaration
2. Function Call
3. Function Definition

Function Declaration

- Like normal variable in a program, the function can also be declared before they
- defined and invoked
- Function declaration must end with semicolon (;)
- A function declaration must declare after the header file
- The list of parameters must be separated by comma.
- The name of the parameter is optional, but the data type is a must.
- If the function does not return any value, then the return type void is must.
- If there are no parameters, simply place void in braces.
- The data type of actual and formal parameter must match.

Syntax:

Return_typefunction_name (datatype parameter1, datatype parameter2,…);

Description:

Return type: type of function

Function_name : name of the function

Parameter list or argument list : list of parameters that the function

can convey.

Example:

int add(int x,inty,int z);

Function Call

The function call be called by simply specifying the name of the function, return

value and parameters if presence.

Syntax:function_name();

function_name(parameter);

return_value =function_name (parameter);

Description:

function_name : Name of the function

Parameter : Actual value passed to the calling function

Example

fun();

fun(a,b);

fun(10,20);

c=fun(a,b);

e=fun(2.3,40);

Function Definition

 It is the process of specifying and establishing the user defined function by specifying all of its element and characteristics.

Syntax:

Return_typefunction_name (datatype parameter1, datatype parameter2)

Example 1

```
#include<stdio.h>
#include<conio.h>
void add(); //Function Declaration void sub();//Function Declaration
void main()
{
clrscr();
add(); //Function call
sub(); //Function call
getch();
}
void add() //Function Definition
{
int a,b,c;
printf("Enter two values");
```

```
scanf(“%d%d”,&a,&b); c=a+b;
printf(,add=%d‘,c);
}
void sub() //Function Definition
{
int a,b,c;
printf(“Enter two values”);
scanf(“%d%d”,&a,&b);
c=a-b;
printf(“sub=%d”,c);
}
```

Example 2 :

//Program to check whether the given number is odd or even

```
#include<stdio.h>
#include<conio.h>
void oddoreven()
{
printf("Enter One value");
scanf("%d",&oe);
if(oe%2==0)
printf("The Given Number%d is even");
else
printf("The Given Number %d is odd");
}
void main()
{
clrscr();
oddoreven();
getch();
}
```

Function Parameter

 The Parameter provides the data communication between the calling function and calledfunction.

 There are two types of parameters.

o Actual parameter: passing the parameters from the calling function to thecalled function i.e the parameter, return in function is called actual parameter

o Formal parameter: the parameter which is defined in the called function i.e. Theparameter, return in the function definition is called formal parameter

Example:

```
main()
{
…….. Where ……..a,b are the actual Fun(a,b);
……..
parameters
……..
} x,y are formal parameter
Fun(int x,int y)
{
……
……
}
```

Example Program

```
#include<stdio.h>
#include<conio.h>
void add(int,int); //Function Declaration Output:
void sub(float,int);//Function Declaration
void main() add=7
{ sub=-2.500000
clrscr();
add(3,4); //Function call
sub(2.5,5); //Function call
getch();
}
```

```
void add(int a,int b)//Function Definition
{
int c;
c=a+b;
printf(“add=%d”,c);
}
void sub(float a, int b) //Function Definition
{
float c;
c=a-b;
printf(“sub=%f”,c);
}
```

Example 2:

//program for factorial of given

```
number #include<stdio.h>
#include<conio.h> void main()
{
int fact(int);
int f;
clrscr();
printf("Enter one value");
scanf("%d",&f);
printf("The Factorial of given number %d is %d",f,fact(f));
getch();
}
int fact(int f)
{
if(f==1) return 1;
else
return(f*fact(f-1));
}
```

Output:

Enter one value 5

The Factorial of given number 5 is 120.

Function Prototype (or) Function Interface

 The functions are classified into four types depends on whether the argumentsare present or not, whether a value is returned or not. These are calledfunction prototype.

 In ‘C’ while defining user defined function, it is must to declare its prototype.

 A prototype states the compiler to check the return type and arguments type ofthe function.

 A function prototype declaration consists of the function’s return type, nameand argument. It always ends with semicolon. The following are the functionprototypes

o Function with no argument and no return value.

o Function with argument and no return value.

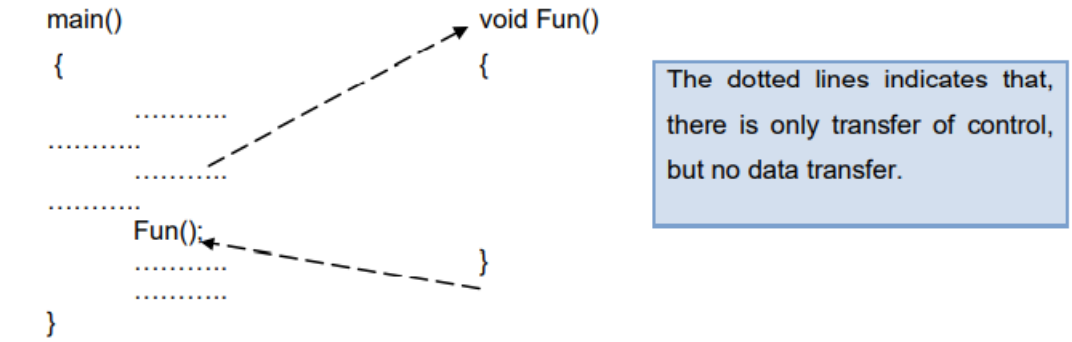
o Function with argument and with return value.

o Function with no argument with return value.

Function with no argument and no return value

 In this prototype, no data transfer takes place between the calling function andthe called function. i.e., the called program does not receive any data from thecalling program and does not send back any value to the calling program.

Syntax:-



Example program 1

```
#include<stdio.h>

#include<conio.h>

void mul();

void main()
```

The dotted lines indicates that,there is only transfer of control,but no data transfer.

Output:

Enter two values 6 4

mul=24

```
{
clrscr();
mul();
getch();
}
void mul()
{
int a,b,c;
printf(“Enter two values”);
scanf(“%d%d”,&a,&b);
c=a*b;
printf(“mul=%d”,c);
}
```

Example program 2

```
//Program for finding the area of a circle using Function with no argument
and no return value
I#include<stdio.h>
#include<conio.h>
void circle();
void main()
{
circle();
}
void circle()
{
int r;
float cir;
printf("Enter radius");
scanf("%d",&r);
cir=3.14*r*r;
printf("The area of circle is %f",cir);
}
```

Output:

Enter radius 5

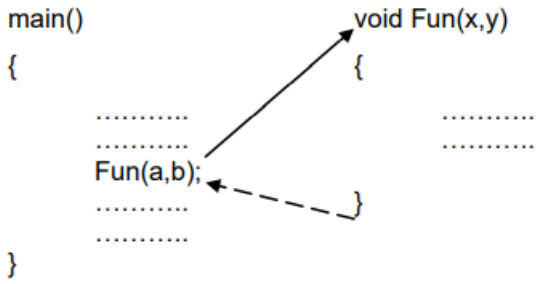
The area of circle 78.500000

Function with argument and no return value

 In this prototype, data is transferred from the calling function to calledfunction. i.e., the called function receives some data from the calling functionand does not send back any values to calling function

 It is one way data communication.

Syntax:-



The solid lines indicate data transfer and dotted line indicates a transfer of control.

a and b are the actual parameters

x and y are formal parameters

Example program 1:

```
#include<stdio.h>
#include<conio.h>
void add(int,int);
void main()
{
clrscr();
int a,b;
printf("Enter two values");
scanf("%d%d",&a,&b);
add(a,b);
getch();
}
void add(int x,int y)
{
int c;
c=x+y;
printf("add=%d",c);
```

```
}
```

Output:

Enter two values 6 4

add=10

Example program 2:

//Program to find the area of a circle using Function with argument and no return value

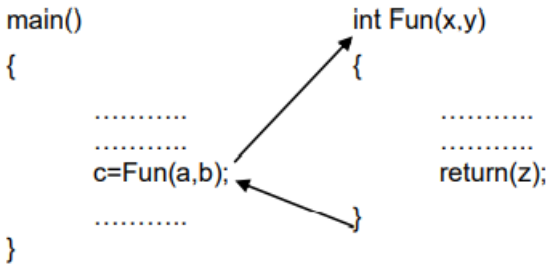
```
#include<stdio.h>
#include<conio.h>
void circle(int);
void main()
{
int r;
clrscr();
printf("Enter radius");
scanf("%d",&r);
circle(r);
}
void circle(int r)
{
float cir;
cir=3.14*r*r;
printf("The area of circle is %f",cir);
getch();
}
```

Function with argument and with return value.

 In this prototype, the data is transferred between the calling function andcalled function. i.e., the called function receives some data from the callingfunction and sends back returned value to the calling function.

 It is twoway data communication

Syntax:-



The solid lines indicates data transfer takes place in between thecalling program and called program

a,b are the actual parameter

x,y are formal parameter

Example program 1:

#include<stdio.h>

#include<conio.h>

void add(int,int);

The solid lines indicates data transfer takes place in between thecalling program and called program

a,b are the actual parameter

x,y are formal parameter

Output:

Enter two values 6 4

Add=10

```
void main()
{
clrscr();
int a,b,c;
printf("Enter two values");
scanf("%d%d",&a,&b);
c=add(a,b);
printf("Add=%d",c);
getch();
}
void add(int x,int y)
{
int m;
m=x+y;
```

```
return m;
}
```

Example Program 2

```
// Program to find the area of a circle using Function with argument
and with return value
#include<stdio.h>
#include<conio.h>
float circle(int);
void main()
{
int r;
clrscr();
printf("Enter radius");
scanf("%d",&r);
printf("the area of circle is %f",circle(r));
getch();
}
float circle(int r)
{
float cir;
cir=3.14*r*r;
return cir;
}
```

Output:

Enter radius 5

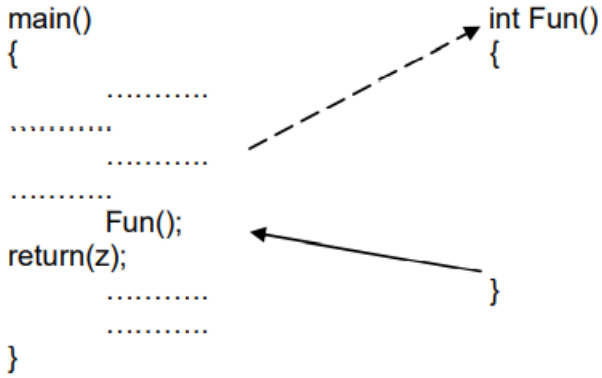
The area of circle 78.500000

Function with no argument with return value

 In this prototype, the calling function cannot pass any arguments to the calledfunction, but the called program may send some return value to the calling function.

 It is one way data communication

Syntax:-



The dotted line indicates a control transfer to the called program and the solid line indicates data return to the calling program

Example program 1

```
#include<stdio.h>
#include<conio.h>
int add();
void main()
{
clrscr();
int z;
z=add();
printf("Add=%d",z);
getch();
}
int add()
{
int a,b,c;
printf("Enter two values");
scanf("%d%d",&a,&b);
c=a+b;
return c;
}
```

Output:

Enter two values 6 4

Add=10

Example Program 2

// Program to the area of a circle using no argument with a return value

```
#include<stdio.h>
#include<conio.h>
float circle();
void main()
{
clrscr();
printf("the area of circle is %f",circle());
getch();
}
float circle()
{
float cir;
int r;
printf("Enter radious");
scanf("%d",&r);
cir=3.14*r*r;
return cir;
}
```

Output:

Enter radius 5

the area of circle 78.500000

Parameter Passing Methods (or) Passing Arguments to Function

 Function is a good programming style in which we can write reusable code thatcan be called whenever required.

 Whenever we call a function, the sequence of executable statements getsexecuted. We can pass some of the information (or) data to the function forprocessing is called a parameter.

 In 'C' Language there are two ways a parameter can be passed to a function.They are

o Call by value

o Call by reference

Call by Value:

 This method copies the value of the actual parameter to the formal parameter of thefunction.

 Here, the changes of the formal parameters cannot affect the actual parameters,because formal parameter are photocopies of the actual parameter.

 The changes made in formal arguments are local to the block of the called function.Once control returns back to the calling function the changes made disappears.

Example Program

```
#include<stdio.h>

#include<conio.h>

void cube(int);

int cube1(int);

void main()

{

int a;

clrscr();

printf("Enter one values");

scanf("%d",&a);
printf("Value of cube function is=%d", cube(a));

printf("Value of cube1 function is =%d", cube1(a ));

getch();

}

void cube(int x)

{

x=x*x*x;

return x;

}

int cube1(int x)

{

x=x*x*x;

return x;

}
```

Output:

Enter one values 3

Value of cube function is 3

Value of cube1 function is 27

Call by reference

 Call by reference is another way of passing parameter to the function.

 Here the address of the argument is copied into the parameter inside the function, theaddress is used to access arguments used in the call.

 Hence, changes made in the arguments are permanent.

 Here pointer is passed to function, just like any other arguments.

Example Program

```
#include<stdio.h>
#include<conio.h>
void swap(int,int);
void main()
{
int a=5,b=10;
clrscr();
printf("Before swapping a=%d b=%d",a,b);
swap(&a,&b);
printf("After swapping a=%d b=%d",a,b);
getch();
}
void swap(int *x,int *y)
{
int *t;
t=*x;
*x=*y;
*y=t;
}
```

Output:

Before swapping a=5 b=10

After swapping a=10 b=5

Nesting of function call in c programming

If we are calling any function inside another function call, then it is known as Nestingfunction call. In other words, a function calling different functions inside is termed as NestingFunctions.

Example:

```
// C program to find the factorial of a number.
#include <stdio.h>
//Nesting of functions
//calling function inside another function
//calling fact inside print_fact_tablefunction
void print_fact_table(int); // function declaration
int fact(int); // function declaration
void main() // main function
{
print_fact_table(5); // function call
}
void print_fact_table(int n) // function definition
{
int i;
for (i=1;i<=n;i++)
printf("%d factorial is %d\n",i,fact(i)); //fact(i)-- function call
}
int fact(int n) // function definition
{
if (n == 1)
return 1;
else
return n * fact(n-1);
}
```

Output:

```
1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 120
```

Recursion

A function calling same function inside itself is called as recursion.

Example: // C program to find the factorial of a number.

```
#include <stdio.h>
int fact(int); // function declaration
void main() // main function
{
printf("Factorial =%d",fact(5)); // fact(5) is the function call
}
int fact(int n) // function definition
{
if (n==1) return 1; else
return n * fact(n-1); // fact(n-1) is the recursive function call
}
```

Output:

Factorial = 120

Discussion:

For 1! , the functions returns 1, for other values, it executes like the one below:When the value is 5, it comes to else part and calculates like this,

= 5 * fact (5-1) = 5 * fact (4)

= 5* 4* fact (4-1) = 5 * 4* fact (3)

= 5* 4* 3* fact (3-1) = 5 * 4* 3* fact (2)

= 5* 4* 3* 2* fact (2-1) = 5 * 4* 3* 2* fact (1)

= 5* 4* 3* 2* 1 (if (n==1) then return 1, hence we get 1)

=120

Example :

```
// A program that contains both nested functions and recursion in it.
// Find the maximum number among five different integers using nested function call
and recursion.
int max(int x,int y) // function defintion
{
return x>y ? x:y; // condition operator is used (exp1?exp2:exp3)
```

```
}
void main() // main function
{
int m;
m=max(max(4,max(11,6)),max(10,5)); //nested, recursive call
of function max
printf("%d",m);
getch();
}
```

Output:

11

Text / Reference Books:

1. Byron S Gottfried, “Programming with C”, Schaum's Outlines, 2 nd Edition, Tata McGrawHill, 2006.
2. Dromey R.G., “How to Solve it by Computer”, Pearson Education, 4 th Reprint, 2007.
3. Kernighan, B.W. and Ritchie, D.M., “The C Programming language”, 2 nd Edition, Pearson Education, 2006.
4. Balaguruswami. E., "Programming in C", TMH Publications, 2003.
5. Yashavant P. Kanetkar, ‘LET US C’, 5 th Edition.2005.
6. Stevens, ‘Graphics programming in C’, BPB Publication, 2006.
7. Subburaj. R , ‘Programming in C’, Vikas Publishing, 1 st Edition, 2000.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF MECHANICAL ENGINEERING

DEPARTMENT OF AUTOMOBILE, AERONUTICAL, MECHATRONICS AND MECHANICAL ENGINEERING

UNIT - III

Programming in C - SCSA1103

ARRAYS AND STRINGS

Arrays: Single and Multidimensional Arrays – Array Declaration and Initialization of Arrays Array as Function Arguments. Strings: Declaration – Initialization and String Handling Functions- Simple programs- sorting- searching – matrix operations. Structure and Union: Definition and Declaration – Nested Structures – Array of Structures – Structure as Function Argument– Function that Returns Structure – Union.

Arrays

So far, we have used only single variable name for storing one data item. If we need to store multiple copies of the same data then it is very difficult for the user. To overcome the difficulty a new data structure is used called arrays.

- An array is a linear and homogeneous data structure
- An array permits homogeneous data. It means that similar types of elements are stored contiguously in the memory under one variable name.
- An array can be declared of any standard or custom datatype.

Example of an Array:

Suppose we have to store the roll numbers of the 100 students the we have to declare 100 variables named as roll1, roll2, roll3, ……. roll100 which is very difficult job. Concept of C programming arrays is introduced in C which gives the capability to store the 100 roll numbers in the contiguous memory which has 100 blocks and which can be accessed by single variable name.

1. C Programming Arrays is the **Collection ofElements**
2. C Programming Arrays is collection of the Elements of the **same datatype.**
3. All Elements are stored in the **Contiguousmemory**
4. All elements in the array are accessed using the subscript variable(index).

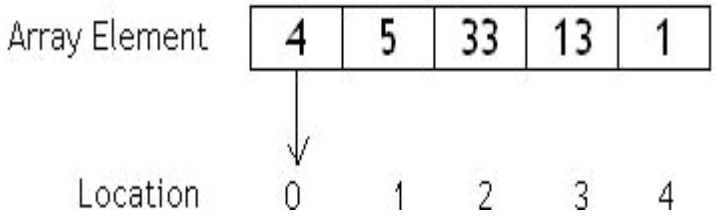


Fig.3.1. Pictorial representation of Array in C

Pictorial representation of C Programming Arrays is shown in figure 3.1

The above array is declared as int a [5];

a[0]=4; a[1]=5; a[2]=33; a[3]=13; a[4] =1;

In the above figure 4, 5, 33, 13, 1 are actual data items. 0, 1, 2, 3, 4 are index variables.

Index or Subscript Variable:

1. Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript variable /index

2. Subscript Variables helps us to identify the item number to be accessed in the contiguous memory.

What is Contiguous Memory?

1. When Big Block of memory is reserved or allocated then that memory block is called as Contiguous MemoryBlock.
2. Alternate meaning of Contiguous Memory is continuousmemory.
3. Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going toreserve contiguous memory.

Contiguous Memory Allocation

1. Two registers are used while implementing the contiguous memory scheme. These registers are base register and limitregister.
2. When OS is executing a process inside the main memory then content of each register are represented as in table 3.1.

Table 3.1. Content of Register

Register	Content of register
Base register	Starting address of the memory location where process execution is happening
Limit register	Total amount of memory in bytes consumed by process

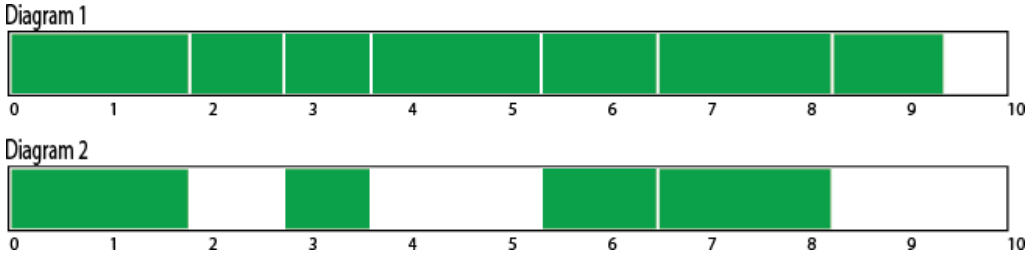


Fig.3.2.Contiguous and Non -contiguous allocation of memory

The figure 3.2 containing e diagram 1 represents the contiguous allocation of memory and diagram 2 represents non- contiguous allocation of memory.

3. When process try to refer a part of the memory then it will firstly refer the base address from base register and then it will refer relative address of memory location with respect to baseaddress.

How to allocate contiguous memory?

1. Using static arraydeclaration.
2. Using alloc ( ) / malloc ( ) function to allocate big chunk of memorydynamically.

Array Terminologies:

Here diagram 1 represents the contiguous allocation of memory and diagram 2 represents non-contiguous allocation of memory.

Size: Number of elements or capacity to store elements in an array. It is always mentioned in square brackets [ ].

Type: Refers to data type. It decides which type of element is stored in the array. It is also instructing the compiler to reserve memory according to the data type.

Base: The address of the first element is a base address. The array name itself stores address of the first element.

Index: The array name is used to refer to the array element. For example:num[x], num is array and x are index. The value of x begins from 0.The index value is always an integer value.

Range: Value of index of an array varies from lower bound to upper bound. For example in num[100] the range of index is 0 to 99.

Word: It indicates the space required for an element. In each memory location, computer can store a data piece. The space occupation varies from machine to machine. If the size of element is more than word (one byte) then it occupies two successive memory locations. The variables of data type int, float, long need more than one byte in memory.

Characteristics of an array:

1. The declaration int a [5] is nothing but creation of five variables of integer types in memory instead of declaring five variables for five values.

2. All the elements of an array share the same name and they are distinguished from one another with the help of the element number.

3. The element number in an array plays a major role for calling each element.

4. Any particular element of an array can be modified separately without disturbing the other elements.

5. Any element of an array a[ ] can be assigned or equated to another ordinary variable or array variable of its type.

6. Array elements are stored in contiguous memory locations.

Array Declaration:

Array has to be declared before using it in C Program. Array is nothing but the collection of elements of similar data types. Table 3.2 and 3.3 represents array declaration and requirements

Syntax: <data type> array name [size1][size2].....[sizen];

Table 3.2. Array Declaration

Syntax Parameter	Significance
Data type	Data Type of Each Element of the array
Array name	Valid variable name
Size	Dimensions of the Array

Array declaration requirements

Table 3.3. Array declaration requirements

Requirement	Explanation
Data Type	Data Type specifies the type of the array. We can compute the size required for storing the single cell of array.
Valid Identifier	Valid identifier is any valid variable or name given to the array. Using this identifier name array can be accessed.
Size of Array	It is maximum size that array can have.

What does Array Declaration tell to Compiler?

1. Type of the Array
2. Name of the Array
3. Number of Dimension
4. Number of Elements in Each Dimension

Types of Array

1. Single Dimensional Array / One Dimensional Array

2. Multi Dimensional Array

Single / One Dimensional Array:

1. Single or OneDimensional array is used to represent and store data in a linear form.
2. Array having only one subscript variable is called **One-Dimensional array**

3. It is also called as **Single Dimensional Array** or **Linear Array**
Single Dimensional Array Declaration and initialization:

Syntax for declaration: <data type><array name> [size];
Examples for declaration: int iarr[3]; **char** carr[20]; **float** farr[3];
Syntax for initialization: <data type><array name> [size] = {val1, val2, …, valn};
Examples for initialization:
int iarr[3] = {2, 3, 4};

char carr[20] = "program";
float farr[3] = {12.5, 13.5, 14.5};

Different Methods of Initializing 1-D Array

Whenever we declare an array, we initialize that array directly at compile time. Initializing 1-D Array is called as compiler time initialization if and only if we assign certain set of values to array element before executing program. i.e. at compilation time.Diagrammatic representation of initializing 1-D array is shown in figure 3.3.

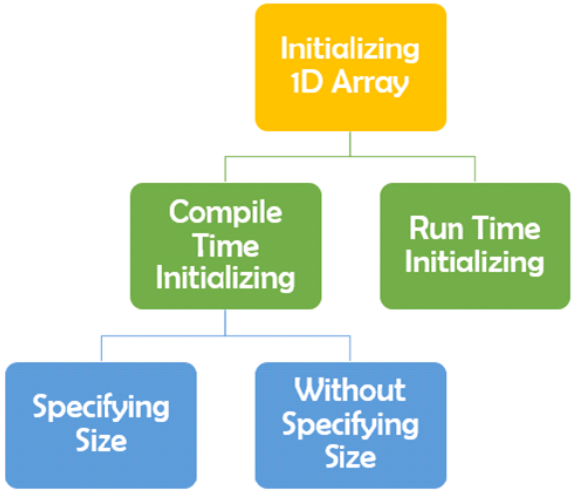


Fig.3.3.Methods of Initializing 1-D Array

Here we are learning the different ways of compile time initialization of an array.

Ways of Array Initializing 1-D Array:

1. Size is Specified Directly
2. Size is Specified Indirectly

Method 1: Array Size Specified Directly

In this method, we try to specify the Array Size directly.

int num [5] = {2,8,7,6,0};
In the above example we have specified the size of array as 5 directly in the initialization statement. Compiler will assign the set of values to particular element of the array.
num[0] = 2; num[1] = 8; num[2] = 7; num[3] = 6; num[4] = 0;
As at the time of compilation all the elements are at specified position So This initialization scheme is Called as "**Compile Time Initialization**". The figure 3.4 shows the graphical representation.

Graphical Representation:

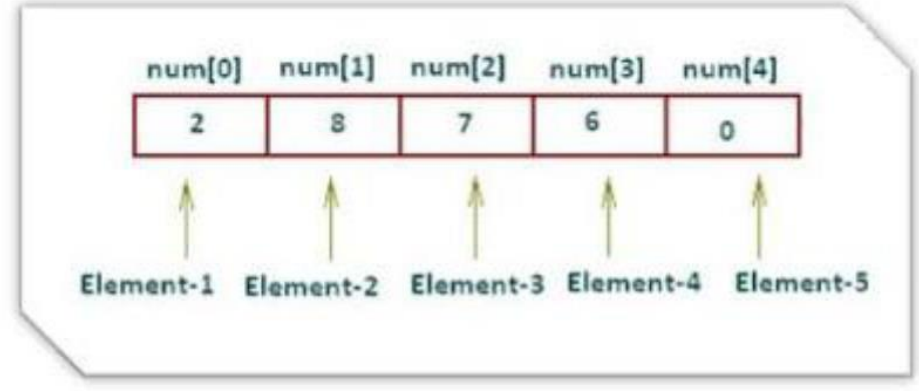


Fig.3.4 Graphical Representation of Array

Method 2: Size Specified Indirectly

In this scheme of compile time Initialization, We do not provide size to an array but instead we provide set of values to the array.
int num[ ] = {2,8,7,6,0};
Explanation:
1. Compiler Counts the Number Of Elements Written Inside Pair of Braces and Determines the Size of An Array.
2. After counting the number of elements inside the braces, The size of array is considered as 5 during complete execution.
3. This type of Initialization Scheme is also Called as "Compile Time Initialization"

Example Program

```
#include <stdio.h>
int main()
int num[] = {2,8,7,6,0};
int i;
for (i=0;i<5;i++) {
printf("\n Array Element num [%d] = %d",i, num[i]); }
return 0; }
```

Output:
Array Element num[0] = 2
Array Element num[1] = 8
Array Element num[2] = 7

Array Element num[3] = 6
Array Element num[4] = 0

Accessing Array

1. We all know that array elements are randomly accessed using the subscript variable.
2. Array can be accessed using array-name and subscript variable written inside pair of square brackets [ ].

Consider the below example of an array shown in figure 3.5.

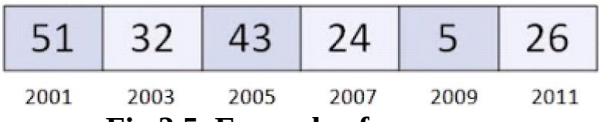


Fig.3.5. Example of an array

In this example we will be accessing array like this
arr[3] = Forth Element of Array
arr[5] = Sixth Element of Array
whereas elements are assigned to an array using below way
arr[0] = 51; arr[1] = 32; arr[2] = 43; arr[3] = 24; arr[4] = 5; arr[5] =26;

Example Program1: Accessing array

```
#include<stdio.h>
#include<conio.h>
void main()
{
int arr[] = {51,32,43,24,5,26};
int i;
for(i=0; i<=5; i++) {
printf("\nElement at arr[%d] is %d",i,arr[i]);
}
getch();
}
```

Output:
Element at arr[0] is 51
Element at arr[1] is 32
Element at arr[2] is 43
Element at arr[3] is 24
Element at arr[4] is 5
Element at arr[5] is 26

How a[i] Works?

We have following array which is declared like int arr[] = { 51,32,43,24,5,26};

As we have elements in an array, so we have track of base address of an array. Things important to access an array are represented in table 3.4.

Table 3.4. Accessing Array

Expression	Description	Example
arr	It returns the base address of an array	Consider 2000
*arr	It gives zeroth element of an array	51
*(arr+0)	It also gives zeroth element of an array	51
*(arr+1)	It gives first element of an array	32

So whenever we tried accessing array using arr[i] then it returns an element at the location*(arr + i) Accessing array a[i] means retrieving element from address (a + i).

Example Program2: Accessing array

```
#include<stdio.h>
#include<conio.h>
void main()
{
int arr[] = {51,32,43,24,5,26};
int i;
for(i=0; i<=5; i++) {
printf("\n%d %d %d %d",arr[i],*(i+arr),*(arr+i),i[arr]);
}
getch();
}
```

Output:

```
51 51 51 51
32 32 32 32
43 43 43 43
24 24 24 24
5 5 5 5
26 26 26 26
```

Operations with One Dimensional Array

1. Deletion – Involves deleting specified elements form an array.
2. Insertion – Used to insert an element at a specified position in an array.
3. Searching – An array element can be searched. The process of seeking specific elements in an array is called searching.
4. Merging – The elements of two arrays are merged into a single one.

5. Sorting – Arranging elements in a specific order either in ascending or in descending order.

Example Programs:

1. C Program for deletion of an element from the specified location from an Array

```
#include<stdio.h>
int main() {
int arr[30], num, i, loc;
printf("\nEnter no of elements:");
scanf("%d", &num);
//Read elements in an array
printf("\nEnter %d elements :", num);
for (i = 0; i<num; i++) {
scanf("%d", &arr[i]); }
//Read the location
printf("\nLocation of the element to be deleted :");
scanf("%d", &loc);
/* loop for the deletion */
while (loc<num) {
arr[loc - 1] = arr[loc];
loc++; }
num--; // No of elements reduced by 1
//Print Array
for (i = 0; i<num; i++)
printf("\n %d", arr[i]);
return (0);
}
```

Output:

```
Enter no of elements: 5
Enter 5 elements: 3 4 1 7 8
Location of the element to be deleted: 3
3 4 7 8
```

2. C Program to delete duplicate elements from an array

```
int main() {
int arr[20], i, j, k, size;
printf("\nEnter array size: ");
scanf("%d", &size);
printf("\nAccept Numbers: ");
for (i = 0; i< size; i++)
scanf("%d", &arr[i]);
printf("\nArray with Unique list: ");
for (i = 0; i< size; i++) {
for (j = i + 1; j < size;) {
if (arr[j] == arr[i]) {
```

```
for (k = j; k < size; k++) {
arr[k] = arr[k + 1]; }
size--; }
else
j++; }
}
for (i = 0; i< size; i++) {
printf("%d ", arr[i]); }
return (0);
}
```

Output:

```
Enter array size: 5
Accept Numbers: 1 3 4 5 3
Array with Unique list: 1 3 4 5
```

3. C Program to insert an element in an array

```
#include<stdio.h>
int main()
 {
 int arr[30], element, num, i, location;
printf("\nEnter no of elements:");
scanf("%d", &num);
for (i = 0; i<num; i++) {
scanf("%d", &arr[i]); }
printf("\nEnter the element to be inserted:");
scanf("%d", &element);
printf("\nEnter the location");
scanf("%d", &location);
//Create space at the specified location
for (i = num; i>= location; i--) {
arr[i] = arr[i - 1]; }
num++;
arr[location - 1] = element;
//Print out the result of insertion
for (i = 0; i<num; i++)
printf("n %d", arr[i]);
return (0);
}
```

Output:

```
Enter no of elements: 5
1 2 3 4 5
Enter the element to be inserted: 6
Enter the location: 2
1 6 2 3 4 5
```

4. C Program to search an element in an array

```
#include<stdio.h>
int main() {
int a[30], ele, num, i;
printf("\nEnter no of elements:");
scanf("%d", &num); printf("\nEnter the values :");
for (i = 0; i<num; i++) {
scanf("%d", &a[i]); }
//Read the element to be searched
printf("\nEnter the elements to be searched :");
scanf("%d", &ele);
//Search starts from the zeroth location
i = 0;
while (i<num&&ele != a[i]) {
i++; }
//If i<num then Match found
if (i<num) {
printf("Number found at the location = %d", i + 1);
}
else {
printf("Number not found"); }
return (0);
}
```

Output:

```
Enter no of elements: 5
11 22 33 44 55
Enter the elements to be searched: 44
Number found at the location = 4
```

5. C Program to copy all elements of an array into another array

```
#include<stdio.h>
int main() {
int arr1[30], arr2[30], i, num;
printf("\nEnter no of elements:");
scanf("%d", &num);
//Accepting values into Array
printf("\nEnter the values:");
for (i = 0; i<num; i++) {
scanf("%d", &arr1[i]); }
/* Copying data from array 'a' to array 'b */
for (i = 0; i<num; i++) {
arr2[i] = arr1[i]; }
//Printing of all elements of array
printf("The copied array is:");
for (i = 0; i<num; i++)
```

```
printf("\narr2[%d] = %d", i, arr2[i]);
return (0);
}
```
Output:
```
Enter no of elements: 5
Enter the values: 11 22 33 44 55
The copied array is: 11 22 33 44 55
```

6. C program to merge two arrays in C Programming

```
#include<stdio.h>
int main() {
int arr1[30], arr2[30], res[60];
int i, j, k, n1, n2;
printf("\nEnter no of elements in 1st array:");
scanf("%d", &n1);
for (i = 0; i< n1; i++) {
scanf("%d", &arr1[i]); }
printf("\nEnter no of elements in 2nd array:");
scanf("%d", &n2);
for (i = 0; i< n2; i++) {
scanf("%d", &arr2[i]); }
i = 0;
j = 0;
k = 0;
// Merging starts
while (i< n1 && j < n2) {
if (arr1[i] <= arr2[j]) {
res[k] = arr1[i];
i++;
k++; }
else {
res[k] = arr2[j];
k++;
j++; }
}
/*Some elements in array 'arr1' are still remaining where as the array 'arr2' is exhausted*/
while (i< n1) {
res[k] = arr1[i];
i++;
k++; }
/*Some elements in array 'arr2' are still remaining where as the array 'arr1' is exhausted */
while (j < n2) {
res[k] = arr2[j];
k++;
j++; }
//Displaying elements of array 'res'
```

```
printf("\nMerged array is:");
for (i = 0; i< n1 + n2; i++)
printf("%d ", res[i]);
return (0);
}
```

```
Enter no of elements in 1st array: 4
11 22 33 44
Enter no of elements in 2nd array: 3
10 40 80
Merged array is: 10 11 22 33 40 44 80
```

Programs for Practice

1 C Program to display array elements with addresses

2 C Program for Reading and printing Array Elements

3 C Program to calculate Addition of All Elements in Array

4 C Program to find Smallest Element in Array

5 C Program to find Largest Element in Array

6 C Program to reversing an Array Elements

1. C Program to display array elements with addresses

```
#include<stdio.h>
#include<stdlib.h>
#define size 10
int main() {
int a[3] = { 11, 22, 33 };
printf("\n a[0],value=%d : address=%u", a[0], &a[0]);
printf("\n a[1],value=%d : address=%u", a[1], &a[1]);
printf("\n a[2],value=%d : address=%u", a[2], &a[2]);
return (0);
}
```

Output:

```
a[0],value=11 : address=2358832
a[1],value=22 : address=2358836
a[2],value=33 : address=2358840
```

2. C Program for Reading and printing Array Elements

```
#include<stdio.h>
int main()
{
int i, arr[50], num;
printf("\nEnter no of elements :");
```

```
scanf("%d", &num);
//Reading values into Array
printf("\nEnter the values :");
for (i = 0; i<num; i++) {
scanf("%d", &arr[i]); }
//Printing of all elements of array
for (i = 0; i<num; i++) {
printf("\narr[%d] = %d", i, arr[i]); }
return (0);
}
```

Output:

Enter no of elements : 5
Enter the values : 10 20 30 40 50
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50

3. C Program to calculate addition of all elements in an array

```
#include<stdio.h>
int main() {
int i, arr[50], sum, num;
printf("\nEnter no of elements :");
scanf("%d", &num);
//Reading values into Array
printf("\nEnter the values :");
for (i = 0; i<num; i++)
scanf("%d", &arr[i]);
//Computation of total
sum = 0;
for (i = 0; i<num; i++)
sum = sum + arr[i];
//Printing of all elements of array
for (i = 0; i<num; i++)
printf("\na[%d]=%d", i, arr[i]);
//Printing of total
printf("\nSum=%d", sum);
return (0);
}
```

Output:

Enter no of elements : 3
Enter the values : 11 22 33
a[0]=11
a[1]=22
a[2]=33

Sum=66

4. C Program to find smallest element in an array

```
#include<stdio.h>
int main() {
int a[30], i, num, smallest;
printf("\nEnter no of elements :");
scanf("%d", &num);
//Read n elements in an array
for (i = 0; i<num; i++)
scanf("%d", &a[i]);
//Consider first element as smallest
smallest = a[0];
for (i = 0; i<num; i++) {
if (a[i] < smallest) {
smallest = a[i]; } }
// Print out the Result
printf("\nSmallest Element : %d", smallest);
return (0);
}
```

Output:

```
Enter no of elements : 5
11 44 22 55 99
Smallest Element : 11
```

5. C Program to find largest element in an array

```
#include<stdio.h>
int main() {
int a[30], i, num, largest;
printf("\nEnter no of elements :");
scanf("%d", &num);
//Read n elements in an array
for (i = 0; i<num; i++)
scanf("%d", &a[i]);
//Consider first element as largest
largest = a[0];
for (i = 0; i<num; i++) {
if (a[i] > largest) {
largest = a[i]; } }
// Print out the Result
printf("\nLargest Element : %d", largest);
return (0);
}
```

Output:

```
Enter no of elements : 5
```

11 55 33 77 22
Largest Element : 77

6. C Program to reverse an array elements in an array

```
#include<stdio.h>
int main()
{
int arr[30], i, j, num, temp;
printf("\nEnter no of elements : ");
scanf("%d", &num);
//Read elements in an array
for (i = 0; i<num; i++) {
scanf("%d", &arr[i]); }
j = i - 1; // j will Point to last Element
i = 0; // i will be pointing to first element
while (i< j) {
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
i++; // increment i
j--; // decrement j
}
//Print out the Result of Insertion
printf("\nResult after reversal : ");
for (i = 0; i<num; i++) {
printf("%d \t", arr[i]); }
return (0);
}
```

Output:
Enter no of elements : 5
11 22 33 44 55
Result after reversal : 55 44 33 22 11

Multi Dimensional Array:

1. Array having more than one subscript variable is called Multi-Dimensional array.

2. Multi Dimensional Array is also called as **Matrix**.

Syntax: <data type><array name> [row subscript][column subscript];
Example: Two Dimensional Arrays
Declaration: Char name[50][20];
Initialization:
int a[3][3] = { 1, 2, 3 5, 6, 7 8, 9, 0};
In the above example we are declaring 2D array which has 2 dimensions. First dimension will refer the row and 2nd dimension will refer the column.

Example: Three Dimensional Arrays

The table 3.5 represents multidimensional array declaration
Declaration: Char name[80][20][40];
The following information are given by the compiler after the declaration
Table 3.5. Multidimensional Array declaration

Example	Type	Array Name	Dimension No.	No. of Elements in Each Dimension
1	integer	roll	1	10
2	character	name	2	80 and 20
3	character	name	3	80 and 20 and 40

Two Dimensional Arrays:

1. Two -Dimensional Array requires **Two Subscript Variables**
2. Two- Dimensional Array stores the values in the form of matrix.
3. One Subscript Variable denotes the "**Row**" of a matrix.
4. Another Subscript Variable denotes the "**Column**" of a matrix.

The table 3.6 represents the row and column matrix of 2D array.

Table 3.6. Declaration of 2D Arrays

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Declaration and use of 2D Arrays:

```
int a[3][4];

for(i=0;i<row,i++)
for(j=0;j<col,j++)
 {
printf("%d",a[i][j]);
}
```

Meaning of Two- Dimensional Arrays:

1. Matrix is having 3 rows (i takes value from 0 to 2)
2. Matrix is having 4 Columns (j takes value from 0 to 3)
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
4. Name of 2-D array is „**a**„ and each block is identified by the row & column number.
5. Row number and Column Number Starts from 0. The table 3.7represents 2Darray declaration.

Two-Dimensional Arrays: Summary with Sample Example:

Table 3.7. 2D Arraydeclaration

Summary Point	Explanation
No of Subscript Variables Required	2
Declaration	a[3][4]
No of Rows	3
No of Columns	4
No of Cells	12
No of for loops required to iterate	2

Memory Representation:

1. 2-D arrays are stored in contiguous memory location **row wise**.
2. 3 X 3 Array is shown below in the first Diagram.
3. Consider **3×3 Array is stored in Contiguous memory** location which starts from 4000.
4. Array element **a[0][0]** will be stored at address **4000** again **a[0][1]** will be stored to next memory location i.e. Elements stored row-wise
5. After **Elements of First Row are stored** in appropriate memory locations, elements of next row get their corresponding memory locations.

	Col 0	Col 1	Col 2
Row 0	1	2	3
Row 1	4	5	6
Row 2	7	8	9

Fig.3.6. Memory representation

6. This is integer array so each element requires 2 bytes of memory.
The memory representation is shown in figure 3.6 and memory location is shown in figure 3.7 and figure 3.8 shows memory allocation.

Basic Memory Address Calculation:

a[0][1] = a[0][0] + Size of Data Type

Element	Memory Location
a[0][0]	4000
a[0][1]	4002
a[0][2]	4004
a[1][0]	4006
a[1][1]	4008
a[1][2]	4010
a[2][0]	4012
a[2][1]	4014

Fig.3.7. Memory location

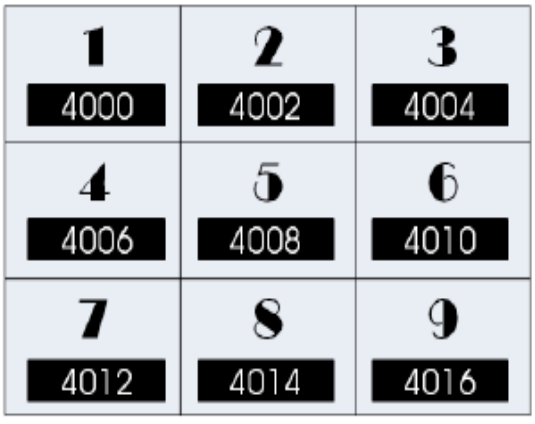


Fig.3.8. Memory Allocation

Initializing 2D Array

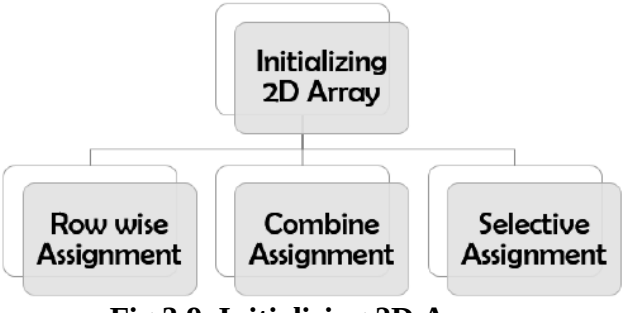


Fig.3.9. Initializing 2D Array

The different methods of initializing 2D array is shown in figure 3.9.

Method 1: Initializing all Elements row wise

For initializing 2D Array we need to assign values to each element of an array using the below syntax.
int a[3][2] = { {1, 4}, {5, 2}, {6, 5} };

Example Program

```
#include<stdio.h>
int main()
{
int i, j;
int a[3][2] = { { 1, 4 }, { 5, 2 }, { 6, 5 } };
for (i = 0; i< 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
```

return 0;
}
Output:
1 4
5 2
6 5
We have declared an array of size 3 X 2, it contains overall 6 elements.

```
Row 1: {1, 4},
Row 2: {5, 2},
Row 3: {6, 5}
We have initialized each row independently
a[0][0] = 1
a[0][1] = 4
```

Method 2: Combine and Initializing 2D Array

Initialize all Array elements but initialization is much straight forward. All values are assigned sequentially and row-wise
int a[3][2] = {1 , 4 , 5 , 2 , 6 , 5 };

Example Program:

```
#include <stdio.h>
int main() {
int i, j;
int a[3][2] = { 1, 4, 5, 2, 6, 5 };
for (i = 0; i< 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
return 0;
}
```

Output:
1 4
5 2
6 5
Method 3: Some Elements could be initialized

```
int a[3][2] = { { 1 }, { 5 , 2 }, { 6 } };
```

Now we have again going with the way 1 but we are removing some of the elements from the array. Uninitialized elements will get default 0 value. In this case we have declared and initialized 2-D array like this

```
#include <stdio.h>
int main() {
int i, j;
int a[3][2] = { { 1 }, { 5, 2 }, { 6 }};
```

```
for (i = 0; i< 3; i++) {
for (j = 0; j < 2; j++) {
printf("%d ", a[i][j]); }
printf("\n"); }
return 0;
}
```

Output:

```
1 0
5 2
6 0
```

Accessing 2D Array Elements:

1. To access every 2D array we requires **2 Subscript variables**.

2. i – Refers the **Row number**

3. j – Refers **Column Number**

4. a[1][0] refers element belonging to **first row and zeroth column**

Example Program: Accept & Print 2×2 Matrix from user

```
#include<stdio.h>
int main() {
int i, j, a[3][3];
// i : For Counting Rows
// j : For Counting Columns
for (i = 0; i< 3; i++)
{
for (j = 0; j < 3; j++)
 {
printf("\nEnter the a[%d][%d] = ", i, j);
scanf("%d", &a[i][j]);
 }
 }
//Print array elements
for (i = 0; i< 3; i++)
 {
for (j = 0; j < 3; j++)
{
printf("%d\t", a[i][j]);
}
printf("\n");
}
return (0);
}
```

How it Works?

1. For Every value of row Subscript , the **column Subscript incremented from 0 to n-1 columns**

2. i.e. For Zeroth row it will accept zeroth, first, second column (a[0][0], a[0][1], a[0][2]) elements

3. In **Next Iteration** Row number will be incremented by 1 and the **column number again initialized to 0.**

4. **Accessing 2-D Array:** a[i][j] Element From ith Row and jth Column

Example programs for practice:

1. C Program for addition of two matrices

2. C Program to find inverse of 3 X # Matrix

3. C Program to Multiply two 3 X 3 Matrices

4. C Program to check whether matrix is magic square or not?

1. C Program for addition of two matrices

```
#include<stdio.h>
int main() {
int i, j, mat1[10][10], mat2[10][10], mat3[10][10];
int row1, col1, row2, col2;
printf("\nEnter the number of Rows of Mat1 : ");
scanf("%d", &row1);
printf("\nEnter the number of Cols of Mat1 : ");
scanf("%d", &col1);
printf("\nEnter the number of Rows of Mat2 : ");
scanf("%d", &row2);
printf("\nEnter the number of Columns of Mat2 : ");
scanf("%d", &col2);
/* before accepting the Elements Check if no of rows and columns of both matrices is equal */
if (row1 != row2 || col1 != col2) {
printf("\nOrder of two matrices is not same ");
exit(0); }
//Accept the Elements in Matrix 1
for (i = 0; i< row1; i++) {
for (j = 0; j < col1; j++) {
printf("Enter the Element a[%d][%d] : ", i, j);
scanf("%d", &mat1[i][j]); } }
//Accept the Elements in Matrix 2
for (i = 0; i< row2; i++)
for (j = 0; j < col2; j++) {
printf("Enter the Element b[%d][%d] : ", i, j);
scanf("%d", &mat2[i][j]); }
//Addition of two matrices
for (i = 0; i< row1; i++)
```

```
for (j = 0; j < col1; j++) {
mat3[i][j] = mat1[i][j] + mat2[i][j];}
//Print out the Resultant Matrix
printf("\nThe Addition of two Matrices is : \n");
for (i = 0; i< row1; i++) {
for (j = 0; j < col1; j++) {
printf("%d\t", mat3[i][j]); }
printf("\n"); }
return (0);
}
```

Output:

```
Enter the number of Rows of Mat1 : 3
Enter the number of Columns of Mat1 : 3
Enter the number of Rows of Mat2 : 3
Enter the number of Columns of Mat2 : 3
Enter the Element a[0][0] : 1
Enter the Element a[0][1] : 2
Enter the Element a[0][2] : 3
Enter the Element a[1][0] : 2
Enter the Element a[1][1] : 1
Enter the Element a[1][2] : 1
Enter the Element a[2][0] : 1
Enter the Element a[2][1] : 2
Enter the Element a[2][2] : 1
Enter the Element b[0][0] : 1
Enter the Element b[0][1] : 2
Enter the Element b[0][2] : 3
Enter the Element b[1][0] : 2
Enter the Element b[1][1] : 1
Enter the Element b[1][2] : 1
Enter the Element b[2][0] : 1
Enter the Element b[2][1] : 2
Enter the Element b[2][2] : 1
The Addition of two Matrices is :
2 4 6
4 2 2
2 4 2
```

2. C Program to find inverse of 3 X 3 Matrix

```
#include<stdio.h>
void reduction(float a[][6], int size, int pivot, int col) {
int i, j;
float factor;
factor = a[pivot][col];
for (i = 0; i< 2 * size; i++)
{
```

```
a[pivot][i] /= factor; }
for (i = 0; i< size; i++) {
if (i != pivot) {
factor = a[i][col];
for (j = 0; j < 2 * size; j++) {
a[i][j] = a[i][j] - a[pivot][j] * factor; } } } }
void main() {
float matrix[3][6];
int i, j;
for (i = 0; i< 3; i++) {
for (j = 0; j < 6; j++) {
if (j == i + 3) {
matrix[i][j] = 1;}
else {
matrix[i][j] = 0; } } }
printf("\nEnter a 3 X 3 Matrix :");
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%f", &matrix[i][j]); } }
for (i = 0; i< 3; i++) {
reduction(matrix, 3, i, i); }
printf("\nInverse Matrix");
for (i = 0; i< 3; i++) {
printf("\n");
for (j = 0; j < 3; j++) {
printf("%8.3f", matrix[i][j + 3]); } } }
```

Output:

```
Enter a 3 X 3 Matrix
1 3 1
1 1 2
2 3 4
Inverse Matrix
2.000 9.000 -5.000
0.000 -2.000 1.000
-1.000 -3.000 2.000
```

3. C Program to Multiply two 3 X 3 Matrices

```
#include<stdio.h>
int main() {
int a[10][10], b[10][10], c[10][10], i, j, k;
int sum = 0;
printf("\nEnter First Matrix : ");
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &a[i][j]); } }
printf("\nEnter Second Matrix :");
```

```
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
scanf("%d", &b[i][j]); } }
printf("The First Matrix is : \n");
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", a[i][j]); }
printf("\n"); }
printf("The Second Matrix is : \n");
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", b[i][j]); }
printf("\n"); }
//Multiplication Logic
for (i = 0; i<= 2; i++) {
for (j = 0; j <= 2; j++) {
sum = 0;
for (k = 0; k <= 2; k++) {
sum = sum + a[i][k] * b[k][j]; }
c[i][j] = sum; } }
printf("\nMultiplication Of Two Matrices : \n");
for (i = 0; i< 3; i++) {
for (j = 0; j < 3; j++) {
printf(" %d ", c[i][j]); }
printf("\n"); }
return (0);
}
```

Output:

```
Enter First Matrix :
1 1 1
1 1 1
1 1 1
Enter Second Matrix :
2 2 2
2 2 2
2 2 2
The First Matrix is :
1 1 1
1 1 1
1 1 1
The Second Matrix is :
2 2 2
2 2 2
2 2 2
Multiplication Of Two Matrices :
6 6 6
6 6 6
```

6 6 6

Multiplication is possible if and only if
i. No. of Columns of Matrix 1 = No of Columns of Matrix 2

ii. Resultant Matrix will be of Dimension – c [No. of Rows of Mat1][No. of Columns of Mat2]

4. C Program to check whether matrix is magic square or not?

What is Magic Square?

1. A magic square is a simple mathematical **game developed during the 1500**.

2. Square is divided into **equal number of rows and columns**.

3. Start filling each square with the number from 1 to num(**wherenum = No of Rows X No of Columns**)

4. You can only **use a number once**.

5. Fill each square so that the **sum of each row is the same as the sum of each column**.

6. In the example shown here, **the sum of each row is 15**, and the sum of each column is also 15.

7. In this Example: The numbers from 1 through 9 is used only once. This is called a **magic square.**

```
#include<stdio.h>
#include<conio.h>
int main() {
int size = 3;
int matrix[3][3]; // = {{4,9,2},{3,5,7},{8,1,6}};
int row, column = 0;
int sum, sum1, sum2;
int flag = 0;
printf("\nEnter matrix : ");
for (row = 0; row < size; row++) {
for (column = 0; column < size; column++)
scanf("%d", &matrix[row][column]); }
printf("Entered matrix is : \n");
for (row = 0; row < size; row++) {
printf("\n");
for (column = 0; column < size; column++) {
printf("\t%d", matrix[row][column]); } }
//For diagonal elements
sum = 0;
for (row = 0; row < size; row++) {
for (column = 0; column < size; column++) {
if (row == column)
sum = sum + matrix[row][column]; } }
//For Rows
for (row = 0; row < size; row++) {
sum1 = 0;
```

```
for (column = 0; column < size; column++) {
sum1 = sum1 + matrix[row][column]; }
if (sum == sum1)
flag = 1;
else {
flag = 0;
break; } }
//For Columns
for (row = 0; row < size; row++) {
sum2 = 0;
for (column = 0; column < size; column++) {
sum2 = sum2 + matrix[column][row]; }
if (sum == sum2)
flag = 1;
else {
flag = 0;
break; } }
if (flag == 1)
printf("\nMagic square");
else
printf("\nNo Magic square");
return 0;
}
```

Output:

```
Enter matrix : 4 9 2 3 5 7 8 1 6
Entered matrix is :
4 9 2
3 5 7
8 1 6
Magic square
Sum of Row1 = Sum of Row2 [Sum of All Rows must be same]
Sum of Col1 = Sum of Col2 [Sum of All Cols must be same]
Sum of Left Diagonal = Sum of Right Diagonal
```

Limitations of Arrays:

Array is very useful which stores multiple data under single name with same data type. Following are some listed limitations of Array in C Programming.

A. Static Data

1. Array is **Static data** Structure is represented in figure 3.10.
2. Memory Allocated during **Compile time**.
3. Once Memory is allocated at Compile Time it cannot be changed during **Run-time**

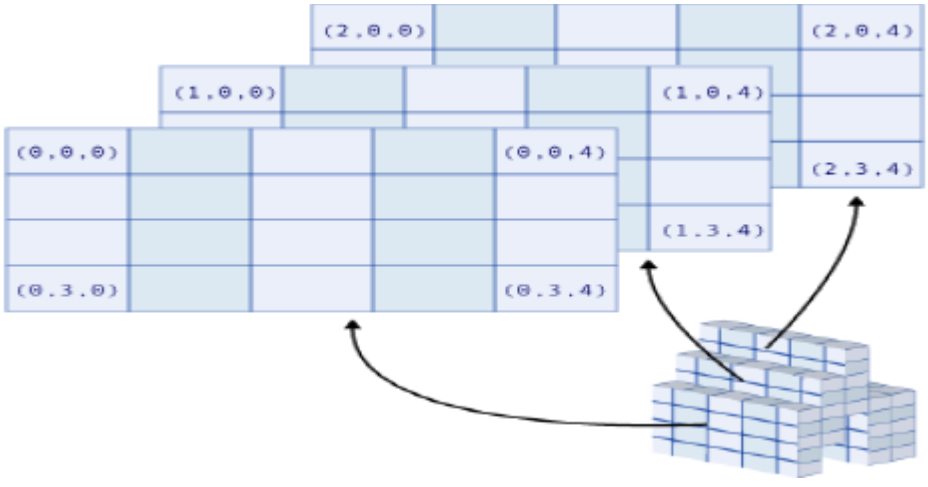


Fig.3.10. Static data structure

B. Can hold data belonging to same Data types

1. Elements belonging to **different data types** cannot be stored in array because array data structure can hold data belonging to same data type.

2. **Example** : Character and Integer values can be stored inside separate array but cannot be stored in single array

C. Inserting data in an array is difficult

1. **Inserting element** is very difficult because before inserting element in an array we have to create empty space by shifting other elements one position ahead.

2. This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non-efficient in case of array with large size.

D. Deletion Operation is difficult

1. Deletion is not easy because the elements are stored in contiguous memory location.

2. Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

E. Bound Checking

1. If we specify the size of array as „N" then we can access elements up to „N-1" but in C if we try to access elements after „N-1" i.e. Nth element or N+1th element then we does not get any error message.

2. Process of checking the extreme limit of array is called Bound Checking and C does not perform **Bound Checking.**

3. If the array range exceeds then we will get garbage value as result.

F. Shortage of Memory

1. Array is Static data structure. Memory can be allocated at compile time only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time.

2. **Shortage of Memory**, if we don't know the size of memory in advance

G. Wastage of Memory

1. **Wastage of Memory**, if array of large size is defined

Applications of Arrays:
Array is used for different verities of applications. Array is used to store the data or values of same data type. Below are the some of the applications of array –

A. Stores Elements of Same Data Type

Array is used to store the number of elements belonging to same data type. int arr[30];
Above array is used to store the integer numbers in an array.
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
Similarly if we declare the character array then it can hold only character. So in short character array can store character variables while floating array stores only floating numbers.

B. Array Used for maintaining multiple variable names using single name

Suppose we need to store 5 roll numbers of students then without declaration of array we need to declare following – int roll1, roll2, roll3, roll4, roll5;
1. Now in order to get roll number of first student we need to access roll1.

2. Guess if we need to store roll numbers of 100 students then what will be the procedure.

3. Maintaining all the variables and remembering all these things is very difficult.

Consider the Array **int** roll[5]; Here we are using array which can store multiple values and we have to remember just single variable name.

C. Array can be used for Sorting Elements

We can store elements to be sorted in an array and then by using different sorting technique we can sort the elements.

Different Sorting Techniques are:
1. Bubble Sort

2. Insertion Sort

3. Selection Sort

4. Bucket Sort

D. Array can perform Matrix Operation

Matrix operations can be performed using the array. We can use 2-D array to store the matrix. Matrix can be multi dimensional.

E. Array can be used in CPU Scheduling

CPU Scheduling is generally managed by Queue. Queue can be managed and implemented using the array. Array may be allocated dynamically i.e at run time. [Animation will Explain more about Round Robin Scheduling Algorithm | Video Animation]

F. Array can be used in Recursive Function

When the function calls another function or the same function again then the current values are stores onto the stack and those values will be retrieving when control comes back. This is similar operation like stack.
Arrays as Function arguments:
Passing array to function:

Array can be passed to function by two ways:
1. Pass Entire array
2. Pass Array element by element

1. Pass Entire array

Here entire array can be passed as a argument to function.
Function gets complete access to the original array.
While passing entire array address of first element is passed to function, any changes made inside function, directly affects the Original value.
Function Passing method : “Pass by Address“

2. Pass Array element by element
Here individual elements are passed to function as argument.
Duplicate carbon copy of Original variable is passed to function.
So any changes made inside function do not affect the original value.
Function doesn‟tget complete access to the original array element.
Function passing method is “Pass by Value“

Passing entire array to function:
Parameter Passing Scheme : Pass by Reference
Pass name of array as function parameter.
Name contains the base address i.e. (Address of 0th element)

Array values are updated in function.
Values are reflected inside main function also.

Example Program #1:

```
#include<stdio.h>
#include<conio.h>
void fun(int arr[ ])
{
int i;
for(i=0;i< 5;i++)
arr[i] = arr[i] + 10;
}
void main( )
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing entire array .....");
fun(arr); // Pass only name of array
for(i=0;i< 5;i++)
printf("\nAfter Function call a[%d] : %d",i,arr[i]);
getch();
}
```

Output :

```
Enter the array elements : 1 2 3 4 5
Passing entire array .....
After Function call a[0] : 11
After Function call a[1] : 12
After Function call a[2] : 13
After Function call a[3] : 14
After Function call a[4] : 15
```

Passing Entire 1-D Array to Function in C Programming:

Array is passed to function completely.

Parameter Passing Method :**Pass by Reference**

It is Also Called "**Pass by Address**"

Original Copy is Passed to Function

Function Body can modify **Original Value.**

Example Program #2:

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
int arr[3] = {1,2,3};
modify(arr);
for(i=0;i<3;i++)
printf("%d",arr[i]);
getch();
}
void modify(int a[3])
{
int i;
for(i=0;i<3;i++)
a[i] = a[i]*a[i];
}
```

Output:

1 4 9

Here “arr” is same as “a” because Base Address of Array “arr” is stored in Array “a”

Alternate Way of Writing Function Header:

void modify(int a[3]) **OR** void modify(int *a)

Passing Entire 2D Array to Function in C Programming:

Example Program #3:

```
#include<stdio.h>
void Function(int c[2][2]);
int main(){
int c[2][2],i,j;
printf("Enter 4 numbers:\n");
for(i=0;i<2;++i)
for(j=0;j<2;++j){
scanf("%d",&c[i][j]); }
Function(c); /* passing multi-dimensional array to function */
return 0;
}
void Function(int c[2][2])
{
/* Instead to above line, void Function(int c[][2]) is also valid */
int i,j;
printf("Displaying:\n");
for(i=0;i<2;++i)
for(j=0;j<2;++j)
printf("%d\n",c[i][j]);
```

}
Output:
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5

Passing array element by element to function:

1. Individual element is passed to function using **Pass By Value** parameter passing scheme

2. An original Array element remains same as Actual Element is never passed to Function. Thus function body cannot modify **Original Value.**

3. Suppose we have declared an array „arr[5]" then its individual elements are arr[0],arr[1]…arr[4]. Thus we need 5 function calls to pass complete array to a function. It is represented in table 3.8.

Consider an array **int** arr[5] = {11, 22, 33, 44, 55};

Table 3.8 Representing passing array element by element to function

Iteration	Element Passed to Function	Value of Element
1	arr[0]	11
2	arr[1]	22
3	arr[2]	33
4	arr[3]	44
5	arr[4]	55

Example Program #1:

```
#include<stdio.h>
#include<conio.h>
void fun(int num)
{
printf("\nElement : %d",num);
}
void main() {
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
```

```
scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");
for(i=0;i< 5;i++)
fun(arr[i]);
getch();
}
```

Output:

```
Enter the array elements : 1 2 3 4 5
Passing array element by element.....
Element : 1
Element : 2
Element : 3
Element : 4
Element : 5
```

Disadvantage of this Scheme:

1. This type of scheme in which we are calling the function again and again but with **different array element is too much time consuming**. In this scheme we need to call function by pushing the current status into the system stack.

2. It is better to pass complete array to the function so that we can save some system time required for pushing and popping.

3. We can also pass the address of the individual array element to function so that function can modify the original copy of the parameter directly.

Example Program #2: Passing 1-D Array Element by Element to function

```
#include<stdio.h>
void show(int b);
void main() {
int arr[3] = {1,2,3};
int i;
for(i=0;i<3;i++)
show(arr[i]);
}
void show(int x)
{
printf("%d ",x);
}
```

Output:

```
1 2 3
```

STRINGS

A string is a sequence of character enclosed with in double quotes (“ ”) but ends with \0. The compiler puts \0 at the end of string to specify the end of the string.To get a value of string variable we can use the two different types of formats.

Using scanf() function as: **scanf(“%s”, string variable);**
Using gets() function as : gets(string variable);

STRING HANDLING FUNCTIONS

C library supports a large number of string handling functions. Those functions are stored under the header file **string.h**in the program. Let us see about some of the string handling functions.

(i) strlen() function

strlen() is used to return the length of the string , that means counts the number of characters present in a string.

Syntax

integer variable = strlen (string variable);

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
int strlength;
clrscr();
printf(,Enter String:‘);
gets(str);
strlength=strlen(str);
printf(,Given String Length Is: %d‘, strlength);
getch();
}
```

Output:

Enter String
Welcome
Given String Length Is:7

(ii) strcat() function

The strcat() is used to concatenate two strings. The second string will be appended to the end of the first string. This process is called concatenation.

Syntax

strcat (StringVariable1, StringVariable 2);

Example:

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

```
void main()
{
char str1[20],str2[20];
clrscr();
printf(,Enter First String:‘);
scanf(,%s‘,str1);
printf(,Enter Second String:‘);
scanf(,%s‘,str2);
printf(, Concatenation String is:%s‘, strcat(str1,str2));
getch();
}
```

Output:

```
Enter First String
Good
Enter Second String
Morning
Concatenation String is: GoodMorning
```

(iii) strcmp() function

strcmp() function is used to compare two strings. strcmp() function does a case sensitive comparison between two strings. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached (whichever occurs first). If the two strings are identical, strcmp( ) returns a value zero. If they‘re not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters.

Syntax

```
strcmp(StringVariable1, StringVariable2);
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20], str2[20];
int res;
clrscr();
printf(,Enter First String:‘);
scanf(,%s‘,str1);
printf(,Enter Second String:‘);
scanf(,%s‘,str2);
res = strcmp(str1,str2);
printf(, Compare String Result is:%d‘,res);
getch();
}
```

Output:

```
Enter First String
Good
```

Enter Second String
Good
Compare String Result is: 0

(iv) strcmpi() function

strcmpi() function is used to compare two strings. strcmpi() function is not case sensitive.

Syntax

strcmpi(StringVariable1, StringVariable2);

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20], str2[20];
int res;
clrscr();
printf(,Enter First String:‘);
scanf(,%s‘,str1);
printf(,Enter Second String:‘);
scanf(,%s‘,str2);
res = strcmpi(str1,str2);
printf(, Compare String Result is:%d‘,res);
getch();
}
```

Output:

Enter First String
WELCOME
Enter Second String
welcome
Compare String Result is: 0

(v) strcpy() function:

strcpy() function is used to copy one string to another. strcpy() function copy the contents of second string to first string.

Syntax

strcpy(StringVariable1, StringVariable2);

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20], str2[20];
int res;
clrscr();
printf(,Enter First String:‘);
scanf(,%s‘,str1);
printf(,Enter Second String:‘);
```

```
scanf(,%s‘,str2);
strcpy(str1,str2)
printf(, First String is:%s‘,str1);
printf(, Second String is:%s‘,str2);
getch();
}
```

Output:

```
Enter First String
Hello
Enter Second String
welcome
First String is: welcome
Second String is: welcome
```

(vi) strlwr () function:

This function converts all characters in a given string from uppercase to lowercase letter.

Syntax

```
strlwr(StringVariable);
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
clrscr();
printf(,Enter String:‘);
gets(str);
printf(,Lowercase String : %s‘, strlwr(str));
getch();
}
```

Output:

```
Enter String
WELCOME
Lowercase String : welcome
```

(vii) strrev() function:

strrev() function is used to reverse characters in a given string.

Syntax

```
strrev(StringVariable);
```

Example:

```
#include<stdio.h>
#include<conio.h> void main()
{
char str[20];
clrscr();
printf(,Enter String:‘);
```

```
gets(str);
printf(,Reverse String : %s', strrev(str));
getch();
}
```
Output:
Enter String
WELCOME
Reverse String :emoclew

(viii) strupr() function:

strupr() function is used to convert all characters in a given string from lower case to uppercase letter.

Syntax

strupr(Stringvariable);

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
clrscr();
printf(,Enter String:');
gets(str);
printf(,Uppercase String : %s', strupr(str));
getch();
}
```
Output:
Enter String
welcome
Uppercase String : WELCOME

STRUCTURES

Arrays are used for storing a group of SIMILAR data items. In order to store a group of data items, we need structures. Structure is a constructed data type for packing different types of data that are logically related. The structure is analogous to the "record" of a database. Structures are used for organizing complex data in a simple and meaningful way.
Example for structures:

Student : regno, student_name, age, address
Book :bookid, bookname, author, price, edition, publisher, year
Employee :employeeid, employee_name, age, sex, dateofbirth, basicpay
Customer :custid, cust_name, cust_address, cust_phone

Structure Definition

Structures are defined first and then it is used for declaring structure variables. Let us see how to define a structure using simple example given below:

```
struct book
{
int bookid;
char bookname[20];
char author[20];
float price;
int year;
int pages;
char publisher[25];
};
```

The keyword "struct" is used for declaring a structure. In this example, book is the name of the structure or the structure tag that is defined by the struct keyword. The book structure has six fields and they are known as structure elements or structure members. Remember each structure member may be of a different data type. The structure tag name or the structure name can be used to declare variables of the structure data type.

The syntax for structure definition is given below:

```
struct tagname
{
Data_type member1;
Data_type member2;
…….
……
};
```

Note:

1. To mark the completion of the template, semicolon is used at the end of the template.
2. Each structure member is declared in a separate line.

Declaring Structure Variables

First, the structure format is defined. Then the variables can be declared of that structure type. A structure can be declared in the same way as the variables are declared. There are two ways for declaring a structure variable.

1) Declaration of structure variable at the time of defining the structure (i.e structure definition and structure variable declaration are combined)

```
struct book
{
int bookid;
char bookname[20];
char author[20];
float price;
int year;
int pages;
char publisher[25]; } b1,b2,b3;
```

The b1, b2, and b3 are structure variables of type struct book.

2) Declaration of structure variable after defining the structure

```
struct book
{
int bookid;
char bookname[20];
char author[20];
float price;
int year;
int pages;
char publisher[25];
};
struct book b1, b2, b3;
```

NOTE:

- Structure tag name is optional.

E.g.

```
struct
{
int bookid;
char bookname[20];
char author[20];
float price;
int year;
int pages;
char publisher[25];
}b1, b2, b3;
```

Declaration of structure variable at a later time is not possible with this type of declaration. It is a drawback in this method. So the second method can be preferred.

- Structure members are not variables. They don"t occupy memory until they are associated with a structure variable.

Accessing Structure Members

There are many ways for storing values into structure variables. The members of a structure can be accessed using a "dot operator" or "period operator".
E.g. b1.author -> b1 is a structure variable and author is a structure member.
Syntax

STRUCTURE_Variable.STRUCTURE_Members

The different ways for storing values into structure variable is given below:

Method 1: Using Simple Assignment Statement

```
b1.pages = 786;
b1.price = 786.50;
```

Method 2: Using strcpy function

```
strcpy(b1.title, ‚Programming in C‘);
strcpy(b1.author, ‚John‘);
```

Method 3: Using scanf function

```
scanf(‚%s \n‘, b1.title);
scanf(‚%d \n‘, &b1.pages);
```

Example

```
#include<stdio.h>
#include<conio.h>
struct book
{
int bookid;
char bookname[20];
char author[20];
float price;
int year;
int pages;
char publisher[25]; };
struct book b1, b2, b3;
main()
{
struct book b1;
clrscr();
printf("Enter the Book Id: ");
scanf("%d", &b1.bookid);
printf("Enter the Book Name: ");
scanf("%s", b1.bookname);
printf("Enter the Author Name: ");
scanf("%s", b1.author);
printf("Enter the Price: ");
scanf("%f", &b1.price);
printf("Enter the Year: ");
scanf("%d", &b1.year);
printf("Enter the Total No. of Pages: ");
scanf("%d", &b1.pages);
printf("Enter the Publisher Name: ");
scanf("%s", b1.publisher);
printf("%d %s %d %f %d %d %s", b1.bookid, b1.bookname, b1.author, b1.price, b1.year, b1.pages,
b1.publisher);
getch();

}
```

Output

```
Enter the Book Id: 786
Enter the Book Name: Programming
Enter the Author Name: John
Enter the Price: 123.50
Enter the Year: 2015
Enter the Total No. of Pages: 649
Enter the Publisher Name: Tata McGraw
786 Programming 2118 123.500000 2015 649 Tata
```

Structure Initialization

Like variables, structures can also be initialized at the compile time.

Example

```
main()
{
struct
{
int rollno;
int attendance;
}
s1={786, 98};
}
```

The above example assigns 786 to the rollno and 98 to the attendance.

Structure variable can be initialized outside the function also.

Example

```
main()
{
struct student
{
int rollno;
int attendance;
};
struct student s1={786, 98};
struct student s2={123, 97};
}
```

Note:

Individual structure members cannot be initialized within the template. Initialization is possible only with the declaration of structure members.

Nested Structures or Structures within Structures

Structures can also be nested. i.e A structure can be defined inside another structure.

Example

```
struct employee
{
int empid;
```

```
char empname[20]; int basicpay;
int da;
int hra;
int cca;
} e1;
```
In the above structure, salary details can be grouped together and defined as a separate structure.
Example
```
struct employee
{
int empid;
char empname[20];
struct
{
int basicpay;
int da;
int hra;
int cca;
} salary;
} e1;
```
The structure employee contains a member named salary which itself is another structure that contains four structure members. The members inside salary structure can be referred as below:
```
e1.salary.basicpay
e1.salary.da;
e1.salary.hra;
e1.salary.cca;
```
However, the inner structure member cannot be accessed without the inner structure variable.
Example
```
e1.basicpay
e1.da
e1.hra
e1.cca
```
are invalid statements

Moreover, when the inner structure variable is used, it must refer to its inner structure member. If it doesn‟trefer to the inner structure member then it will be considered as an error.
Example

e1.salary (salary is not referring to any inner structure member. Hence it is wrong)
Note: C permits 15 levels of nesting and C99 permits 63 levels of nesting.

Array of Structures

A Structure variable can hold information of one particular record. For example, single record of student or employee. Suppose, if multiple records are to be maintained, it is impractical to create multiple structure variables. It is like the relationship between a variable and an array. Why do we go for an array? Because we don‟twant to declare multiple variables and it is practically impossible. Assume that you want to store 1000 values. Do you declare 1000 variables like a1, a2, a3….Upto a1000? Is it easy to

maintain such code ? Is it a good coding? No. It is not. Therefore, we go for Arrays. With a single name, with a single variable, we can store 1000 values. Similarly, to store 1000 records, we cannot declare 1000 structure variables. But we need “Array of Structures”.

An array of structure is a group of structure elements under the same structure variables.
struct student s1[1000];

The above code creates 1000 elements of structure type student. Each element will be structure data type called student. The values can be stored into the array of structures as follows:
s1[0].student_age = 19;

Example

```
#include<stdio.h>
#include<conio.h>
struct book
{
int bookid;
char bookname[20];
char author[20];
};
Struct b1[5];
main()
{
int i;
clrscr();
for (i=0;i<5;i++)
{
printf("Enter the Book Id: ");
scanf("%d", &b1[i].bookid);
printf("Enter the Book Name: ");
scanf("%s", b1[i].bookname);
printf("Enter the Author Name: ");
scanf("%s", b1[i].author);
}
for (i=0;i<5;i++)
{
printf("%d \t %s \t %s \n", b1[i].bookid, b1[i].bookname, b1[i].author);
}
getch();
}
```

Output:

```
Enter the Book Id: 786
Enter the Book Name: Programming
Enter the Author Name: Dennis Ritchie
Enter the Book Id: 101
Enter the Book Name: Java Complete Reference
```

Enter the Author Name: Herbert Schildt
Enter the Book Id: 008
Enter the Book Name: Computer Graphics
Enter the Author Name: Hearn and Baker
786 Programming Dennis Ritchie
101 Java Complete Reference Herbert Schildt
008 Computer Graphics Hearn and Baker

Structure as Function Argument

Example

```
struct sample
{
int no;
float avg;
} a;
void main( )
{
a.no=75;
a.avg=90.25;
fun(a);
}
void fun(struct sample p)
{
printf(,The no is=%d Average is %f',p.no , p.avg);
}
```

Output

The no is 75 Average is 90.25

Function that returns Structure

The members of a structure can be passed to a function. If a structure is to be passed to a called function , we can use any one of the following method.

Method 1 :- Individual member of the structure is passed as an actual argument of the function call. The actual arguments are treated independently. This method is not suitable if a structure is very large structure.

Method 2:-Entire structure is passed to the called function. Since the structure declared as the argument of the function, it is local to the function only. The members are valid for the function only. Hence if any modification done on any member of the structure , it is not reflected in the original structure.

Method 3 :- Pointers can be used for passing the structure to a user defined function. When the pointers are used , the address of the structure is copied to the function. Hence if any modification done on any member of the structure , it is reflected in the original structure.

```
Return data type function name ( structured variable )
Structured Data type for the structured variable;
{
```

```
Local Variable declaration;
Statement 1;
Statement 2;
--------------
-------------
Statement n;
}
```

Example :

```
#include <stdio.h>
struct st
{
char name[20];
int no;
int marks;
};
int main( )
{
struct stx ,y;
int res;
printf(,\n Enter the First Recordʻ);
scanf(,%s%d%dʻ,x.name,&x.no,&x.marks);
printf(,\n Enter the Second Recordʻ);
scanf(,%s%d%dʻ,y.name,&y.no,&y.marks);
res = compare ( x , y );
if (res == 1)
printf(,\n First student has got the Highest Marksʻ);
else
printf(,\n Second student has got the Highest Marksʻ);
}
compare ( structst st1 , struct st st2)
{
if (st1.marks > st2. marks )
return ( 1 );
else
return ( 0 );
}
```

In the above example , x and y are the structures sent from the main ( ) function as the actual parameter to the formal parameters st1 and st2 of the function compare ( ).

Example program (1) – passing structure to function

```
#include<stdio.h>
#include<conio.h>
//-----------------------------------
struct Example
{
int num1;
```

```
int num2;
}s[3];
//-------------------------------------
void accept(struct Example *sptr)
{
printf("\nEnter num1 : ");
scanf("%d",&sptr->num1);
printf("\nEnter num2 : ");
scanf("%d",&sptr->num2);
}
//-------------------------------------
void print(struct Example *sptr)
{
printf("\nNum1 : %d",sptr->num1);
printf("\nNum2 : %d",sptr->num2);
}
//-------------------------------------
void main()
{
int i;
clrscr();
for(i=0;i<3;i++)
accept(&s[i]);
for(i=0;i<3;i++)
print(&s[i]);
getch();
}
```

Output :

```
Enter num1 : 10
Enter num2 : 20
Enter num1 : 30
Enter num2 : 40
Enter num1 : 50
Enter num2 : 60
Num1 : 10
Num2 : 20
Num1 : 30
Num2 : 40
Num1 : 50
Num2 : 60
```

Example program (2) – passing structure to function in C by value:

In this program, the whole structure is passed to another function 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.

```
#include <stdio.h>
#include <string.h>
struct student
{
int id;
char name[20];
float percentage;
};
void func(struct student record);
void main()
{
struct student record;
record.id=1;
strcpy(record.name, "Raju");
record.percentage = 86.5;
func(record);
getch();
}
void func(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
```

Example program (3) Passing structure by value

A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student
{
char name[50];
int roll;
```

```
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration otherwise compiler shows error */
int main()
{
struct student s1;
printf("Enter student's name: ");
scanf("%s",&s1.name);
printf("Enter roll number:");
scanf("%d",&s1.roll);
Display(s1); // passing structure variable s1 as argument
return 0;
}
void Display(struct student stu){
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);
}
```

Output

Enter student's name: Kevin Amla
Enter roll number: 149
Output

Example program (4) – Passing structure to function in C by address:

In this program, the whole structure is passed to another function 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.

```
#include <stdio.h>
#include <string.h>
struct student
{
int id;
char name[20];
float percentage;
};
void func(struct student *record);
void main()
{
struct student record;
record.id=1;
strcpy(record.name, "Raju");
record.percentage = 86.5;
func(&record);
getch();
}
```

```
void func(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
```

Example program (5) Passing structure by reference

The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct distance
{
int feet;
float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d",&dist1.feet);
printf("Enter inch: ");
scanf("%f",&dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d",&dist2.feet);
printf("Enter inch: ");
scanf("%f",&dist2.inch);
Add(dist1, dist2, &dist3);
```

```
/*passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by
reference */
printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
d3->feet=d1.feet+d2.feet;
d3->inch=d1.inch+d2.inch;
if (d3->inch>=12) { /* if inch is greater or equal to 12, converting it to feet. */
d3->inch-=12;
++d3->feet;
}
}
```

Output

```
First distance
Enter feet: 12
Enter inch: 6.8
Second distance
Enter feet: 5
Enter inch: 7.5
Sum of distances = 18'-2.3"
```

Explanation

In this program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument. Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seed in dist3 variable in main function.

Example program(6) 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‟tneed to pass the structure to any function separately.

```
#include <stdio.h>
#include <string.h>
struct student
{
int id;
char name[20];
float percentage;
};
struct student record; // Global declaration of structure
void structure_demo();
int main()
{
record.id=1;
strcpy(record.name, "Raju");
record.percentage = 86.5; structure_demo();
return 0;
}
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
```

Example program(7)Passing Array of Structure to Function in C Programming

Array of Structure can be passed to function as a Parameter.function can also return Structure as return type.Structure can be passed as follow

Example :

```
#include<stdio.h>
#include<conio.h>
//-------------------------------------
struct Example
{
int num1;
int num2;
}s[3];
//-------------------------------------
void accept(struct Example sptr[],int n)
{
int i;
for(i=0;i<n;i++)
```

```
{
printf("\nEnter num1 : ");
scanf("%d",&sptr[i].num1);
printf("\nEnter num2 : ");
scanf("%d",&sptr[i].num2);
}
}
//-------------------------------------
void print(struct Example sptr[],int n)
{
int i;
for(i=0;i<n;i++)
{
printf("\nNum1 : %d",sptr[i].num1);
printf("\nNum2 : %d",sptr[i].num2);
}
}
//-------------------------------------
void main()
{
int i;
clrscr();
accept(s,3);
print(s,3);
getch();
}
```

Output :

```
Enter num1 : 10
Enter num2 : 20
Enter num1 : 30
Enter num2 : 40
Enter num1 : 50
Enter num2 : 60
Num1 : 10
Num2 : 20
Num1 : 30
Num2 : 40
Num1 : 50
Num2 : 60
```

Explanation :

Inside main structure and size of structure array is passed. When reference (i.e ampersand) is not specified in main , so this passing is simple pass by value. Elements can be accessed by using dot [.] operator

Union

The concept of Union is borrowed from structures and the formats are also same. The distinction between them is in terms of storage. In structures , each member is stored in its

own location but in Union , all the members are sharing the same location. Though Union consists of more than one members , only one member can be used at a particular time. The size of the cell allocated for an Union variable depends upon the size of any member within Union occupying more no:- of bytes. The syntax is the same as structures but we use the keyword **union** instead of **struct.**

Example:- the employee record is declared and processed as follows

```
union emp
{
char name[20];
int eno;
float salary;
} employee;
```

where employee is the union variable which consists of the member name,no and salary. The compiler allocates only one cell for the union variable as

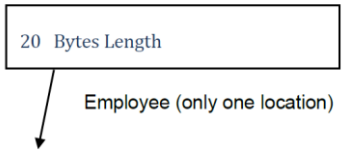


Fig.3.11.Union representation

20 Bytes Length Location / Cell for name,no and salary 20 bytes cell can be shared by all the members because the member name is occupying the highest no:- of bytes. At a particular time we can handle only one member.To access the members of an union , we have to use the same format of structures. Union representation is shown in figure 3.11.

Example program for C union:

```
#include <stdio.h>
#include <string.h>
union student
{
char name[20];
char subject[20];
float percentage;
};
int main()
{
union student record1;
union student record2;
// assigning values to record1 union variable
strcpy(record1.name, "Raju");
strcpy(record1.subject, "Maths");
record1.percentage = 86.50;
printf("Union record1 values example\n");
printf(" Name : %s \n", record1.name);
printf(" Subject : %s \n", record1.subject);
printf(" Percentage : %f \n\n", record1.percentage);
// assigning values to record2 union variable
```

```
printf("Union record2 values example\n");
strcpy(record2.name, "Mani");
printf(" Name : %s \n", record2.name);
strcpy(record2.subject, "Physics");
printf(" Subject : %s \n", record2.subject);
record2.percentage = 99.50;
printf(" Percentage : %f \n", record2.percentage);
return 0;
}
```

Output:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

Explanation for above C union program:

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

Record1 union variable:
“Raju” is assigned to union member “record1.name” . The memory location name is “record1.name” and the value stored in this location is “Raju”.
Then, “Maths” is assigned to union member “record1.subject”. Now, memory location name is changed to “record1.subject” with the value “Maths” (Union can hold only one member at a time).
Then, “86.50” is assigned to union member “record1.percentage”. Now, memory location name is changed to “record1.percentage” with value “86.50”.
Like this, name and value of union member is replaced every time on the common storage space.
So, we can always access only one union member for which value is assigned at last. We can‟t access other member values.
So, only “record1.percentage” value is displayed in output. “record1.name” and “record1.percentage” are empty.

Record2 union variable:
If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
Each union members are accessed in record2 example immediately after assigning values to them.
If we don‟t access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
We can‟t access all members in union at same time but structure can do that.

Example program – Another way of declaring C union:
In this program, union variable "record" is declared while declaring union itself as shown in the below program.

```
#include <stdio.h>
#include <string.h> union student
{
char name[20];
char subject[20];
float percentage;
}record;
int main()
{
strcpy(record.name, "Raju");
strcpy(record.subject, "Maths");
record.percentage = 86.50;
printf(" Name : %s \n", record.name);
printf(" Subject : %s \n", record.subject);
printf(" Percentage : %f \n", record.percentage);
return 0;
}
```

Output:

```
Name :
Subject :
Percentage : 86.500000
```

Note:

We can access only one member of union at a time. We can‟t access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

Difference between structure and union in C:

The difference between structure and union in C is listed in table 3.9

Table 3.9.Difference between structure and union in C

S.no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	`struct student`
	`{`
	`int mark;`
	`char name[6];`
	`double average;`
	`};`
		Union example:
		`union student {`
		`int mark;`
		`char name[6];`
		`double average;`
		`};`

5	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

Assignment Question

1. Create a structure to store the employee number, name, department and basic salary. Create a array of structure to accept and display the values of 10 employees.

Practice Questions

Programs for Practice:

1) Write a C program to initialize an array using functions.

2) Write a C program to interchange array elements of two arrays using functions.

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

4) Write a c program to check whether a given string is a palindrome or not

5) What would be the output of the following programs:

```
main( )
{
char c[2] = "A" ;
printf( "\n%c", c[0] ) ;
printf( "\n%s", c ) ; }
main( )
{
char str1[ ] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’ } ;
char str2[ ] = "Hello" ;
printf( "\n%s", str1 ) ;
printf( "\n%s", str2 ) ;
}
```

(a) main( )

```
{
printf( 5 + "Good Morning " ) ;
}
```

6) Point out the errors,ifany,in the following programs

(a) main( )

```
{
char *str1 = "United" ;
char *str2 = "Front" ;
char *str3 ;
str3 = strcat( str1, str2 ) ;
printf( "\n%s", str3 ) ;
}
```

7) Which is more appropriate for reading in a multi-word string?

` gets( )printf( ) scanf( ) puts( )

8) If the string "Alice in wonder land" is feed to the following

scanf( ) statement, what will be the contents of the arrays
str1, str2, str3 and str4 ?
scanf("%s%s%s%s%s", str1, str2, str3, str4) ;

9) Fill in the blanks:

a. "A" is a _____ while ‟A‟ is a _____.

b. A string is terminated by a _____ character, which is written

as _____.
c. The array char name [10] can consist of a maximum of _____ characters.

1. Write a C program to initialize an array using functions.

```
#include<stdio.h>
int main()
int k, c(), d[ ]={c(),c(),c(),c(),c()};
printf(,\nArray d[] elements are:‘);
for(k=0;k<5;k++)
printf(,%2d‘,d[k]);
return(0);
}
c()
{
int m,n;
m++;
printf(,\nEnter number d[%d] : ‘,m);
scanf(,%d‘,&n);
return(n);
}
```

Output:

```
Enter Number d[1] : 20
Enter Number d[2] : 30
Enter Number d[3] : 40
Enter Number d[4] : 50
Enter Number d[5] : 60
Array d[] elements are: 20 30 40 50 60
```

2. Write a C program to interchange array elements of two arrays using functions.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int read();
void change(int*,int*);
```

```
int x,a[5],b[5];
clrscr();
printf(,Enter 10 Numbers :');
for(x=0;x<10;x++)
{
if(x<5)
a[x]=read();
else
b[x-5]=read();
}
printf(,\nArray A & B');
for(x=0;x<5;x++)
{
printf(,\n%7d%8d',a[x],b[x]);
change(&a[x],&b[x]);
}
printf(,\nNow A & B');
for(x=0;x<5;x++)
{
printf(,\n%7d%8d',a[x],b[x]);
}
}
int read()
{
int x;
scanf(,%d',&x);
return(x);
}
void change(int *a,int *b)
{
int *k;
*a=*a+*b;
*b=*a-*b;
*a=*a-*b;
}
```

Output:

```
Enter 10 Numbers:
0 1 2 3 4 5 6 7 8 9
Array A & B
0 5

1 6

2 7

3 8

4 9

Now A & B
5 0
```

6 1

7 2

8 3

9 4

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

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

Output:

Average age=27.08

Text / Reference Books:

1. Byron S Gottfried, “Programming with C”, Schaum's Outlines, 2 nd Edition, Tata McGrawHill, 2006.
2. Dromey R.G., “How to Solve it by Computer”, Pearson Education, 4 th Reprint, 2007.
3. Kernighan, B.W. and Ritchie, D.M., “The C Programming language”, 2 nd Edition, Pearson Education, 2006.
4. Balaguruswami. E., "Programming in C", TMH Publications, 2003.
5. Yashavant P. Kanetkar, ‘LET US C’, 5 th Edition.2005.
6. Stevens, ‘Graphics programming in C’, BPB Publication, 2006.
7. Subburaj. R , ‘Programming in C’, Vikas Publishing, 1 st Edition, 2000.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF MECHANICAL ENGINEERING

DEPARTMENT OF AUTOMOBILE, AERONUTICAL, MECHATRONICS AND MECHANICAL ENGINEERING

UNIT - IV

Programming in C - SCSA1103

STORAGE CLASS AND POINTERS

Storage Class Specifier: Auto, Extern, Static, & Register. Pointers: The '&' and' *' Operators – Pointers Expressions – Pointers arithmetic- Example Problems. Arrays Using Pointers – Structures Using Pointers– Functions Using Pointer – Function as Arguments – Command Line Arguments

Storage Class Specifier

The scope of the variable specifies the part / parts in which the variable is alive. Depending on the place of the variable declared, the variables are classified into two broad categories as Global and Local variables. In some languages like BASIC, all the variables are global and the values are retained throughout the program. But in C language , the availability of value of the variable depends on the 'storage ' class of variable. In C , there are four types of storage classes. They are

1) Local or Automatic variables
2) Global or External variables
3) Static Variables
4) Register Variables.

1. Automatic variable:

An Automatic variable is a local variable which is declared inside the function. The memory cell is created at the time of declaration statement is executed and is destroyed when the flow comes out of the function. These variables are also called as the internal variables. A variable which is declared inside the function without using any storage class is assumed as the local variable because the default storage class is automatic. A variable can be declared automatic explicitly by using the keyword "auto" as

2. External variable:

```
main ( )
{
auto int x;
…
…..
}
```

External variable is a global variable which is declared outside the function. The memory cell is created at the time of declaration statement is executed and is not destroyed when the flow comes out of the function. The global variables can be accessed by any function in the same program. A global variable can be declared externally in the global variable declaration section.

```
int x = 100; main ( )
{
…..
x = 200; f1 ( );
```

```
f2();                                                    …
…..
}

f1 ( )
{
x = x + 1;
….
…..
}
f2 ( )
{
x = x + 100;
….
…..
}
```

The variable x is declared above all the functions. It can be accessed by all the functions as main( ) , f1( ) and f2( ). The final value of x is 301.

1. Static Variables

These variables are alive throughout the program. A variable can be declared as static by using the keyword " static "as

staic int x ;

A static variable can be initialized only once at the time of declaration. The initialization part is executed only once and retain the remainder value of the program.

Example

```
main ( )
{
void f1( );
f1 ( );
f1 ( );
f1 ( );
}
void f1( )
{
static int x = 0; x = x + 1;
printf("\n The Value of X is %d ",x );
}
```

The above program produces the following output The Value of X is 1
The Value of X is 2 The Value of X is 3.

1. RegisterVariables

If we want to store the variable in a register instead of memory, the variable can be declared as the register variables by using the keyword “register” as

```
register int x;
```

If the variables are stored in the registers , they can be accessed faster than a memory access. So the frequently accessed variables can be declared as the register variables.

Example

```
main ( )
{
register x , y, z; scanf(“%d%d”,&x,&y); z=x+y;
printf(“\n The Output is %d”,z);
}
```

In the above program , all the variables are stored in the registers instead of memory.

ExamplesAuto

```
eg:1 #include<stdio.h>
void main()
{
auto mum = 20 ;
{
auto num = 60 ;
printf("nNum : %d",num);
}
printf("nNum : %d",num);
}
```

Output :

Num : 60

Num : 20

Note :

Two variables are declared in different blocks, so they are treated as different variables.

eg:2

```
#include<stdio.h>
void increment(void);
 void main()
{
increment();
increment();
increment();
increment();
}
void increment(void)
{
auto int i = 0 ;
printf( “%d “, i ) ;
i++;
}
```

Output:

0 0 0 0

Extern

eg:1

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

```
int num =75 ;
void display();
void main()
{
extern int num ;
printf("nNum : %d",num);
display();
}
void display()
{
extern int num ;
printf("nNum : %d",num);
}
```

Output :

Num : 75

Num : 75

Note :

Declaration within the function indicates that the function uses external variable Functions belonging to same source code do not require declaration (no need to write extern). If variable is defined outside the source code, then declaration using extern keyword is required.

eg:2

```
#include<stdio.h>
int x = 10 ;
 void main()
        {
        extern int y;
printf(“The value of x is %d \n”,x);
printf(“The value of y is %d”,y);

}
int y=50;
```

Output:

The value of x is 10 The value of y is 50

Example program for register variable in C:

Static

eg:1

```
#include<stdio.h>
void Check();
int main()
{
Check();
Check();
Check();
        }
void Check()
        {
static int c=0;
printf("%d\t",c);
c+=5;
        }
```

Output

0 5 10

Note:

During first function call, it will display 0. Then, during second function call, variable c will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call. If variable c had been automatic variable, the output would have been:

0 0 0

eg:2

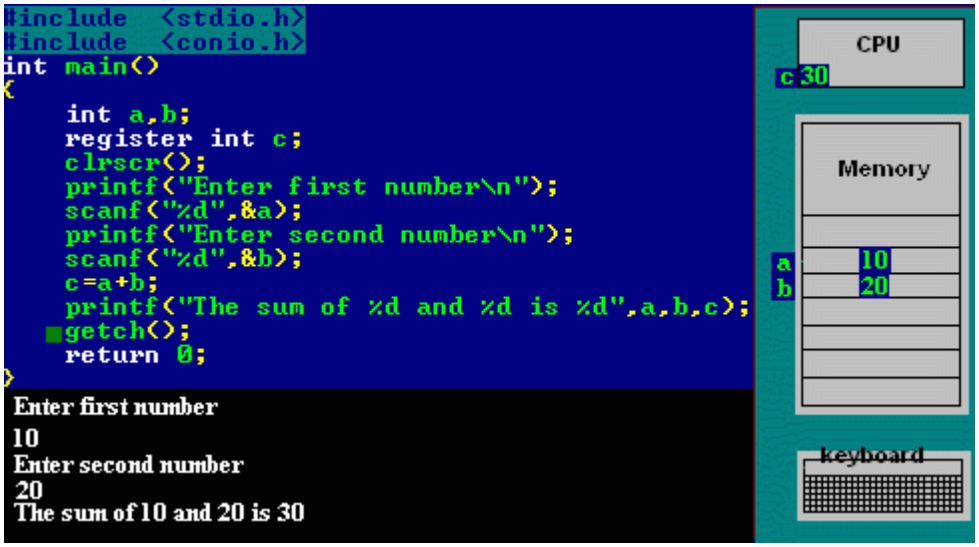
```
#include<stdio.h>
voidincrement(void);
 intmain()
{
increment();
increment();
increment();
```

increment();
return 0;
}
void increment(void)
{
static int i = 0
;printf (“%d “, i
) ; i++;
}
Output: 0 1 23

Register

eg:1

#include<stdio
.h> void
main()
{
int
num1,num2;
register int
sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&num1);
printf("\nEnter the Number 2 :");
scanf("%d",&num2);
sum = num1 +num2;
printf("\nSum of Numbers : %d",sum);
}



eg:2

```
#include <stdio.h>
 int main()
{
register int i;
int arr[5];// declaring array arr[0] = 10;// Initializing array arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
for (i=0;i<5;i++)
{
// Accessing each variable
printf("value of arr[%d] is %d \n", i, arr[i]);
}
return 0;
}
```

Output:

```
value of arr[0] is 10
 value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

POINTERS

Definition:-

A pointer is a variable whose value is the address of another variable. Like any variables, we must declare a pointer variable at the beginning of the program. We can create pointer to any variable type as given in te below examples.
The general format of a pointer variable declaration is as follows:- datatype *pointervariable;
Examples:

```
 int *ip;                  //pointer to an integer variable
float *fp;        //pointer to a float variable
double *dp;     //pointer to a double variable
char *cp;         //pointer to a character variable
```

POINTER OPERATORS:

Operator	Operator Name	Purpose
*	Value at address Operator	Gives Value stored at Particular address
&	Address Operator	Gives Address of Variable

POINTER ADDRESS OPERATOR

1. Pointer address operator is denoted by '&' symbol
2. When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

Take an example – &n - It gives an address of variable n

WORKING OF ADDRESS OPERATOR

Examples:

(1)

```
#include<stdio.h>
void main()
{
int n = 10;
printf("\nValue of n is : %d",n);
printf("\nAddress of n is : %u",&n);
}
```

Output :

Value of n is: 10
Address of n is : 1002

Explanation:

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with %u

printf("\nValue of &n is : %u",&n);

(2)

```
#include<stdio.h>
Voidmain()
{
Int n=20;
Printf("The value of n is: %d",n);
Printf("The address of n is: %u",&n);
Printf("The value of n is: %d",*(&n));
        }
```

OUTPUT:
The value of n is:20
The address of n is:1002
The value of n is:20

Explanation:

In the above program, first printf displays the value of n. The second printf displays the address of the variable n i.e) 1002, which is obtained by using &n(address of variable n). The last printf can be explained as follows,

*(&n) = *(Address of variable n)

=*(1002)

=Value at address 1002

Therefore *(&n)=20

UNDERSTANDING ADDRESS OPERATOR

Initialization of Pointer can be done using following 4 Steps :

i. Declare a Pointer Variable and Note down the Data Type.
ii. Declare another Variable with Same Data Type as that of Pointer Variable.
iii. Initialize Ordinary Variable and assign some value to it.
iv. Now Initialize pointer by assigning the address of ordinary variable to pointer variable.

Below example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{
int a;    // Step 1
 int *ptr;          // Step 2
a = 10; // Step 3
ptr = &a;// Step 4
return(0);
}
```

Explanation of Above Program :

- Pointer should not be used before initialization.
- “ptr” is pointer variable used to store the address of the variable.
- Stores address of the variable ‘a’ .
- Now “ptr” will contain the address of the variable “a” .

Note :

Pointers are always initialized before using it in the program

Consider the following program –

```
#include<stdio.h>
void main()
{
int i = 5; int *ptr; ptr = &i;
printf("\nAddress of i  : %u",&i);
printf("\nValue of ptr is : %u",ptr);
}
```

OUTPUT:

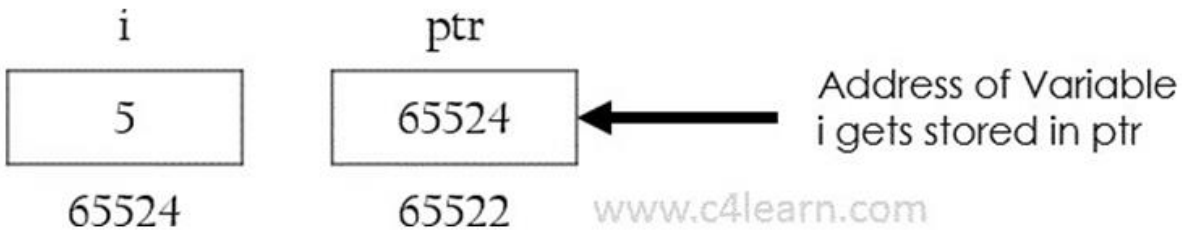
Address of i : 65524

Value of ptris : 65524

After declaration memory map will be like this –

```
 int i = 5;
int *ptr;
```

After assigning the address of variable to pointer, i.e after the execution of this statement – ptr = &i;



Program : accessing value and address of Pointer

/* Program to display the contents of the variable and their address using pointer variable*/

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

```
int i= 3,
*j; j =
&i;

printf("\nAddress of i = %u", &i);
printf("\nAddress of i = %u", j);
printf("\nAddress of j = %u", &j);
printf("\nValue of j = %u", j);
printf("\nValue of i = %d", i);
printf("\nValue of i = %d", *(&i));
printf("\nValue of i = %d", *j);
}
```

Output :

Address of i= 65524
Address of i= 65524
Address of j = 65522
Value of j = 65524
Value of i= 3
Value of i= 3
Value of i=3

Variable	Actual Value
Value of i	3
Value of j	65524
Address of i	65524
Address of j	65522

(1)

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

{
int num,
*intptr;
float x,
*floptr;
char ch,
*cptr;
num=123;
x=12.34;
ch='a';
intptr=&num;
cptr=&ch;
floptr=&x;
printf("Num %d stored at address %u\n",*intptr,intptr);
printf("Value %f stored at address %u\n",*floptr,floptr);
printf("Character %c stored at address %u\n",*cptr,cptr);
}
```

Output :

Num 123 stored at address 1000

Value 12.34 stored at address 2000

Character a stored at address 3000

POINTER EXPRESSIONS

Like any other variables pointer variables can be used in an expression. In general, expressions involving pointer conform to the same rules as other expressions. The pointer expression is a linear combination of pointer variables, variables and operators. Pointer expression gives either numerical output or address output.

Example:

```
y = *p1 * *p2;
sum = sum + *p1;
z = 5 - *p2/*p1;
*p2 = *p2 + 10;/*Pointer expression and pointer arithmetic*/
#include<stdio.h>
void main()
{
int *ptr1,*ptr2;
int a,b,x,y;
a=30;
b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 –b;
y=b - *ptr1/ *ptr2 +a;
printf(“\nAddress of a %u”,ptr1);
printf(“\nAddress of b %u”,ptr2);
printf(“\na=%d, b=%d”,a,b);
printf(“\nx=%d,y=%d”,x,y);
}
```

OUTPUT Address of a 65522

Address of b 65524 a=30 b=6

x=30 y=31

EXPLANATION OF PROGRAM:

In the above example program, ptr1, ptr2 are the pointer variables which are used to store the address of the two variables a and b respectively using the statements ptr1=&a, ptr2=&b. In the pointer expressions which are given below, the value of x and y are calculated as follows,

x=*ptr1+ *ptr2 - b;

=30 + 6 – 6 x=30

y=b - *ptr1/ *ptr2 +a;

=6 - 30/6 + 30

=6 – 5 + 30 y=31

POINTER ASSIGNMENT

We can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example,

#include<stdio.h>

void main()

```
{
int *p1,*p2;
int x=99;
p1=&x;
p2=p1; /*pointer assignment*/
printf("\nValues at p1 and p2: %d %d",*p1,*p2); /*print the value of x twice*/
printf("\nAddresses pointed to by p1 and p2: %u %u",p1,p2); /*print the address of x twice*/
}
```

OUTPUT:

Values at p1 and p2: 99 99

Addresses pointed to by p1 and p2: 5000 5000

EXPLANATION OF PROGRAM:

After the assignment sequence,

p1=&x; p2=p1;

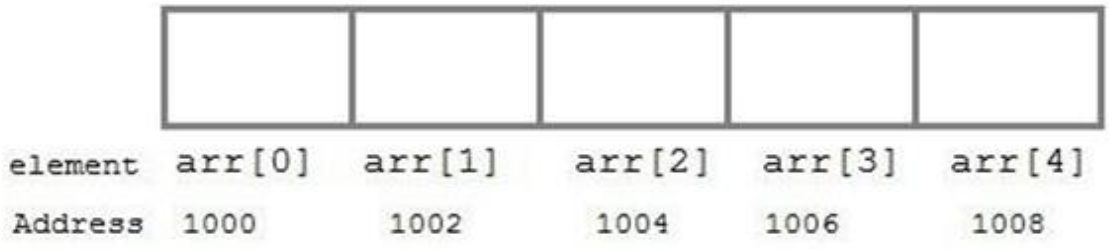
Both p1and p2 point to x. Thus both p1 and p2 refer to the same value.

ARRAYS USING POINTER

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address gives location of the first element which is also allocated by the compiler.

Suppose we declare an array arr, int arr[5]={ 1, 2, 3, 4, 5 };

Assuming that the base address of arr is 1000 and each integer requires two byte, the five element will be stored as follows



Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000. arr is equal to &arr[0] // by default We can declare a pointer of type int to point to the array arr. int *p;
p = arr; or p = &arr[0]; //both the statements are equivalent.
Now we can access every element of array arr using p++ to move from one element to another.

NOTE : You cannot decrement a pointer once incremented. p-- won't work.

POINTER TO ARRAY

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Lets have an example,

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a;       // same as int*p = &a[0]
for (i=0; i<5; i++)
{
printf("%d", *p); p++;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as pointer and print all the values.

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a);⟶ **will print value of a[0] only**

a++;⟶ **Compile time error, we cannot change base address of the array.**

/*Program to print the addresses of array elements */

```
#include <stdio.h>
 void main()
{
char c[4];
int i;
for(i=0;i<4;++i)
{
printf("Address of c[%d]=%x\n",i,&c[i]);
}
}
```

OUTPUT:

```
Address of c[0]=28ff44
Address of c[1]=28ff45
Address of c[2]=28ff46
Address of c[3]=28ff47
```

Notice, that there is equal difference (difference of 1 byte) between any two consecutive elements of array.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array, i.e. using my_array[0] through my_array[5]. But, we could alternatively access them via a pointer as follows:

```
int *ptr;
ptr = &my_array[0];   /* pointer points to the first integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
#include <stdio.h>
 int main(void)
{
int my_array[] = {1,23,17,4,-5,100};
int *ptr;
int i;
ptr = &my_array[0] ; /* point pointing to the first element of the array */
printf("\n\n");
for (i = 0; i< 6; i++)
{
printf("my_array[%d] = %d ",i,my_array[i]); /*<-- A */
printf("ptr + %d = %d\n",i, *(ptr + i));          /*<-- B */
}
return 0;
}
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the new pointer. Change line B to read:

```
printf("ptr + %d = %d\n",i, *ptr++); and run it again. then change it to:
printf("ptr + %d = %d\n",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

ptr = my_array;
to achieve the same result.

/* Example program to print the array elements using pointer */

```
#include <stdio.h>
int main()
{
int data[5],
i;
printf("Enter elements: ");
for(i=0;i<5;++i)
scanf("%d",data[i]);
printf("You entered: ");
for(i=0;i<5;++i)
printf("%d\n",*(data+i));
 return 0;
}
```

Output

```
Enter elements: 1
2
3
5
4
You entered: 1
2
3
5
4
```

/* Program to find sum of array elements using pointer */

```
#include<stdio.h>
 #include<conio.h>
void main()
{
int numArray[10];
int i, sum = 0;
int *ptr;
printf("\nEnter 10 elements : ");
for (i = 0; i< 10; i++)
scanf("%d", &numArray[i]);

ptr = numArray;

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

```
{
        sum = sum + *ptr;
        ptr++;
      }
    printf("The sum of array elements :
%d", sum);
}
```

OUTPUT

Enter 10 elements : 11 12 13 14 15 16 17 18 19 20 The sum of array elements is 155

EXPLANATION OF PROGRAM:

Accept the 10 elements from the user in the array.

```
for (i = 0; i< 10; i++)
scanf ("%d", &num Array[i]);
```

We are storing the address of the array into the pointer.

```
ptr=num Array;          /* a=&a[0]*/
```

Now in the for loop we are fetching the value from the location pointer by pointer variable. Using De-referencing pointer we are able to get the value at address.

```
r (i = 0; i< 10; i++)
 sum = sum + *ptr;
ptr++;
}
```

Suppose we have 2000 as starting address of the array. Then in the first loop we are fetching the value at 2000. i.e

```
sum = sum + (value at 2000)
= 0 + 11
= 11
```

In the Second iteration we will have following calculation –

```
sum = sum + (value at 2002)
= 11 + 12
= 23
```

Pointer example-1

```
#include<stdio.h>
#include <math.h>
main()
{
int num [ ] = { 10,20,30,40,50 }
print ( &num, 5, num);
}
print ( int *j, int n, int b[5])
{
int i;
for(i=0;i<=4;i++){
printf( " %u %d %d %u \n ", &j[i]  ,  *j , *(b+i)  ,  &b);
j++;
```

```
}
}
```

In this example we have a single dimensional array num and a function print . We are passing, the address to the first element of the array, the number of elements and the array itself, to this function. When the function receives this arguments, it maps the first one to another pointer j and the array num is copied into another array b . (The type declarations are made here itself. Note that these declarations can also be given just below this line). j is now a pointer to the array b.

Inside the function we are printing out the address of the array element and the value of the array element in two ways. One using the pointer j and the other using the array b. If we compile and run this code we get the following output,

```
3221223408 10 10 3221223376

3221223416 20 20 3221223376

3221223424 30 30 3221223376

3221223432 40 40 3221223376
```

Note that as we increment j it points to the successive elements of the array. We can get both the address of the array elements and the value stored there using this. However the array name, which acts also as the pointer to its base address, is not able to give us the address of its elements. Or in other words, the array name is a constant pointer. Also note that while j is points to the elements of the array num, b is pointing to its copy.

Next we have an example that uses a two dimensional array. Here care should be taken to declare the number of columns correctly.

Pointer example-2

```
#include <stdio.h>
 #include <math.h>
main()
{
int arr[ ][3] = {{11,12,13}, {21,22,23},{31,32,33},{41,42,43},{51,52,53}};
```

```
        int I , j ;
        int *p , (*q) [3], *r ; p
        = (int *) arr ;
        q = arr;
        r = (int *) q ;
        printf( " %u %u %d %d %d  %d \n  ", p ,  q  ,  *p , *(r)  ,  *(r+1),  *(r+2)); p++ ;
        q++ ;
        r = (int *) q ;
        printf( " %u %u %d %d %d %d \n ", p , q , *p , *(r) , *(r+1), *(r+2));


}
```

Here we have a pointer p and a pointer array q. The first assignment statement is to make the pointer p points to the array arr. While assigning, we also declare the type of the variable arr. Note that variables on both side of this statement should have the same type. Next line is a similar statement, now with q and arr. Since q is a pointer array, the array can be directly assigned to it and there is no need for specifying the type of the variable. In the next line we make the pointer r to point to the pointer array q . Then we will print out the different values. Here is what we get from this,

```
3221223344  3221223344  11 11 12 13
```

ARRAY OF POINTERS

Just like array of integers or characters, there can be array of pointers too. An array of pointers can be declared as :

<datatype> *<pointername> [number-of-elements];

For example :

```
char *ptr[3];
```

The above line declares an array of three-character pointers. Let’s take a working example:

```
#include<stdio.h>

 int main(void)
{
char *p1 = "Himanshu";
char *p2 = "Arora";
 char *p3 ="India";

char *arr[3];

arr[0] = p1; arr[1] = p2;
arr[2] = p3;

printf("\n p1 = [%s]\n",p1);
printf("\n p2 = [%s]\n",p2);
printf("\n p3 = [%s]\n",p3);

printf("\n arr[0] = [%s]\n",arr[0]);
printf("\n arr[1] = [%s]\n",arr[1]);
printf("\n arr[2] = [%s]\n",arr[2]);

return 0;
}
```

In the above code, we took three pointers pointing to three strings. Then we declared an array that can contain three pointers. We assigned the pointers ‘p1′, ‘p2′ and ‘p3′ to the 0,1 and 2 index of array.

Let’s see the output :

So we see that array now holds the address of strings.

Let us consider the following example, which makes use of an array of 3 integers:

```
int main ()
{
int  var[] = {10, 100, 200};
int i;

for (i = 0; i < 3; i++)
{
printf("Value of var[%d] = %d\n", i, var[i] );
}
return 0;
}
```

OUTPUT:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[3];
```

This declares ptr as an array of 3 integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
int main ()
{
int  var[] = {10, 100, 200};
int i, *ptr[3];
ptr[0] = &var[0]; /* assign the address of 1st integer element */
ptr[1] = &var[1]; /* assign the address of 2nd integer element */
 ptr[2] = &var[2]; /* assign the address of 3rd integer element */
for ( i = 0; i < 3; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

OUTPUT:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Pointer and Function
Function Pointer

```
#include <stdio.h>
void subtractAndPrint(int x, int y);
void subtractAndPrint(int x, int y)
{
 int z = x - y;
printf("Simon says, the answer is: %d\n", z);
}
int main()
{
void (*sapPtr)(int, int) = subtractAndPrint;
```

```
 (*sapPtr)(10, 2);
sapPtr(10, 2);
}
```

The pointer can be used as an argument in functions. The arguments or parameters to the function are passed in two ways.
Call by value
Call by reference
In 'C Language there are two ways that the parameter can be passed to a function they are
Call by value
Call by reference

Call by Value:
This method copies the value of actual parameter into the formal parameter of the function. The changes of the formal parameters cannot affect the actual parameters, because formal arguments are photocopy of the actual argument.
The changes made in formal argument are local to the block of the called functions. Once control return back to the calling function the changes made disappear.

Example:

```
#include<stdio.h>
#include<conio.h>
void cube(int);
int cube1(int);
void main()
{
int a;
clrscr();
printf("Enter one values");
scanf("%d",&a);
printf("Value of cube function is=%d", cube(a));
printf("Value of cube1 function is =%d", cube1(a ));
getch();
}
void cube(int x)
{
x=x*x*x*; return x;
}

int cube1(int x)
{
x=x*x*x*; return x;
}
```

Output:

Enter one values 3

Value of cube function is 3
Value of cube1 function is 729

Call by reference

- Call by reference is another way of passing parameter to the function.
- Here the address of argument are copied into the parameter inside the function, the address is used to access arguments used in the call.
- Hence changes made in the arguments are permanent.

- Here pointer are passed to function, just like any other arguments.

Example:-

```
#include<stdio.h>
 #include<conio.h>
void swap(int,int);
void main()
{
int a=5,b=10;
clrscr();
printf("Before swapping a=%d b=%d",a,b);
swap(&a,&b);
printf("After swapping a=%d b=%d",a,b);
getch();
}
void swap(int *x,int *y)
{
int *t; t=*x;
*x=*y;
*y=t;
}
```

Output:

Before swapping a=5 b=10
After swapping a=10 b=5

Function Returning Pointer

A function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in c, we can also force a function to return a pointer to the calling function.

Program:

```
int *larger(int*,int*);
 void main()
{
int a=10; int b=20;
```

Output:

20

```
 int *p;
p =larger(&a,&b);
printf("%d",*p);
}
int *larger(int *x, int *y)
{
if(*x>*y)
return(x);

else
return(y);
}
```

STRUCTURE USING POINTER

```
struct name
{
member1; member2;
};
```

-------- Inside function -------

```
struct name *ptr;
```

Here, the pointer variable of type **struct name** is created.

Structure's member through pointer can be used in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

Consider an example to access structure's member through pointer.

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

```
int a; float b;
};
int main()
{
struct name *ptr,p;
ptr=&p;          /* Referencing pointer to memory address of p */ printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

In this example, the pointer variable of type **struct name** is referenced to the address of *p*. Then, only the structure member through pointer can can accessed.

Structure pointer member can also be accessed using -> operator.

(*ptr).a is same as ptr->a (*ptr).b is same as ptr->b

ACCESSING STRUCTURE MEMBERS WITH POINTER

To access members of structure with structure variable, we used the dot operator.But when we have a pointer of structure type, we use arrow -> to access structure members.

```
struct Book
{
char name[10];
 int price;
}
int main()
{
struct Book b;
struct Book* ptr = &b;
```

```
ptr->name = "Dan Brown";    //Accessing Structure Members ptr->price = 500;
}
```

Example program for C structure using pointer:

In this program, “record1″ is normal structure variable and “ptr” is pointer structure variable. As you know, Dot(.) operator is used to access the data using normal structure variable and arrow(->) is used to access data using pointer variable.

```
#include <stdio.h>
 #include <string.h>
struct student
{
int id;
char name[30];
 float percentage;
};
int main()
{
int i;
struct student record1 = {1, "Raju", 90.5};
struct student *ptr;
ptr = &record1;
printf("Records of STUDENT1: \n");
printf("Id is: %d \n", ptr->id);
printf("Name is: %s \n", ptr->name);
printf("Percentage is: %f \n\n", ptr->percentage);
 return 0;
}
```

Output:

```
Records of STUDENT1:
Id is: 1
Name is: Raju
```

Text / Reference Books:

1. Byron S Gottfried, “Programming with C”, Schaum's Outlines, 2 nd Edition, Tata McGrawHill, 2006.

2. Dromey R.G., “How to Solve it by Computer”, Pearson Education, 4 th Reprint, 2007.

3. Kernighan, B.W. and Ritchie, D.M., “The C Programming language”, 2 nd Edition, Pearson Education, 2006.

4. Balaguruswami. E., "Programming in C", TMH Publications, 2003.

5. Yashavant P. Kanetkar, ‘LET US C’, 5 th Edition.2005.

6. Stevens, ‘Graphics programming in C’, BPB Publication, 2006.

7. Subburaj. R , ‘Programming in C’, Vikas Publishing, 1 st Edition, 2000.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF MECHANICAL ENGINEERING

DEPARTMENT OF AUTOMOBILE, AERONUTICAL, MECHATRONICS AND MECHANICAL ENGINEERING

UNIT - V

Programming in C - SCSA1103

MEMORY MANAGEMENT AND FILES

Contents

DMA functions: malloc (), calloc (), sizeof (), free () and realloc (). Pre-processor directives. File management: File operations - opening & closing a file, Read and write binary files ,input and output statements, Control statements.

Static and Dynamic Memory Allocation in C

It is the procedure to allocate memory cells to the variables. There are two memory allocation methods. They are,

- Static memory allocation
- Dynamic memory allocation

Static Memory Allocation

The process of allocating memory space at compile time is known as static memory allocation. In static memory allocation size is fixed.

Example: Arrays

int a[5];

The above declaration of an array allocates spaces for storing 5 int Examples in the memory. Static Memory allocation has the following Disadvantages:

- Size is not expandable
- Insertion is difficult(Time consuming process)
- It requires a lot of data movement taking more time.
- Deletion is also difficult.

Dynamic Memory Allocation

The process of allocating memory space at run time known as dynamic memory allocation. In dynamic memory allocation size (no of cells is not allocated) is not fixed.

Example: Linked List

Linked List:

A linked list is collection of more than one nodes linked together. Each node has an element. A linked list has the following characteristics

- To store each element, we use a node.
- A node consists of data (elements) and link field.

- If there is only one link field, it is called as singly linked list.
- If it has 2 links, it is doubly linked list.
- The nodes need not be stored continuously in the memory.

The advantages of linked list are

- Size is not fixed.
- Insertion is easy which does not require any data movement.
- Deletion is also easy.

Dynamic Memory Allocation Functions: (malloc(), calloc(), sizeof(), free() and realloc())

DMA predefined functions are available in c library. These are used to allocating and reallocating the memory space in memory.DMA functions are available through the header file is stdlib.h and alloc.h.so u must include this library in order to use them.

(i) malloc( )

This function is used to allocate memory dynamically. Use the malloc function to allocate a single block of memory space of variable in specified size.

If there is not enough memory available it will return NULL.

Syntax:

pointer_variable =(type cast*)malloc(size in bytes);

typecast is a datatype. It will allocate the memory space with size of byte.

Example1:

a = (char*)malloc(10);

It will occupies10 bytes memory and assign of first byte to a.

Example2:

a=(int*)malloc(40*sizeof(int));

Size of pointer is how many bytes it takes to store the pointer, not the size of the memory block the pointer is pointing at.

(ii) realloc( )

It is necessary to alter the previously allocated memory. i.e.,to add additional memory or to reduce as and when required. Before using this statement,the user must allocate some memory previously by using the malloc( ) and calloc( )function.

Syntax:

pointer_variable =realloc(pointer variable , new size);

Example : Program to altering the allocated memory.

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
main()
{
char*p;
clrscr( );
p=(char*)malloc(6);
strcpy(p,"MADRAS");
printf("memory contains:%s\n",p);
 p=(char*)realloc(p,7);
strcpy(p,"CHENNAI");
printf("memory contains:%s\n",p);
 free(p);
getch( );
}
```

Output:

memory contains : MADRAS memory contains :CHENNAI

(iii) calloc( )

This function is used to allocates multiple block of memory space of specified size and each block contains same size and initializes them with zeros.

Syntax:

pointer_variable =(type cast*)calloc(n,elementsize);

(iv) Example

a = (int *)calloc(10,sizeof(int)*2);

It occupies 10 blocks each of the 4 bytes memory and assign the address of first byte of fist block top.

Example Program:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
int *p = (int*)calloc(2,sizeof(int)*2);
clrscr();
for(int i=0;i<4;i++)
p[i]=i;
printf("Memory Allocated Dynamically for 2 block each has 4 bytes");
for(i=0;i<4;i++)
printf("%d",p[i]);
 realloc(p,sizeof(int)*6);
for( i=0;i<6;i++)
p[i]=i;
printf("Memory Allocated Dynamically for 2 block each has 6 bytes");
for( i=0;i<6;i++)
printf("%d",p[i]);
getch();
}
```

sizeof( )

It is an operator to find the size of the data-type or variable in terms of bytes.

Syntax: int x=sizeof(data_type/variable);

Example : Program Depicting the Use of Function sizeof( )in C Programming

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

```
char p;
clrscr( );
printf("%d", sizeof(p));
getch();
}
```

Output:

1

Explanation: We know that a character variable takes 1 byte memory. Here, p is a character variable and sizeof(p) is 1(byte).

Example : Program Depicting the Use of Function sizeof()in C Programming

```
#include<stdio.h>
#include<conio.h>
void main()
{
intp[5];
clrscr();
printf("%d", sizeof(p));
getch();
}
```

Output:

10

Explanation: We know that a intExampleer occupies 2 bytes in the memory. Here, p is a array of 5 elements and sizeof(p) is 10 bytes (5 elements X 2 byes each = 10 bytes).

free( )

It is used to delete an existing memory space

Syntax: free(p);

Example: Sum of array elements using pointer

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

```
void main()
{
int *a,i,n,result=0;
 clrscr();
printf("Enter the no. of elements to be stored in an array :");
scanf("%d",&n);
a=(int *)malloc(n*sizeof(int));
 printf("\nEnter the elements :");
 for(i=0;i<n;i++)
scanf("%d",a+i);
for(i=0;i<n;i++) result+=*(a+i);
printf("\nThe sum of elements in an array is :%d",result);
 free(a);
getch();
}
```

Output:

Enter the no. of elements to be stored in an array: 5

Enter the elements: 2 20 25 35 18

The sum of elements in an array is : 100

PREPROCESSOR DIRECTIVES

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the preprocessor. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive ends. No semicolon is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

Simply a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation.

There are different types of directives. The important directives are

1. Macro Substitution Directives
2. File inclusion directives
3. Compiler Control directives

Macro Substitution Directives

#define Preprocessor defines a constant/identifier and a value that is substituted for identifier/constant each time it is encountered in the source file. Generally macro-identifiers/constant defined using #define directive are written in the capital case to distinguish it from other variables. Constants defined using #define directive are like a name-value pair.

These directives are used for

- assigning a constant to an identifier
- Assigning a symbol to an identifier
- assigning an expression to a identifier
- assigning a name to a function declaring a function The general format of the simple macro directive is

Examples:

#define identifier constant / symbol / function.

#define ALPHA 1000

#define LT <=

#define INPUT scanf

#define PI 3.14

Example 1:

```
#include<stdio.h>
 #define PI 3.14
Void main()
{
float radius, area;
printf(“Enter the radius of the circle:”);
scanf(“%f”, &radius);
area = PI * radius * radius;
printf(“\n Area of the Circle is %f”, area);
}
```

Output:

```
Enter the radius of the circle: 4
Area of the Circle is 50.24000
```

Example 2:

```
#include<stdio.h>
 #include<conio.h>
#define SQUARE(x) x*x
void main()
{
int x;
 clrscr();
printf("\nEnter the side of the square:");
 scanf("%d",&x);
printf("\nArea of the square is %d", SQUARE(x));
getch();
```

}

Output:

Enter the side of the square: 4

Area of the square is 16

File Inclusion Directives

It is used for linking header file or another C file to the current C File to call the library and user defined functions. It has two formats. They are

Format (i): # include <filename with extension >

Format (ii):# include “filename with extension ”

The format (i) is used for linking the Header file to the current C File. A Header file is a file which has the collection of library functions.

The format (ii) is used for linking another C file to the current C File where the C file has one or more library functions.

Examples:

#include <string.h>

// to call the string functions

#include <math.h>

// To call the mathematical functions

#include “p2.c”

// To call one or more user defined functions in p2.c in the current C file.

Compiler Control Directives

These directives are the special directives used for controlling the flow of execution.

These directives are used for many situations.

Example 1:

```
#include <stdio.h>
#define RAJU 100
int main()
{
#ifndef PINKY
{
printf("PINKY is not defined. So, now we are going to " \ "define here\n");
#define PINKY 300
}
#else
printf("PINKY is already defined in the program”);

#endif return 0;

}
```

Output:

PINKY is not defined. So, now we are going to define here

FILE

A file is a place on the disk where a group of related data is stored. C supports a number of functions to perform basic file operations, which include

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file and
- Closing a file

Function Name	Operation

fopen() Creates a new file for use,Opens a new existing file for use

fclose() Closes a file which has been opened for use

getc() Reads a character from a file

putc() Writes a character to a file

fprintf() Writes a set of data values to a file

fscanf() Reads a set of data values from a file

getw() Reads a integer from a file

putw() Writes an integer to the file

fseek() Sets the position to a desired point in the file

ftell() Gives the current position in the file

rewind() Sets the position to the begining of the file

Defining a file

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

- Fielname
- data structure
- purpose

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

Examples

Input.data

store PROG.C

Student.c

Text.out

Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined datatype.

When we open a file, we must specify what we want to do with the file. Example, we may write data to the file or read the already existing data.

Example:

FILE *fp; Opening a file: The general format

FILE *fp; fp=fopen(“filename”,”mode”);

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening the file. The mode does this job.

r open the file for read only.

w open the file for writing only.

a open the file for appending data to it.

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

1. When the mode is ‘writing’, a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.

2. When the purpose is “appending”, the file is opened with the current contents safe. A file with the specified name is created if the file does not exists.

3. If the purpose id “reading”, and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

Consider the following statements:

FILE *p1, *p2; p1=fopen(“data”,”r”);

p2=fopen(“results”,”w”);

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

Additional modes of operation.

r + The existing file is opened to the beginning for both reading and writing w+ open for reading and writing(overwrite file).

a+ open for reading and writing(append if file exists)

Closing a file:

The input output library supports the function to close a file; it is in the following format.

Fclose (file pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.

Observe the following program.

….

FILE *p1 *p2;

p1=fopen (“Input”,”w”);

p2=fopen (“Output”,”r”);

….

… fclose(p1);

fclose(p2)

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

INPUT/OUTPUT OPERATIONS ON FILES

Following functions are used in input/output operations on files.

- getc and putc
- getw and putw
- fprintf and fscanf

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1.

putc(c,fp1);

similarly getc function is used to read a character from a file that has been open in read mode.

c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include<stdio.h>
 #include<conio.h>
void main()
{
FILE *f1;
char c;
clrscr();
printf("Data input output");
f1=fopen("Input.txt","w"); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf("\nData output\n");
f1=fopen("INPUT.txt","r"); /*Reopen the file input*/ while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
getch();
}
```

Output

Data input output LALITHA

THENMOZHI^Z

Data output LALITHA THENMOZHI

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of getw and putw are:

putw(integer,fp);

getw(fp);

```
/*Example program for using getw and putw functions*/
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *f1;
int c,i;
clrscr();
f1=fopen("Input1.txt","w");
for(i=0;i<5;i++)
{
scanf("%d",&c);
putw(c,f1);
}
fclose(f1);
f1=fopen("Input1.txt","r"); while((c=getw(f1))!=EOF) printf("%d\n",c); fclose(f1);
getch();
}
```

Output

23

23

45

56

56

23

23

45

56

56

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of theses functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f",name,age,7.5);

Here name is an array variable of type char and age is an int variable The general format of fscanf is

fscanf(fp,"controlstring",list);

This statement would cause the reading of items in the control string.

Example:

fscanf(f2,"5s%d",item,&quantity");

Like scanf, fscanf also returns the number of items that are successfully read.

/*Program to handle mixed data types*/

```
 #include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
int number,i;
char item[10];
clrscr();
fp=fopen("input2.txt","w");
fscanf(stdin,"%s%d",item,&number);
fprintf(fp,"%s%d",item,number);
fclose (fp);
fp=fopen("input2.txt","r");
 fprintf(stdout,"%s%d",item,number);
 fclose(fp);
getch();
}
```

Output

Pencil 10

Pencil 10

Random access to files:

Sometimes it is required to access only a particular part of the and not the complete file. This can be accomplished by using the following function:

fseek function:

The general format of fseek function is a s follows:

fseek(file pointer,offset, position);

This function is used to move the file position to a desired location within the file. Fileptr is a pointer to the file concerned. Offset is a number or variable of type long, and position in an integer number. Offset specifies the number of positions (bytes) to be moved from the location specified bt the position. The position can take the 3 values.

Value	Meaning
0	Beginning of the file
1	Current position
2	End of the file.

The offset may be positive or negative. Positive means move forward, negative means move backward.

Example

Statement Meaning

fseek(fp,0L,0);Go to the beginning (similar to rewind)

fseek(fp,0L,1);Stay at the current position. (Rarely used)

fseek(fp,0L,2);Go to the end of the file, past the last character of the file. fseek(fp,m,0);

Move to (m+1)th byte in the file.

fseek(fp,m,1); Go forward by m bytes.

fseek(fp,-m,1);Go backward by m bytes from the current position.

fseek(fp,-m,2);Go backward by m bytes from the end. (Position the file to the

m th character from the end).

When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and fseek returns -1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

ftell

ftell takes a file pointer and returns a number of type long, that corresponds to the

current position. This function is useful in saving the current position of a file. n=ftell(fp);

n would give the relative offset(in bytes)of the current position. This means that n bytes have already been read (or written).

Example of fseek and ftell

```
#include<stdio.h>
#include<conio.h>
#include<stdio.h>
void main()
{
FILE *fp; long n; char c; clrscr();
fp=fopen("g1.txt","w");
while((c=getchar())!=EOF)
putc(c,fp);
printf("No. of characters entered=%ld\n",ftell(fp));
fclose(fp);
fp=fopen("g1.txt","r");
n=0L;
while(feof(fp)==0)
{
fseek(fp,n,0);
printf("Position of %c is %ld\n",getc(fp),ftell(fp));
n=n+5L;
}
getch();
}
```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z

No. Of characters entered =26 Position of A is 0

Position of F is 5 Position of K is 5 Position of P is 15 Position of U is 20 Position of Z is 25 Position of is 30

Explanation

During the first reading, the file pointer crosses the end-of-file mark when the parameter n of fseek(fp,n,0) becomes 30. Therefore, after printing the content of position 30, the loop is terminated. (There is nothing in the position 30)

For reading the file from the end, we use the statement

fseek(fp,-1L,2)

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This achieved by the function.

fseek(fp,-2L,1);

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

rewind

rewind takes a file pointer and resets the position to the start of the file.

rewind(fp) n=ftell(fp);

would assign 0 to n because the file position has been set to the start of the file by rewind. Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a rewind is done implicitly.

Binary files

Binary files are very similar to arrays of structures. Binary files have two features that distinguish them from text files:

- we can instantly use any structure in the file.
- we can change the contents of a structure anywhere in the file.

After opened the binary file, we can read and write a structure or seek a specific position in the file. A file position indicator points to record0 when the file is opened.A read operation reads the structure where the file position indicator is pointing to. After reading the structure the pointer is moved to point at the next structure.A write operation will write to the currently

pointed-to structure. After the write operation the file position indicator is moved to point at the next structure.The fseek function will move the file position indicator to the record that is requested.

The file position indicator can not only point at the beginning of a structure, but can also point to any byte in the file.

The fread and fwrite function takes four parameters:

- A memory address
- Number of bytes to read per block
- Number of blocks to read
- A file variable

Example of 'write':

```
#include<stdio.h>
 #include<conio.h>
struct rec
{
int x;
};

void main()
{
int i;
FILE *fp;
struct rec my1;
clrscr();
fp=fopen("test.txt","w");
for (i=1; i <= 10;i++)
{
```

```
my1.x= i;
fwrite(&my1, sizeof(struct rec), 1,fp);
}
fclose(fp);
fp=fopen("test.txt","r");
for (i=1; i <= 10;i++)
{
fread(&my1, sizeof(struct rec), 1,fp);
printf("%d\n",my1.x);
}
```

```
}
```

Output

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

```
fclose(fp);
getch();
```

In this example we declare a structure rec with the members x,y and z of the type integer. In the main function we open (fopen) a file for writing (w). Then we check if the file is open, if

not, an error message is displayed and we exit the program. In the “for loop” we fill the structure member x with a number. Then we write the record to the file. We do this ten times, thus creating ten records. After writing the ten records, we will close the file.

Questions for Practice:

1. Add two number using pointer and Dynamic memory allocation

```
#include<stdio.h>
 #include<stdlib.h>
int main()
{
int *ptr,sum=0,i;
// Allocate memory Equivalent to 1 intExampleer ptr = (int *)malloc(sizeof(int));

for(i=0;i<2;i++)
{
printf("Enter number : ");
 scanf("%d",ptr);
sum = sum + (*ptr);
}
printf("nSum = %d",sum);
return(0);
}
```

1. Reading & Accessing array using Malloc function

```
#include<stdio.h>
 #include<stdlib.h>
#include<conio.h>
 void main()
{
clrscr();
int *ptr,*temp;
```

```
int i;
ptr = (int *)malloc(4*sizeof(int));// Allocating 8 bytes temp = ptr;  // Storing Current Pointer Value
for(i=0;i < 4;i++)
{
printf("Enter the Number %d : ",i);
scanf("%d",ptr);
ptr++;// New Location i.e increment Pointer
}
ptr = temp;
for(i=0;i < 4;i++)
{
printf("\nNumber(%d) : %d",i,*ptr);
ptr++;
}
getch();
}
```

Output:

```
Enter the Number 0 : 45
Enter the Number 1 : 35
Enter the Number 2 : 25
Enter the Number 3 : 15
 Number(0) : 45
Number(1) : 35
Number(2) : 25
Number(3) : 15
```

Text / Reference Books:

1. Byron S Gottfried, “Programming with C”, Schaum's Outlines, 2 nd Edition, Tata McGrawHill, 2006.

2. Dromey R.G., “How to Solve it by Computer”, Pearson Education, 4 th Reprint, 2007.

3. Kernighan, B.W. and Ritchie, D.M., “The C Programming language”, 2 nd Edition, Pearson Education, 2006.

4. Balaguruswami. E., "Programming in C", TMH Publications, 2003.

5. Yashavant P. Kanetkar, ‘LET US C’, 5 th Edition.2005.

6. Stevens, ‘Graphics programming in C’, BPB Publication, 2006.

7. Subburaj. R , ‘Programming in C’, Vikas Publishing, 1 st Edition, 2000.