ALAGAPPA UNIVERSITY

(Accredited with 'A+' Grade by NAAC (with CGPA: 3.64) in the Third Cycle and Graded as category - I University by MHRD-UGC)
(A State University Established by the Government of Tamilnadu)

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

B.COM. (COMPUTER APPLICATIONS)

123 32

SECOND YEAR – THIRD SEMESTER

PRINCIPLES OF C PROGRAMMING

Author :
Dr. K. Sivakumar,
Associate Professor & Head,
Department of Computer Applications,
Park's College (Autonomous),
Tirupur- 641 605.

Reviewer:
Dr. P. Prabhu,
Assistant Professor in Information Technology,
Directorate of Distance Education
Alagappa University,
Karaikudi.

Syllabi **PAGE NO**

Syllabi **PAGE NO**

CONTENT

PAGE NO

BLOCK I: PROGRAMMING CONCEPTS & C LANGUAGE

UNIT I - PRINCIPLES OF PROGRAMMING

Structure

1.1. INTRODUCTION

As an end in itself, understanding programming language concepts and terms is important to enable you learn more about programming and programming languages after the course is over. Without understanding these concepts and terms, you will have difficulty discussing programming language ideas with others, and will have difficulty in reading the technical literature. Since computer science is rapidly evolving new programming languages and since language issues are important in many areas of computer science the ability to learn more quickly is important to maintaining your technical edge.

1.2. OBJECTIVES

After going through this lesson you would be in a positions to

- Use ideas from the various paradigms when programming in a language that is not explicitly suited to that paradigm.
- Implement important run-time data structures and algorithms, and state a first-order approximation to the time and space costs incurred by programs that use such implementations.
- Evaluate design alternatives for language features by applying general principles of language design or by analogy to historically important languages.
- Design features of programming languages, and justify your design decisions.

1.3. PROGRAMMING

Programming is the process of writing a sequence of instructions to be executed by a computer to solve a problem. It is also considered as the act of writing computer programs. Computer programs are set of instructions that tell a computer to perform certain operations. The instructions in programs are logically sequenced and assembled through the act of programming. Computer programming has many facets: It is like engineering because computer programs must be carefully designed to be reliable and inexpensive to maintain. It is an art because good programs require that the programmer use intuition and a personal sense of style. It is a literary effort because programs must be understood by computers, and this requires mastery of a programming language.

Reasons for Studying Concepts of Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing.

1.4. PROGRAMMING DOMAIN

Programming domain defines the ability of using a specific language for a specific usage. There are many programming domains, but let's take the common domains.

1.4.1. Scientific applications

Scientific Applications Typically, scientific applications have simple data structures but require large numbers of floating-point arithmetic computations. For some scientific applications where efficiency is the primary concern, like those that were common in the 1950's and 1960's, no subsequent language is significantly better than FORTRAN.FORTRAN.

1.4.2. Business applications

The use of computers for business applications began in the 1950's. The first successful high-level language for business was COBOL which appeared in 1960.COBOL Business languages are characterized, according to the needs of the application, by elaborate input and output facilities and

decimal data types. With the advent of microcomputers came new ways of businesses, especially small businesses, to use computers. Two specific tools, spreadsheet systems and database systems, were developed for business and now are widely used.

1.4.3. Artificial intelligence

AI is a broad area of computer applications characterized by the absence of exact algorithms and the use of symbolic computations rather than numeric computation. Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. The first widely used programming language developed for AI applications was the functional language LISP (Scheme) which appeared in 1959.LISP. An alternative approach to these applications appeared in the early 1970's: logic programming using Prolog language Prolog.

1.4.4. Systems programming

The operating system and all of the programming support tools of a computer system are collectively known as its systems software. Systems software is used almost continuously and therefore must have execution efficiency. A language for this domain must have low-level features that allow the software to external devices to be written. In the 1960's and 1970's, some computer manufacturers, such as IBM, Digital, and Burroughs (now UNISYS) developed special machine-oriented high level languages for systems software on their machines. For IBM mainframe computers, the language was PL/S, a dialect of PL/I; for Digital, it Is BLISS, a language at a level just above assembly language; for Burroughs, it was Extended Algol.PL/S BLISSAlgol The UNIX operating system is written almost entirely in C, which was made it relatively easy to port, or move, to different machines.

1.4.5. Web Software

For web software we can write applications in any whatever language platform is, we can build web software using C/C++. It is more complex for making web software, but need another programming environment such as, mark-up languages, pre-processor hypertext and .net framework to be executed properly. Java is actually the most commonly used programming language for building and running web applications. It has powerful feature than the other programming languages which is applets that provide creating a web based application.

1.5. CATEGORIES OF PROGRAMMING LANGUAGES

Computer language or programming language is a coded syntax used by computer programmers to communicate with a computer. It is the only language that computers, software programs and computer hardware can understand. Computer language establishes a flow of communication

between software programs. The language enables a computer user to dictate what commands the computer must perform to process data. Computer language comes in various types that employ different sets of syntax.

There are three types of programming languages, which can be categorized into the following way,

1. Low-Level Language (Or) Machine Level Language.
2. Assembly Level Language.
3. High Level Language.

These languages are not mutually exclusive, and some languages can belong to multiple categories. The terms low-level and high-level are also open to interpretation, and some languages that were once considered high-level are now considered low-level as languages have continued to develop.

1.5.1. Low-Level Language (Or) Machine Level Language

Machine language or machine code is the native language directly understood by the computer's central processing unit or CPU. This type of computer language is not easy to understand, as it only uses a binary system, an element of notations containing only a series of numbers consisting of one and zero, to produce commands. The computer's processor needs to convert high-level languages into this language before it can run a program or do a user-defined command. To convert a certain language into machine code, the computer processor needs a compiler, a program that converts a source code written in one language into different language syntax. The compiler generates a binary file, or executable file, that the CPU will execute. Every computer processor has its own set of machine code. The machine code will determine what the computer processor should do, and how it should do it.

1.5.2. Assembly Level Language

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

1.5.3. High-Level Languages

High level computer languages give formats close to English language and the purpose of developing high level languages is to enable people to write programs easily and in their own native language environment (English). High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. Each instruction in the high level language is translated into many machine language instructions thus showing one-to-many translation.

- **Problem-Oriented Language:** These are languages used for handling specialized types of data processing problems where programmer only specifies the input/output requirements and other relative information of the problem, that are to be solved. The programmer does not have to specify the procedure to be followed in solving that particular problem.
- **Procedural Language:** These are general purpose languages that are designed to express the logic of a data processing problem.
- **Non-procedural Language:** Computer Programming Languages that allow users and professional programmers to specify the results they want without specifying how to solve the problem.

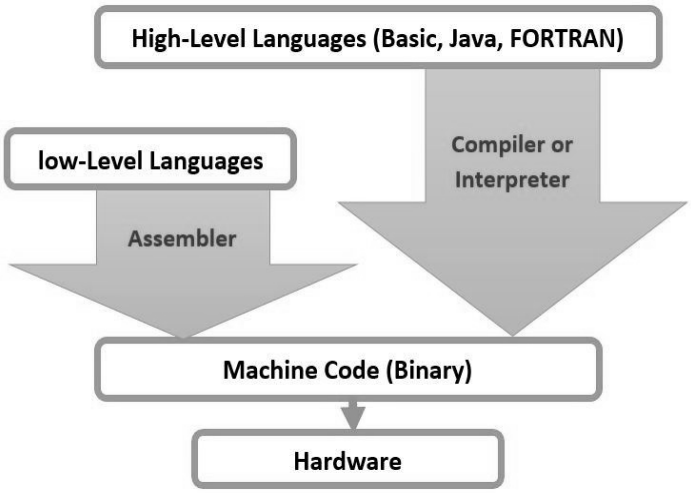

Figure 1: - High-Level Language

1.6. PROGRAMMING DESIGN METHODOLOGIES

When programs are developed to solve real-life problems like inventory management, payroll processing, student admissions, examination result processing, etc. they tend to be huge and complex. The approach to analyzing such complex problems, planning for software development and controlling the development process is called programming methodology. Programming Methodology is the approach to analyzing such complex problems by planning the software development and controlling the development process.

Here the problem is broken down into logical units rather than functional units. Software developers may choose one or a combination of more than one of these methodologies to develop software. Note that in each of the methodologies discussed, problem has to be broken down into smaller units. To do this, developers use any of the following two approaches −

- Top-down approach
- Bottom-up approach

1.6.1. Top-Down and Bottom-Up

Top-Down Approach

The basic idea in top-down approach is to break a complex algorithm or a problem into smaller segments called modules, this process is also called as *modularization*. The modules are further decomposed until there is no space left for breaking the modules without hampering the originality. The

uniqueness of the problem must be retained and preserved. The decomposition of the modules is restricted after achieving a certain level of modularity. The top-down way of solving a program is step-by-step process of breaking down the problem into chunks for organizing and solving the sole problem. Advantages of top-down approach is as follows,

1. In this approach, first, we develop and test most important module.
2. This approach is easy to see the progress of the project by developer or customer.
3. Using this approach, we can utilize computer resources in a proper manner according to the project.
4. Testing and debugging is easier and efficient.
5. In this approach, project implementation is smoother and shorter.
6. This approach is good for detecting and correcting time delays.

Bottom-Up Approach

As the name suggests, this method of solving a problem works exactly opposite of how the top-down approach works. In this approach we start working from the most basic level of problem solving and moving up in conjugation of several parts of the solution to achieve required results. The most fundamental units, modules and sub-modules are designed and solved individually, these units are then integrated together to get a more concrete base to problem solving.

This bottom-up approach works in different phases or layers. Each module designed is tested at fundamental level that means unit testing is done before the integration of the individual modules to get solution. Advantages of bottom-up approach is as follows,

1. Solves the fundamental low-level problem and integrates them into a larger one.
2. Examine what data is to be encapsulated, and implies the concept of information hiding.
3. Needs a specific amount of communication.
4. Redundancy can be eliminated.
5. Object-oriented programming languages follow the bottom-up approach.

1.7. PROGRAM DEVELOPMENT CYCLE

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

- Problem Definition
- Problem Analysis
- Algorithm Development
- Coding & Documentation
- Testing & Debugging
- Maintenance

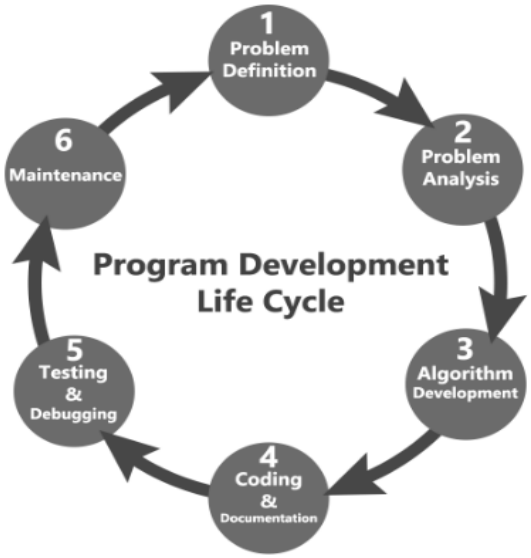

Figure 2: - Program Development Life Cycle

- **Problem Definition**

  In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirment, what should be the output of the problem solution? These are defined in this first phase of the program development life cycle.

- **Problem Analysis**

  In second phase, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

- **Algorithm Development**

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

- **Coding & Documentation**

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

- **Testing & Debugging**

  During this phase, we check whether the code written in previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test that whether it is providing the desired output or not.

- **Maintenance**

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

enhancement, then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

1.7. PROGRAM EXECUTION AND TRANSLATION PROCESS

Assembly language is machine dependent yet mnemonics that are being used to represent instructions in it are not directly understandable by machine and high Level language is machine independent. A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code. The programs are written mostly in high level languages like Java, C++, and Python etc. and are called source code. This source code cannot be executed directly by the computer and must be converted into machine language to be executed. Hence, a special translator system software is used to translate the program written in high-level language into machine code is called Language Processor and the program after translated into machine code.

The language processors can be any of the following three types:

Compiler: -

The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.

Assembler: -

The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.

Interpreter: -

The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

Execution of a C program involves four stages using different compiling/execution tool, these tools are set of programs which help to complete the C program's execution process.

1. Preprocessor
2. Compiler
3. Linker
4. Loader

These tools make the program running.

Preprocessor: -

This is the first stage of any C program execution process; in this stage Preprocessor processes the program before compilation. Preprocessor include header files, expand the Macros.

Complier: -

This is the second stage of any C program execution process, in this stage generated output file after preprocessing (with source code) will be passed to the compiler for compilation. Complier will compile the program, checks the errors and generates the object file (this object file contains assembly code).

Linker: -

This is the third stage of any C program execution process, in this stage Linker links the more than one object files or libraries and generates the executable file.

Loader: -

This is the fourth or final stage of any C program execution process, in this stage Loader loads the executable file into the main/primary memory. And program run.

1.9. PROBLEM SOLVING USING ALGORITHMS AND FLOWCHARTS

During the process of solving any problem, one tries to find the necessary steps to be taken in a sequence. A typical programming task can be divided into two phases:

1. Problem solving phase.
2. Implementation phase.

It produces an ordered sequence of steps that describe solution of problem. This sequence of steps is called an algorithm. Implement the program in some programming language

Algorithm

The word "algorithm" relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique. Software Engineer commonly uses an algorithm for planning and solving the problems. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time.

Algorithm has the following characteristics,

- **Input:** An algorithm may or may not require input
- **Output:** Each algorithm is expected to produce at least one result
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps

The algorithm and flowchart include following three types of control structures.

1. **Sequence:** In the sequence structure, statements are placed one after the other and the execution takes place starting from up to down.
2. **Branching (Selection):** In branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches is explored; but in the case of FALSE condition, the other alternative is taken. Generally, the 'IF-THEN' is used to represent branch control.

3. **Loop (Repetition):** The Loop or Repetition allows a statement(s) to be executed repeatedly based on certain loop condition e.g. WHILE, FOR loops.

Advantages of algorithm

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.

Flowchart

Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flowchart one can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

Advantages of flowchart:

- Flowchart is an excellent way of communicating the logic of a program.
- Easy and efficient to analyze problem using flowchart.
- During program development cycle, the flowchart plays the role of a blueprint, which makes program development process easier.
- After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
- It is easy to convert the flowchart into any programming language code. Flowchart is diagrammatic /Graphical representation of sequence of steps to solve a problem.

1.10. PERFORMANCE ANALYSIS AND MEASUREMENTS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- A Priori Analysis − this is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- A Posterior Analysis − this is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

1.10.1. Time and Space complexity

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of:

- **Space Complexity.**
- **Time Complexity.**

Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components:

- **Fixed Component**: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.
- **Variable Component**: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'.

Time Complexity of an algorithm (basically when converted to program) is the amount of computer time it needs to run to completion. The time taken by a program is the sum of the compile time and the run/execution time. The compile time is independent of the instance (problem specific) characteristics. Following factors affect the time complexity:

- Characteristics of compiler used to compile the program.
- Computer Machine on which the program is executed and physically clocked.
- Multiuser execution system.
- Number of program steps.

Here the number of steps is the most prominent instance characteristics and the number of steps any program statement is assigned depends on the kind of statement like

- Comments count as zero steps,
- An assignment statement which does not involve any calls to other algorithm is counted as one step,
- For iterative statements, we consider the steps count only for the control part of the statement etc.

Therefore to calculate total number program of program steps we use following procedure. For this we build a table in which we list the total number of steps contributed by each statement.

1.11. LET US SUM UP

In this unit, you have learnt about the basics of programming, categories of programming languages and problem solving strategies. This knowledge would make you understand the various programming domain and types of languages used to develop the programs for real time problems.

Thus, the principles of programming unit would have brought you to closer to know the concept of basics of programming.

1.12. UNIT – END QUESTIONS

1. List out the various programming domains used for real time problems.
2. Explain about the categories of programming languages.
3. Define algorithm?

1.13. ANSWER TO CHECK YOUR PROGRESS

1. Scientific Applications Typically, scientific applications have simple data structures but require large numbers of floating-point arithmetic computations. Business languages are characterized, according to the needs of the application, by elaborate input and output facilities and decimal data types. AI is a broad area of computer applications characterized by the absence of exact algorithms and the use of symbolic computations rather than numeric computation. The operating system and all of the programming support tools of a computer system are collectively known as its systems software. Systems software is used almost continuously and therefore must have execution efficiency.
2. Machine language or machine code is the native language directly understood by the computer's central processing unit or CPU. This type of computer language is not easy to understand, as it only uses a binary system, an element of notations containing only a series of numbers consisting of one and zero, to produce commands. Assembly language is very difficult and needs a lot of practice to master it because very small English support is given. This symbolic language helps in compiler orientations. High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes.
3. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time.

1.14. SUGGESTED READNGS

1. “The C Programming Language”, Brain W. Kernighan / Dennis Ritchie, Pearson Publications, 2015.
2. “C: The Complete Reference”, Herbert Schildt, McGraw Hill Publications, Fourth Edition, 2017.

UNIT II – C PROGRAMMING

Structure

2.1. INTRODUCTION

In this lesson you will be aware with the basic elements used to construct simple C statements. These elements include the C character set, keywords and identifiers, constants, data types, variables, arrays, declarations, expressions and statements. These basic elements are used to construct more comprehensive program components. Some of the basic elements needs very detailed information, however, the purpose of this type of basic elements is to introduce certain basic concepts and to provide some necessary definitions for the topics that follow in next few lessons.

2.2. OBJECTIVES

After going through this lesson you would be in a positions to

- Recognize ‘C’ character set.
- Recognize keywords and identifiers.
- Define constants, data types, variables and arrays.
- Explain the concept of declaration.

2.3. FEATURES OF C AND BASIC STRUCTURE

The C Language is developed for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc. C

programming is considered as the base for other programming languages, that is why it is known as mother language.

C Language is an amazing language when it comes to simplicity of syntax with decent functionality. It is a robust language with a rich set of built-in functions and operators that can be used to write any complex program. The C compiler combines the capabilities of an assembly language with features of a high-level language. Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators. C is highly portable this means that programs once were written can be run on another machine with little or no modification.

Features of C language

- It is a robust language with rich set of built-in functions and operators that can be used to write any complex program.
- The C compiler combines the capabilities of an assembly language with features of a high-level language.
- Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators.
- It is many time faster than BASIC.
- C is highly portable this means that programs once written can be run on another machines with little or no modification.
- Another important feature of C program is its ability to extend itself.
- A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.
- C language is the most widely used language in operating systems and embedded system development today.

2.3.1. Basic Structure of a C Program

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 you 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.

Documentation section
Link section
Definition section
Global declaration section

main( ) function section
{
declaration part
executable part
}

Subprogram section function 1 function 2 function 3 } User Defined functions . .

function n

Figure 3: - Structure of a C Program

Documentation Section

This section consists of comment lines which include the name of programmer, the author and other details like time and date of writing the program. Documentation section helps anyone to get an overview of the program.

Link Section

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library.

Definition Section

All the symbolic constants are written in definition section. Macros are known as symbolic constants.

Global Declaration Section

The global variables that can be used anywhere in the program are declared in global declaration section. This section also declares the user defined functions.

main() Function Section

It is necessary have one main() function section in every C program. This section contains two parts, declaration and executable part. The declaration part declares all the variables that are used in executable part. These two parts must be written in between the opening and closing braces. Each statement in the declaration and executable part must end with a semicolon (;). The execution of program starts at opening braces and ends at closing braces.

Subprogram Section

The subprogram section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.

2.3.2. Simple C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

```
#include <stdio.h>
int main()
{
printf("Hello C Language");
return 0;
}
```

- **#include <stdio.h>** includes the **standard input output** library functions. The printf() function is defined in stdio.h .
- **int main()** The **main() function is the entry point of every program** in c language.
- **printf()** The printf() function is **used to print data** on the console.

- **return 0** The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

By menu

- Now **click on the compile menu then compile sub menu** to compile the c program.
- Then **click on the run menu then run sub menu** to run the c program.

By shortcut

- **Or, press ctrl+f9** keys compile and run the program directly.
- You can view the user screen any time by pressing the **alt+f5** keys.
- Now **press Esc** to return to the turbo c console.

2.4. CONSTANTS

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called constants. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Constants

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order. Here are some examples of integer constants −

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Floating-point or Real Constants

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point constants either in decimal form or exponential form. While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E. Here are some examples of floating-point constants −

```
3.14159             /* Legal */
314159E-5L          /* Legal */
510E                /* Illegal: incomplete exponent */
210f                /* Illegal: no decimal or exponent */
```

Single Character Constants

It simply contains a single character enclosed within ' and ' (a pair of single quote). It is to be noted that the character '8' is not the same as 8. Character constants have a specific set of integer values known as ASCII values (American Standard Code for Information Interchange).
Example: - 'X', '5', ';'

String Constants

These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blank spaces. It is again to be noted that "G" and 'G' are different - because "G" represents a string as it is enclosed within a pair of double quotes whereas 'G' represents a single character.
Example: - "Hello!", "2015", "2+1"

Backslash Character constants

C supports some character constants having a backslash in front of it. The lists of backslash characters have a specific meaning which is known to the compiler. They are also termed as "Escape Sequence".
For Example:
\t is used to give a tab.
\n is used to give a new line.

Table 1: - Backslash Character Constants

\a	beep sound
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\'	single quote
\"	double quote
\\	backslash
\0	null
\f	form feed

2.5. CONCEPT OF AN INTEGER AND VARIABLE

Integers are whole numbers, having no fractional component, in contrast to real numbers (floating point in C). In the C programming language, character data is considered an integer data type. Some properties of note in the C language are that integer types have various sizes, with the implication that the value of the number that can be stored / represented in any particular integer variable has a maximum. The number of bytes in an integer word defines this maximum. In C, integers can be defined as either signed, or unsigned. Naturally, unsigned integer data types cannot store negative values.

Integer data can be used as indices to arrays. Integer data can always be used to test for equality (as opposed to floating point variables, where testing for equality is only guaranteed to work when one value is zero). The language defines the value zero to be a word which has all bits equal to zero. Non-zero integers may be stored in any notation, although two's complement notation is used virtually universally. It is meaningful to perform Boolean arithmetic (left and right shifts, ands, or, xors, complements, etc) on integer data (again, in contrast to floating point). The C language defines various notations for expressing integer values in source code (decimal, hex, octal, binary, character, etc), whereas it almost never makes sense to use any of these to create floating point constants.

2.5.1. Rules for naming variables and assigning values to variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

***Rules* for forming a variable:**

1. The starting character should be a letter. The first character may be followed by a sequence of letters or digits.
2. The maximum number of characters in a variable may be 8 characters. The number of characters differs from compiler to compiler.
3. Upper and lower case are significant. Example: TOTAL is not same as total or Total.
4. The variable should not be a keyword.
5. White space is not allowed.
6. Special characters except _(underscore) symbol are not allowed.

Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. Variables are used to store information to be referenced and manipulated in a computer program. Value of the variable can change, depending on conditions or on information passed to the program.

Variable Definition and Initialization in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows –

type variable_list;

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

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

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initialize consists of an equal sign followed by a constant expression as follows −

type variable_name = value;

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initialize: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

2.6. LET US SUM UP

In this unit, you have learnt about the structure of C programming, constants and integer and values. This knowledge would make you understand the section wise structure of C programming language and types of constants used to develop the C programs for real time problems. Thus, the C programming unit would have brought you to closer to know the concept of programming structure and aspects.

2.7. UNIT – END QUESTIONS

1. Discuss about the structure of C Programming.
2. Explain about the constants and its types.
3. Define variables?

2.8. ANSWER TO CHECK YOUR PROGRESS

1. Documentation Section consists of comment lines which include the name of programmer, the author and other details like time and date of writing the program. Link Section consists of the header files of the functions that are used in the program. Definition Section defines all the symbolic constants are written in definition section. Macros are known as symbolic constants. The global variables that can be used anywhere in the program are declared in global declaration section. This section also declares the user defined functions. main () Function Section defines necessary have one main() function section in every C program. This section contains two parts, declaration and executable part. Subprogram Section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.
2. Integer Constants is an integer literal can be a decimal, octal, or hexadecimal constant. Floating-point or Real Constants are literal has an integer part, a decimal point, a fractional part, and an exponent part. Single Character Constants is simply contains a single character enclosed within ' and ' (a pair of single quote). String Constants are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and

blank spaces. Backslash Character constants support some character constants having a backslash in front of it.

3. A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

2.9. SUGGESTED READINGS

1. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
2. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.
3. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.

UNIT III – OPERATORS AND EXPRESSIONS

Structure

3.1. INTRODUCTION

Operators form expressions by joining individual constants, variables, array elements as discussed in previous lesson. C includes a large number of operators which fall into different categories. In this lesson we will see how arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operators are used to form expressions. The data items on which operators act upon are called operands. Some operators require two operands while others require only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variable as operand.

3.2. OBJECTIVES

After going through this lesson you will be able to

- l recognize arithmetic operators
- l explain unary operators
- l define relational, logical, assignment & conditional operators
- l explain library functions

3.3. OPERATORS

An Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

Operators are the foundation of any programming language. Thus the functionality of C language is incomplete without the use of operators. Operators allow us to perform different kinds of operations on operands. In C, operators in can be categorized in following categories:

- Arithmetic operators.
- Relational Operators.
- Logical Operators.
- Increment and Decrement Operators.
- Conditional Operators.
- Bitwise Operators.

C supports many operators to perform various kinds of operations. With C operators, you can do arithmetic operations, comparing data, modifying variables, combining relationship logically, etc.

C operators operate on one or more operands to produce a value.

- The operators that take one operand are called *unary* operators.
- The operators that require two operands are called *binary* operators.

3.3.1. Arithmetic Operator

C supports almost common arithmetic operators such as +,-,*, / and modulus operator %. The modulus operator (%) returns the remainder of integer division calculation. Note that the modulus operator cannot be applied to a *double* or *float*.

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Table 2: - Arithmetic Operator

Operands can be integer quantities, floating-point quantities or characters. The modulus operator requires that both operands be integers & the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero, though the operands need not be integers. Division of one integer quantity by another is referred to as integer division. With this division the decimal portion of the quotient will be dropped. If division operation is carried out with two floating- point

numbers, or with one floating point number. & one integer, the result will be a floating-point quotient.

The example program for arithmetic operator is as follows,

```
#include<stdio.h>
void main( )
{
int m1,m2,m3,m4,m5,tot,avg;
clrscr();
printf("Enter the five subject marks:");
scanf("%d%d%d%d%d",&m1,&m2,&m3,&m4,&m5);
tot=m1+m2+m3+m4+m5;
avg=tot/5;
printf("\n the total for five subject mark is=%d",tot);
printf("\n the average for five subject mark is=%d",avg);
getch();
}
```

3.3.2. Relational Operator

Relational operators are symbols that are used to test the relationship between two variables, or between a variable and a constant. The test for equality is made by means of two adjacent equal signs with no space separating them. 'C' has six relational operators as follows:

Operator	Meaning of Operator
==	Equal to
>	Greater than
<	Less than
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Table 3: - Relational Operator

These operators all fall within the same precedence group, which is lower than the unary and arithmetic operators. The associativity of these operators is left-to-right. The equality operators ==and != fall into a separate precedence group, beneath the relational operators. Their associativity is also from left-to-right. These relational operator are used to form logical expression representing condition thet are either true or false. The resulting expression will be of type integer, since true is represented by the integer value and false is represented by the value0.

The example program for relational operator is as follows,

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

```
{
int a,b;
clrscr();
printf("Enter the a and b value");
scanf("%d%d", &a, &b);
if(a>b)
{
printf("a is greater than b");
}
Else
{
printf("b is greater than a");
}
getch();
}
```

3.3.3. Logical Operator

C logical operators to connect expressions and/or variables to form compound conditions. The C logical expression returns an integer (int). The result has value 1 if the expression is evaluated to true otherwise it returns 0. C uses the following symbols for the Boolean operations AND, OR, and NOT.

C has the following three logical operators.

- && (logical AND)
- || (logical OR)
- ! (logical NOT)

These operators are referred to as logical and, logical or, respectively. The result of a logical and operation will be true only if both operands are true, whereas the result of a logical or operation will be true if either operand is true or if both operands are true. The logical operators act upon operands that are themselves logical expressions.

Each of the logical operators falls into its own precedence group. Logical and has a higher precedence than logical or. Both precedence groups are lower than the group containing the equality operators. The associativity is left to right. 'C' also includes the unary operator !, that negates the value of a logical expression. This is known as logical negation or logical NOT operator. The associativity of negation operator is right to left.

The following program illustrate the operating procedure of logical operator.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
clrscr();
printf("Enter the a, b and c value");
scanf("%d%d%d",&a,&b,&c);
if ((a>b) && (a>c))
{
```

```
printf(“ a is greater than b & c”);
}
else if (b>c)
{
printf(“b is greater than c”);
}
else
{
Printf("c is greater than b”);
}
getch();
}
```

3.3.4. The Conditional Operator

The conditional operators in C language are known by two more names

- Ternary Operator
- ? : Operator

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is:

expression 1 ? expression 2: expression 3

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

The following example illustrate how to use the conditional operator.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
clrscr();
printf(“Enter the a & b value”);
scanf(“%d%d”, &a,&b);
c=(a>b)?a:b;
printf(“The greater value of a & b is = %d”,c);
getch();
}
```

3.3.5. Library Functions

Library functions carry out various commonly used operations or calculations. Some functions return a data item to their access point, others indicate whether a condition is true or false by returning 1 or 0

respectively, still others carry out specific operations on data items but do not return anything. Features which tend to be computer dependent are generally written as library functions.

Functionally similar library functions are usually grouped together as object programs in separate library files. These library files are supplied as a part of each C compiler. A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The arguments can be constants, variable names or more complex expressions. The parentheses must be present, even if there are no arguments, A function that returns a data item can appear anywhere within an expression in place of a constant or an identifier. A function that carries out operations on data items but does not return anything can be accessed simply by writing the function name, since this type of function reference constitutes an expression statement. In order to use a library function it may be necessary to include certain specific information within the main portion of the program. This information is generally stored in special files supplied with the compiler. Thus, the required information can be obtained simply by accessing these special files. This is accomplished with the preprocessor statement.

Header file	Description
stdio.h	This is standard input/output header file - I/O functions are declared
conio.h	This is console input/output header file
string.h	All string related functions are defined in this header file
stdlib.h	This header file contains general functions used in C programs
math.h	All maths related functions are defined in this header file
time.h	This header file contains time and clock related functions
ctype.h	All character handling functions are defined in this header file
stdarg.h	Variable argument functions are declared in this header file
setjmp.h	This file contains all jump functions
locale.h	This file contains locale functions
errno.h	Error handling functions are given in this file

Table 4: - Library Functions

3.3.6. Bitwise Operator

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

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

Table 5: - Bitwise Operator

- Binary AND Operator copies a bit to the result if it exists in both operands.
- Binary OR Operator copies a bit if it exists in either operand.
- Binary XOR Operator copies the bit if it is set in one operand but not both.
- Binary One's Complement Operator is unary and has the effect of 'flipping' bits.
- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

3.3.7. Increment and Decrement Operator

C has two very useful operators that are not generally found in other languages. These are,

- ++ - The increment Operator:
- - - - The decrement Operator.

The increment operator causes its operand to increase by one, whereas the decrement operator causes its operand to be decreased by one. The operand used with each of these operators must be a single variable. For example, x is an integer variable that has been assigned a value of 10. The expression ++ x, which is equivalent to writing x= x+1, causes the value of x to be creased to 11. Similarly the expression --x, which is equivalent to x=x-1, causes the original value of x to be decreased to 9. The increment and decrement operators can each be utilized in two different ways, depending on whether the operator is written before or after the

operand. If the operator precedes the operand, then the value of operand will be altered before it is used for its intended purpose within the program. If, however the operator follows the operand then the value of the operand will be changed after it is used. Similarly for decrement operator the current value of x will be 9 if we say -- x.

The following example illustrate the increment and decrement operator.

```
#include<stdio.h>
#include<conio.h>
void main()     {
int n,i;
clrscr();
printf("Enter the value of n");
scanf("%d", &n);
for(i=0;i<=n;i++)
{
printf("\n the value of i is = %d",i);
}
getch();          }
```

3.3.8. The Sizeof() Operator

The size of is a compiler time operator and, when used with an operand, it returns the number of bytes the operand occupies. Sizeof() operator is used to return the size of a variable. Suppose we have an integer variable 'i', so the value of sizeof(i) will be 4 because on declaring the variable 'i' as of type integer, the size of the variable becomes 4 bytes.

Example:

1) m = sizeof(sum);
2) n = sizeof(long int)
3) k = sizeof(235L)

3.3.9. Precedence of Operators

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

In general, the precedence of the operators in an expression determines whether it is necessary for you to put parentheses in to get the result you want, but if you are unsure of the precedence of the operators you are using, it does no harm to put the parentheses in. Below table shows the order of precedence for all the operators in C, from highest at the top to lowest at the bottom.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Table 6: - Operator Precedence

3.4. LET US SUM UP

In this unit, you have learnt about the various types of operators and precedence of operators for evaluating the expression. This knowledge would make you understand the available operators that C language supports and its syntax and priority of operators while executing the expressions. Thus, the operators and expressions unit would have brought you to closer to know the concept of operator usage and evaluation of expressions also.

3.5. UNIT – END QUESTIONS

1. List out various operators supported by C language.
2. Describe about the precedence of operators.

3.6. ANSWER TO CHECK YOUR PROGRESS

1. An Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. In C, operators in can be categorized in following categories:
   Arithmetic operators, Relational Operators, Logical Operators, Increment and Decrement Operators, Conditional Operators, and Bitwise Operators.
   C supports many operators to perform various kinds of operations. With C operators, you can do arithmetic operations, comparing data, modifying variables, combining relationship logically, etc.

2. Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.
   For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.
   Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.
   In general, the precedence of the operators in an expression determines whether it is necessary for you to put parentheses in to get the result you want, but if you are unsure of the precedence of the operators you are using, it does no harm to put the parentheses in.

3.7. SUGGESTED READINGS

1. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.
2. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.
3. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.

UNIT IV – DATA TYPES AND INPUT/OUTPUT OPERATORS

Structure

4.1. INTRODUCTION

In 'C' language input and output of data is done by a collection of library functions like getchar, putchar, scanf, printf, gets and puts. These functions permit the transfer of information between the computer and the standard input/output devices. An input/ output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of parameters enclosed in parentheses. Some input/output functions do not require parameters, but the empty parentheses must appear. In this lesson we will discuss some input/output functions in detail.

4.2. OBJECTIVES

After going through this lesson you would be able to

- Explain getchar function
- Define putchar function
- Describe gets & puts function
- Use interactive programming
-

4.3. DATA TYPES

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data

types to handle various kinds of data that we can use in our program. These data types have different storage capacities. C language supports the following different types of data types:

1. **Primary data types**:
   These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.
2. **Derived data types**:
   Derived data types are nothing but primary data types but a little twisted or grouped together like **array**, **structure**, **union** and **pointer**.
3. **Enumerated types**
   They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
4. **The type void**
   The type specifier *void* indicates that no value is available.

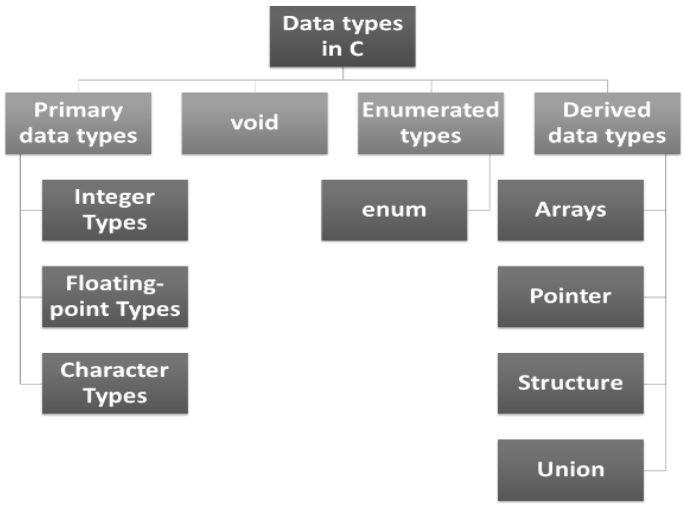

Figure 4: - C Data Types

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is. The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. Each data type requires different memory requirements which may vary from one C compiler to another.

Integer Types

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 25, 52.

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d
char	1	%c
float	4	%f

Type	Size (bytes)	Format Specifier
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%li
long long int	at least 8	%lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu

Table 7: - Integer Data types

If short int and int both have the same memory requirements (e.g, 2 bytes), then long int will generally have double the requirements. (e.g., 4 bytes) Similarly if int & long int both have the same memory requirements (e.g., 4 bytes) then short int will have half the memory requirements (e.g. 2 bytes). An unsigned int means all the bits are used to represent the numerical value unlike in the case of ordinary int in which the leftmost bit is reserved for the sign. Thus the size of an unsigned int can be approximately twice as large as an ordinary int. For example if an ordinary int can vary from -32,768 to +32,767 then an unsigned int can vary from 0 to 65,535.

4.3.1. Floating Point Numbers

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range
float	4 byte	1.2E-38 to 3.4E+38
double	8 byte	2.3E-308 to 1.7E+308
long double	10 byte	3.4E-4932 to 1.1E+4932

Table 8: - Floating Point Numbers

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs.

As discussed above that floating point numbers have a decimal point. The C compiler differentiates between floating point numbers & integers because they are stored differently in the computer. Floating point numbers sometimes are referred to as real numbers. They include

all the numbers between the integers. Some of the differences are listed below between floating point numbers and integer.

1. Integer includes only whole numbers, but floating point numbers can be either whole or fractional.
2. Integers are always exact, whereas floating point numbers sometimes can lead to loss of mathematical precision.
3. Floating point operations are slower in execution and often occupy more memory than integer operations.

Floating point numbers may also be expressed in scientific notation. For example, the expression 2345.34e6 represents a floating point number in scientific notation. The letter e stands for the word exponent. The exponent is the whole number following the letter e; the part of the number before the letter e is called the mantissa. The number 2345.34e6 should be interpreted as: 2345.34 times 10 to the 6^{th} power.

4.3.3. Mixed-Mode Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable.

Rules for Evaluation Mixed Mode Arithmetic Expression

Rule 1

Evaluate Expressions always from Left to Right.

For Example: 3 + 5 - 4 = 4.

Rule

Priority of an operator is also considered while calculating an expression.

A mixed mode expression is an expression in which the two operands are not of the same time. For example one might be an int and another float.

Examples: 1.5*3, 5/3.0, 4+1.1, -3-3.0

In general the integral value will be promoted to a real value and the result will be a real. In general try to avoid mixing types when doing math unless the answer is really obvious.

For example take:

6.0 * (1/2)

The answer to that is 0, and NOT 3. The division in the brackets happens first. So you get integer division 1/2 = 0. Then the 0 gets promoted to 0.0 and the final answer is 6.0 * 0.0 = 0.0

4.3.4. The type cast Operator

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can typecast long to int. You can convert values from one type to another explicitly using the cast operator. There are two types of type casting in c languages that are Implicit conversions and Explicit Conversions.

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the

values from one type to another explicitly using the cast operators follows −

(type_name) expression

Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as implicit type conversion or type promotion.

Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion. The explicit type conversion is also known as type casting.

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.

4.3.5. The type char

The char type is used to represent individual characters. Hence, the char type will generally require only1 byte of memory. With most compilers, a char data type will permit a range of values extending from 0 to 255.

A single character can be defined as a character type data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned chars have values between 0 and 255, signed chars have values from -128 to 127.

Every identifier that represents a number or a character within a C program must be associated with one of the basic data types before the identifier appears in an executable statement.

4.4. KEYWORDS

C programs are constructed from a set of reserved words which provide control and from libraries which perform special functions. The basic instructions are built up using a reserved set of words, such as main, for, if, while, default, double, extern, for, and int, etc., C demands that they are used only for giving commands or making statements. You cannot use a default, for example, as the name of a variable. An attempt to do so will result in a compilation error.

Restrictions apply to keywords

- Keywords are the words whose meaning has already been explained to the C compiler and their meanings cannot be changed.
- Keywords can be used only for their intended purpose.
- Keywords cannot be used as user-defined variables.
- All keywords must be written in lowercase.

Keywords have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they

cannot be used as programmer-defined identifiers. Keywords are an essential part of a language definition. They implement specific features of the language. Every C word is classified as either a keyword or an identifier. A keyword is a sequence of characters that the C compiler readily accepts and recognizes while being used in a program. Note that the keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier.

32 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

Table 9: - Keywords in C

4.5. CHARACTER INPUT AND OUTPUT

In 'C' language input and output of data is done by a collection of library functions like getchar, putchar, scanf, printf, gets and puts. These functions permit the transfer of information between the computer and the standard input/output devices. The library function getchar and putchar as the name suggests, allow single characters to be transferred into and out of the computer, scanf and printf permit the transfer of single characters, numerical values and strings, gets and puts facilitate the input and output of strings. An input/ output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of parameters enclosed in parentheses. Some input/output functions do not require parameters, but the empty parentheses must appear. 'C' includes a collection of header file that provide necessary information in support of the various library functions. The header file stdio.h contains the information about input/output library functions.

getchar Function

The getchar function reads a single character from standard input. It takes no parameters and its returned value is the input

character. In general, a reference to the getchar function is written as character variable = getchar();
For example char c;

c= getchar () ;

The second line causes a single character to be entered from the standard input device and then assigned to c. If an end-of-file condition is encountered when reading a character with the getchar function, the value of the symbolic constant EOF will automatically be returned. This function can also be used to read multicharacter strings, by reading one character at a time within a multipass loop.

putchar() Function

The standard C function that prints or displays a single character by sending it to standard output is called putchar. This function takes one argument, which is the character to be sent. It also returns this character as its result. If an error occurs, an error value is returned. Therefore, if the returned value of putchar is used, it should be declared as a function returning an int.

When putchar is used, however, each character must be output separately. The parameter to the function calls in the given statements is character constants, represented between apostrophes as usual. Of course, the arguments could be character variables instead. Two functions that require FILE pointers are getc and putc. These functions are similar to getchar and putchar, except that they can operate on files other than the standard input and output. The getc function is called with one argument, which is a FILE pointer representing the file from which the input is to taken. The expression getc(stdin) is similar to getchar() and the expression putc(c, stdout) is same as putchar(c).

4.5. FORMATTED INPUT AND OUTPUT

These functions read and write all types of data values. Require conversion symbols to identify the data type. The formatted functions return the values after execution. The return value is equal to the number of variables successfully read/written.

Scanf() Function: -

Input data can be entered into the computer from a standard input device by means of C library function scanf. This function can be used to enter any combination of numerical values, characters single character and strings. The function returns the number of data items that have been entered successfully. In general terms, the scanf function is written as

scanf (string, parameter 1, parameter 2…, parameter n);

Where string= string containing certain required formatting information, and Parameter 1, parameter 2.. = parameters that represent the individual input data item. The control string or string comprises individual groups of characters, with one character group for each input data item. Each character group must start with percent sign (%). In the string, multiple character groups can be contiguous, or separated by

white space characters. The conversion character that is used with % sign are many in number and all have different meaning corresponding to type of data item that is to be input from keyboard.

The parameters are written as variables or arrays, whose types match the corresponding characters groups in the control string. Each variable name must be preceded by an ampersand (&).

Conversion character	Meaning
c	type of data item is single character
d	type of data item is decimal integer
e	type of data item is floating-point value
f	type of data item is floating-point value
h	type of data item is short-integer
i	type of data item is decimal, hexadecimal or octal integer
o	type of data item is octal integer
s	type of data item is string
u	type of data item is unsigned decimal integer

Table 10: - Conversion Character

If two or more data items are entered, they must be separated by white space characters. Data items may continue onto two or more lines, since the newline character is considered to be a whitespace character. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.

Printf() function: -

printf function moves data from the computer's memory to the standard output device, whereas the scanf function enters data from the standard input device and stores it in the computer's memory. The general form is:

printf(string, parameter1, parameter2,……, parameter n)

where string refers to a string that contains formatting information, and parameter 1, parameter2… parameter n are arguments that represents the individual output data items. The parameters can be written as constants, single variable or array names or more complex expressions. Unlike scanf function, the parameters in a printf function do not represent memory addresses and therefore they are not preceded by ampersand (&) sign. The control string or string is composed of individual groups of characters, with one character group for each output data item. Each character group must start with a percent sign like in scanf function followed by a conversion character indicating the type of the corresponding data item. Multiple character groups can be contiguous, or they can be separated by other characters, including whitespace characters.

The printf statement to display three variables is,

printf("%s %d %f"', name, roll_no, marks);

4.6.1. The gets() and puts() functions

'C' contains a number of other library functions that permit some form of data transfer into or out of the computer. Gets and puts functions facilitate the transfer of strings between the computer and the standard input/output devices. Each of these functions accepts a single argument or parameter. The parameter must be a data item that represents a string. The string may include whitespace characters. In the case of gets, the string will be entered from the keyboard and will terminate with a newline character. The gets and puts functions are alternative use for scanf and printf for reading and displaying strings.

For example:

```
char school[40];
gets(school);
puts(school);
```

These lines use the gets and puts to transfer the line of text into and out of the computer. When this program is executed, it will give the same result as that with scanf and printf function for input and output of given variable or array.

4.7. INTERACTIVE PROGRAMMING

Interactive programming means to create an interactive dialog between user and computer. This is some sort of question and answer. This can be created by alternate use of scanf and printf functions. This type of interactive programming is useful in the case of useful reports or data entry

4.8. LET US SUM UP

In this unit, you have learnt about the data types of C language supports; keywords used and formatted input and output functions. This knowledge would make you understand the various data types supported by C language and usage of keywords and implementation of formatted input and output functions used in program development. Thus, the data types and input and output operators unit would have brought you to closer to know the concept of data types and the use of input and output functions.

4.9. UNIT – END QUESTIONS

1. Classify the C data types.
2. Describe about the use of keywords in C.
3. How to use the formatted input/output functions in C language?

4.10. ANSWER TO CHECK YOUR PROGRESS

1. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These data types have different storage capacities. C language supports the following different types of data types: Primary data types are fundamental data types in C namely integer (int), floating point(float), character(char)

and void. Derived data types are nothing but primary data types but a little twisted or grouped together like array, structure, union and pointer. Enumerated types are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program. The type void type specifier *void* indicates that no value is available.

2. C programs are constructed from a set of reserved words which provide control and from libraries which perform special functions. The basic instructions are built up using a reserved set of words, such as main, for, if, while, default, double, extern, for, and int, etc., C demands that they are used only for giving commands or making statements. You cannot use a default, for example, as the name of a variable.
3. These functions read and write all types of data values. Require conversion symbols to identify the data type. The formatted functions return the values after execution. The return value is equal to the number of variables successfully read/written.

   Scanf() Function: - Input data can be entered into the computer from a standard input device by means of C library function scanf. This function can be used to enter any combination of numerical values, characters single character and strings.

   Printf() function: - printf function moves data from the computer's memory to the standard output device, whereas the scanf function enters data from the standard input device and stores it in the computer's memory.

4.11. SUGGESTED READINGS

1. "C Primer Plus", Stephen Prata, Addison-Wesley Professional, Sixth Edition, 2013.
2. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
3. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.

BLOCK II: BASICS OF OPERATOR AND DATATYPES

UNIT V – CONTROL STATEMENTS AND DECISION MAKING

Structure

5.1. INTRODUCTION

So far we have seen that in C programs the instructions are executed in the same order in which they appear in the program. Each instruction is executed once and once only. Programs do not include any logical control structures. Most programs, however, require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as looping. Most of the programs require that a logical test be carried out at some particular point within the program. An action will then be carried out whose exact nature depends upon the outcome of the logical test. This is known as conditional execution.

5.2. OBJECTIVES

After going through this lesson you would be able to

- Define 'while' statement, for statement and nested loops.
- Explain switch statement and goto statement.

5.3. THE goto() STATEMENT

The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program. It is written as

goto label;

Where label is an identifier used to label the target statement to which control will be transferred. Control may be transferred to any other statement within the program. The target statement must be labeled, and the label must be followed by a colon. Thus the target statement will appear as

Label : statement

No two statements cannot have the same label Goto statements has many advantages like branching around statements or groups of statements under certain conditions, jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass, jumping completely out of a loop under certain conditions, thus terminating the execution of a loop..

Flow Diagram

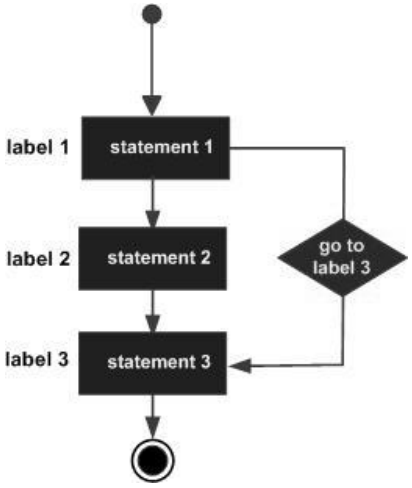

Figure 5: - Goto Statement Flow Diagram

The following example illustrate the functions goto statement,

```
#include<stdio.h>
#include<conio.h>
void main()
{
double x,y;
clrscr();
read:
        scanf("%f", &x);
        if (x<0) goto read;
        y=sqrt(x);
        printf("%f %f". x,y);
        goto read;
getch();
}
```

5.4. THE if STATEMENT

Decision making is an important part of programming. Every programming language supports decision making statements allowing programmers to branch according to the condition. In C programming language**,** if statement is used to check condition and make decision. The decisions or statements are enclosed inside curly braces, however if only a single statement has to be executed, curly braces are not mandatory. Depending upon the number of conditions to be checked, we have following types of if statement:

1. **if statement**
2. **if ... else statement**
3. **Nested if**

Simple if Statement

if statement is used for branching when a single condition is to be checked. The condition enclosed in if statement decides the sequence of execution of instruction. If the condition is true, the statements inside if statement are executed, otherwise they are skipped. In C programming language, any non zero value is considered as true and zero or null is considered false.

Syntax of if statement

```
If (condition)
{
        Statements;
        …. …. ….
        …. …. ….
}
```

Flowchart of if statement

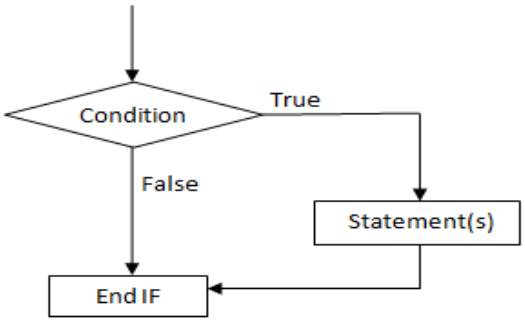

Figure 6: - If Statement Flow Diagram

The following program illustrate the functions of the simple if statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c,d;
float ratio;
clrscr();
printf("Enter four integer values");
```

```
scanf(%d%d%d%d",&a,&b,&c,&d);
if (c-d != 0)
{
ratio = (float)(a+b)/(float)(c-d);
printf("Ratio=%f",ratio);
}
getch();
}
```

5.5. THE if…else STATEMENT

if ... else statement is a two way branching statement. It consists of two blocks of statements each enclosed inside if block and else block respectively. If the condition inside if statement is true, statements inside if block are executed, otherwise statements inside else block are executed. Else block is optional and it may be absent in a program.

Syntax of if...else statement

```
If (test_expression)
{
  //execute your code
}
else
{
  //execute your code
}
```

Flowchart for if…else statement

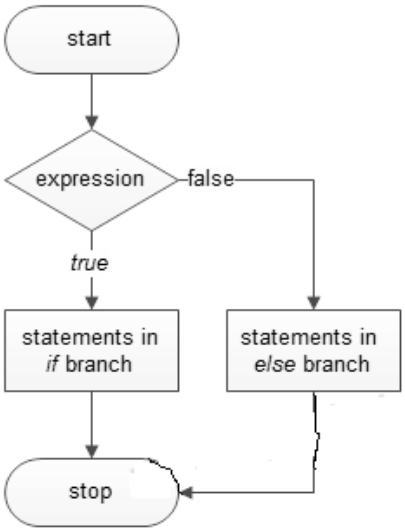

Figure 7: - Flow Diagram of If…Else Statement

Both if and else clause are terminated by semicolons. Let us consider an example of if…else statement.

```
#include main()
{
char grade;
```

```
printf("Enter a character value for grade:");
scanf("%c", &grade);
if(grade= ='A')
printf("grade is excellent \n");
else
printf("grade is other than excellent\n");
}
```

The user can use compound statements both in if and else statements. The first printf is executed if and only if grade is equal to 'A', if grade is not equal to 'A', the first printf is ignored, and the second printf, the one following the word else, is executed.

5.6. Nesting of if STATEMENT

When a if statement is kept inside another if statement, it is called nested if statement. If Else statement in C allows us to print different statements depending upon the expression result (TRUE, FALSE). Sometimes we have to check further even when the condition is TRUE. In this situation, we can use these C Nested IF statements but be careful while using it. Nested if statements are used if there is a sub condition to be tested. The depth of nested if statements depends upon the number of conditions to be checked.

The syntax for a **nested if** statement is as follows −

```
if( boolean_expression 1)
{
  If (boolean_expression 2)
{
    /* Executes when the boolean expression 2 is true */
  }
}
```

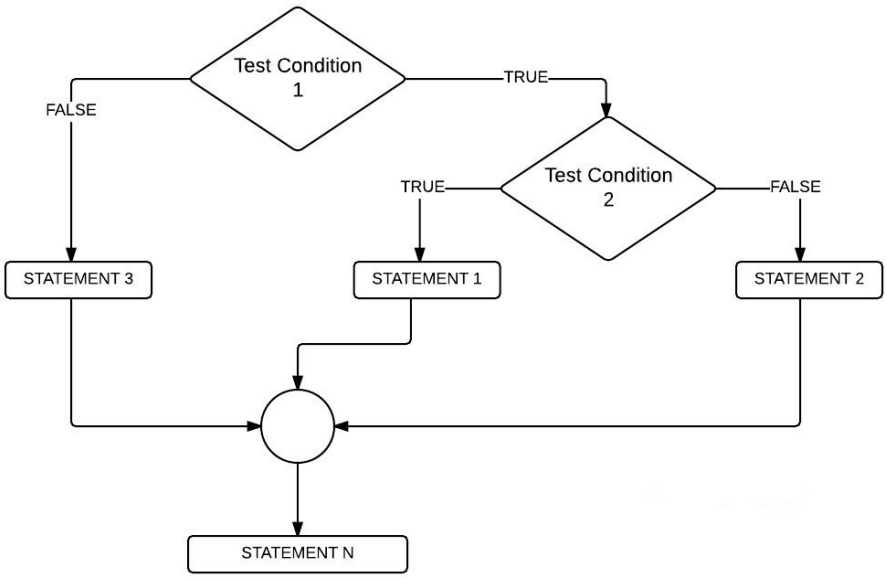

Figure 8: - Flowchart for Nesting of if statement

It is very important to be sure which else clause goes with which if clause. The rule is that each else matches the nearest if preceding it which has not already been matched by an else. Addition of braces prevents any association between the if statement within the braces and the else clause outside them. Even where braces are not necessary, they may still be used to promote clarity.

The following example illustrate the nesting of if…else statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
float a,b,c;
printf("Enter three values");
scanf("%f%f%f",&a,&b,&c);
printf("The Largest Value is");
if (a>b)
{
        if (a>c)
        {
                printf("A is greater");
        else
                printf("C is greater");
        }
        else
        {
                if (c>b)
                        printf("C is greater");
                else
                        printf("B is greater");
        }
}
```

5.7. CONDITIONAL EXPRESSION

The statements

```
If(a>b)
        z=a;
else
        z=b;
```

compute in z the maximum of a and b. The *conditional expression*, written with the ternary operator ``?:", provides an alternate way to write this and similar constructions. In the expression

$expr_1$? $expr_2$: $expr_3$

the expression $expr_1$ is evaluated first. If it is non-zero (true), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated, and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set z to the maximum of a and b,

```
z=(a < b) ? a: b;    /* z = max(a, b) */
```

It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If $expr_2$ and $expr_3$ are of different types, the type of the result is determined by the conversion rules discussed earlier in this chapter. For example, if f is a float and n an int, then the expression

(n > 0) ? f : n

is of type float regardless of whether n is positive.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?:is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

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

printf("%6d %c", a[i], (i%10==9 || i==n-1) ? '\n' : '');

A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent if-else. Another good example is

printf("You save %d items %s \n", n, n==1 ? "" : "s");

5.8. THE SWITCH STATEMENT

The switch statement causes a particular group of statements to be chosen from several available groups. The selection is based upon the current value of on expression that is included within the switch statement. The general form of the switch statement is

switch (expression) statement

Where expression results in an integer value. Expression may also be of type char, since individual characters have equivalent integer values. The embedded statement is generally a compound statement that specifies alternate courses of action. Each alternative is expressed as a group of one or more individual statements within the overall embedded statement. For each alternative, the first statement within the group must be preceded by one or more case labels. The case labels identify the different groups of statements and distinguish them from one another. The case labels must therefore be unique within a given switch statement.

Thus, the switch statement is in effect an extension of the familiar if…else statement. Rather than permitting maximum of only two branches, the switch statement permits virtually any number of branches.

In general terms, each group of statements is written as

```
switch(expression) {
   case constant-expression  :
      statement(s);
      break; /* optional */
      case constant-expression  :
      statement(s);
      break; /* optional */
     /* you can have any number of case statements */
```

```
   default : /* Optional */
   statement(s);
}
```

The following rules apply to a **switch** statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

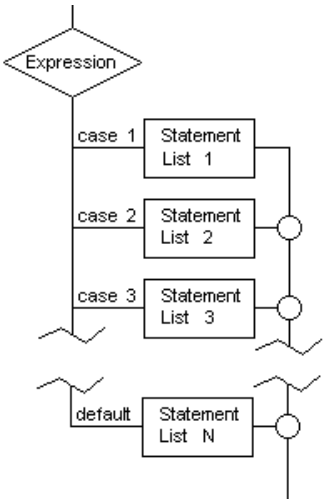

Figure 9: - The Switch Statement Flowchart

Nested Switch

In C, we can have an inner switch embedded in an outer switch. Also, the case constants of the inner and outer switch may have common values and without any conflicts.

The following program illustrates switch statement of C statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c,s;
clrscr();
printf("1. Addition");
printf("2. Subtraction");
printf("3. Multiplication");
printf("4. Division");
```

```
printf("Choose the option");
scanf("%d",&s);
switch(s)
{
case 1:
        c=a+b;
        printf("The Addition result is = %d",c);
        break;
case 2:
        c=a-b;
        printf("The Subtraction result is = %d",c);
        break;
case 3:
        c=a*b;
        printf("The Multiplication result is = %d",c);
        break;
case 4:
        c=a/b;
        printf("The Division result is = %d",c);
        break;

default:
        printf("Wrong Choice");
        break;
}
getch();
}
```

5.9. THE WHILE LOOP

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –

The while statement is used to carry out looping operations. The general form of the statement is

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

The loop operates in the following fashion: The value of the test expression enclosed in parentheses is evaluated. If the result is true, then the program statement (the body of the loop) is executed. The statement may be a compound statement. Then the test expression, which may be just as complex as any of those found in if statement is evaluated again. If it is again true, the statement is executed once more. This process continues until the test expression becomes false. At that point, the loop is terminated immediately,

and program execution continues with the statement (if any) following the while loop. If there are no more statements, the program terminates. Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, the program control passes to the line immediately following the loop.

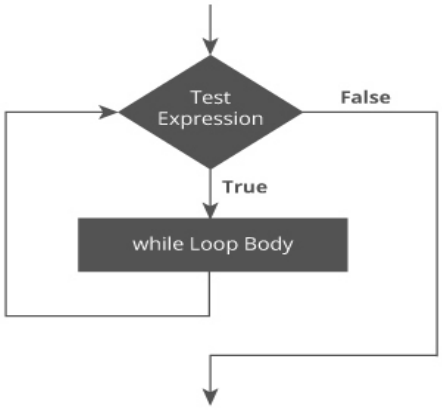

Figure 10: - While Loop Flowchart

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

All variables used in the test expression of the while statement must be initialized at some point before the while loop is reached. In addition, the body of the loop must do something to change the value of the variable used in the expression being tested. Otherwise the condition would remain true and the loop would never terminate. This situation, known as an infinite loop, is illustrated next.

The following program illustrates the while…loop

```
#include<stdio.h>
#include<conio.h>
void main()     {
int sum, n;
sum=0; n=1;
clrscr();
while(n<=10)
{
sum=sum+n*n;
n=n+1;
}
printf("sum=%d",sum);
getch();          }
```

5.10. THE DO…WHILE LOOP

The do… while loop differs from its counterpart, the while loop in that it makes what is called a loop post-test. That is the condition is not tested until the body of the loop has been executed once. In the while loop, by contrast, the

test is made on entry to the loop rather than at the end. The effect is that even if the condition is false when the do-while loop is first encountered, the body of the loop is executed at least once. If the condition is false after the first iteration, the loop terminates. If the first iteration has made the condition true, however the loop continues.

The general form of the do….while loop is as follows:

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

The fact that the while clause is located after the statement reflects the fact that the test is made after the statement is executed. If the body of the loop is a single statement, it must be terminated with a semicolon. For example:

do {
a=a+10; }
while (a < b);

This semicolon marks the end of the inner statement only not of the entire loop construct. In every situation that requires a loop, either one of these two loops can be used.

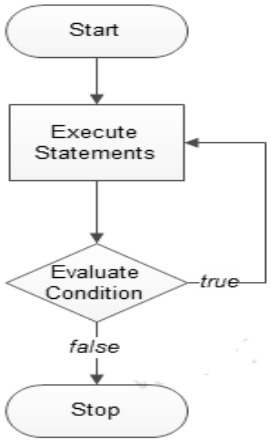

Figure 11: - The do…while loop flowchart

The following program illustrates the do…while statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int sum,i;
i=1;
sum=0;
clrscr();
do
{
        sum=sum + 2;
        i=i+2;
}
while(sum<40 || i<10);
printf("The value of sum is = %d",sum);
getch();
}
```

5.11. THE FOR LOOP

The for statement is the most commonly used looping statement in 'C'. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued and the third expression that allows the index to be modified at the end of each pass.

The general form of the for statement is

for (init; condition; increment) {

statement(s);

}

Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

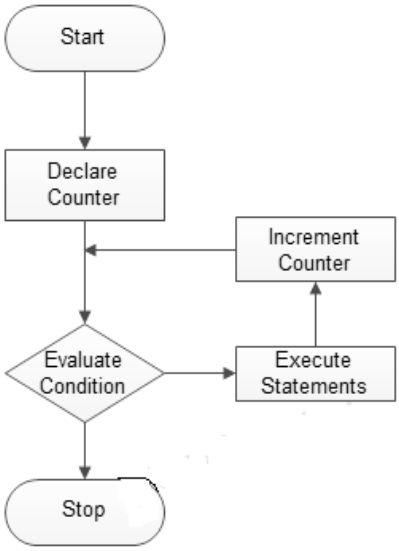

Figure 12: - The For Loop Flowchart

Let us understand the concept of for loop with the help of an example:

```
#include <stdio.h>
#include<conio.h>
void main()
{
int sum,i;
```

```
sum=0; i=1;
for(i=1; i<=10; i++)
{
        sum=sum+i*I;
}
printf("The sum value is = %d", sum);
getch();
}
```

5.12. THE NESTING OF FOR LOOP

\Loops can be nested or embedded one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential; however that one loop be completely embedded within the other there can be no overlap. Each loop must be controlled by a different index.

The syntax for a **nested for loop** statement in C is as follows –

```
for ( init; condition; increment ) {

   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s);
}
```

The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```
#include <stdio.h>
 int main () {
    int i, j;
   for(i = 2; i<100; i++) {
      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }
   return 0;}
```

5.13. THE BREAK AND CONTINUE STATEMENT

The Break Statement

The break statement is used to force fully terminate loops or to exit from a switch. It can be used within a while, a do-while, for or a switch statement. The format is simple as

break;

Without any embedded expression or statements. The break statement causes a transfer of control out of the entire switch statement, to the first statement following the switch statement.

If a break statement is included in a while, in do while or in for loop, then control will immediately be transferred out of the loop when the break

statement is encountered. Thus provides a convenient way to terminate the loop if an error or other irregular condition is detected.

The Continue Statement

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered, instead the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. It can also be included within a while, do while or a for statement as like break statement. It is also written simply as

continue;

without any embedded statement or expression.

5.14. LET US SUM UP

In this unit, you have learnt about the decision making statements of C language; looping statements and other statements supported for decision making and looping statements. This knowledge would make you understand the various decision making statements like simple if, if…else and nested if…else etc. and looping statements like while, do…while and for statements and break and continue statements also. Thus, the control statements and decision making unit would have brought you to closer to know the concept and usage of decision making and control statements too.

5.15. UNIT – END QUESTIONS

1. List out the decision making statements of C language with example.
2. Identify the use of loop control statements with example.
3. What is the use of break and continue statement?

5.16. ANSWER TO CHECK YOUR PROGRESS

1. Decision making is an important part of programming. Every programming language supports decision making statements allowing programmers to branch according to the condition. In C programming language**,** if statement is used to check condition and make decision. The decisions or statements are enclosed inside curly braces, however if only a single statement has to be executed, curly braces are not mandatory. Depending upon the number of conditions to be checked, we have following types of if statement:
   1. if statement. 2. if ... else statement. 3. Nested if. 4. Switch statement.
2. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –
   1. while loop. 2. do…while loop 3. for loop.

3. The break statement is used to force fully terminate loops or to exit from a switch. It can be used within a while, a do-while, for or a switch statement.
   The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered, instead the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.

5.17. SUGGESTED READINGS

1. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.
2. "The C Programming Language", Brain W. Kernighan / Dennis Ritchie, Pearson Publications, 2015.
3. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.

UNIT VI – ARRAYS AND STRINGS

Structure

6.1. INTRODUCTION

As we stated earlier that the variables are the entities in 'C' which are used to hold data in memory. But the concept of variables does not solve the indeterminate number of values is to be stored and operated upon. In case of storing say 100 values; it is a very difficult task to store 100 different variable names with their values. Arrays are the solution to this problem. Arrays are nothing but a single name to a whole group of similar data. The position in terms of arrays is known as subscript. Each subscript must be expressed as a non-negative integer.

6.2. OBJECTIVES

After going through this lesson you will be able in a position to

- Define AN ARRAY
- Process AN ARRAY
- Pass ARRAYS TO FUNCTIONS
- Process MULTIDEMNSIONAL ARRAYS AND STRINGS.
-

6.3. ARRAYS

As we stated earlier that the variables are the entities in 'C' which are used to hold data in memory. But the concept of variables does not solve the indeterminate number of values is to be stored and operated upon. In case of storing say 100 values; it is a very difficult task to store 100 different variable names with their values. Arrays are the solution to this problem. Arrays are nothing but a single name to a whole group of similar data. The position in terms of arrays is known as subscript. Each subscript must be expressed as a non-negative integer.

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

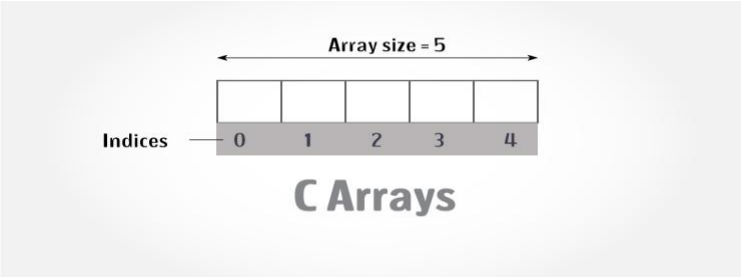

Figure 13: - The Array representation

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Array can be classified into following types

- One Dimensional Array
- Two Dimensional Array
- Multi-Dimensional Array

6.3.1. One Dimensional Array

Arrays are defined in much the same manner as ordinary variables except that each array name must be followed by a size specification. For a one-dimensional array, the size is specified by a positive integer expression enclosed in square brackets.

Declaring Arrays

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

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example,

int a[10];
char text[100];
static char text[100];

The array's size can be defined in terms of a symbolic constant rather than a fixed integer quantity.

Automatic arrays, unlike automatic variables, cannot be initialized. But the definitions of external and static arrays can include the assignment of initial values if required. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas. The general form is

storage class data-type array. name[expression]={value1, value2 - - - , value n};

For example,

```
char school[4] ={'O','P','E','N'};
int marks[10]={1,2,3,4,5,6,7,8,9,10};
static float y[3]={o,0.3,0.2};
```

All individual array elements that are not assigned explicit initial values will automatically be set to zero. This includes the remaining elements of an array in which certain element have been assigned non zero values. The array size need not be specified explicitly when initial values are included as a part of an array definition. With a numerical array, the array size will automatically be set equal to the number of initial values included within the definition.

For example

```
int array[]={4,8,3,7,5};
```

Since the square brackets following the array name are empty, the compiler determines how many elements to allocate for the array by counting the number. of values within the curly braces. This approach can help to avoid errors. If the dimension is specified explicitly, and the curly braces contain more initialization values than are needed, a syntax error is flagged by the compiler. The case of strings is different. The array size specification in this case is usually omitted. The proper array size will be assigned automatically. This will include a provision for the null character.

If a program requires a one-dimensional array declaration the declaration is written in the same manner as the array definition with the following exception. 1. The square brackets may be empty, since the array size will have been specified as a part of the array definition. Array declarations are customarily written in this form. 2. Initial values cannot be included in the declaration. Following are the examples of defining an External array.

Let us understand the concept of one-dimensional array using the following example,

```
#include<stdio.h>
#include<conio.h>
void main()     {
int i;
float a[10], value, total;
printf("Enter 10 real numbers");
for(i=0;i<10;i++)
{
        scanf("%f", &value);
        x[i] = value;
}
total = 0.0;
for (i=0;i<10;i++)
        total=total + x[i] * x[i];
printf("\n");
for (i=0;i<10;i++)
        printf("x[%2d] = %5.2f\n", i+1, x[i]);
```

```
printf("\n total = %.2f\n", total);
getch();          }
```

6.3.2. Passing Arrays to Functions

An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function.

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument or parameter within the function call. The corresponding formal parameter is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

When an array is passed to a function, however, the values of the array element are not passed to the function. But the array name is interpreted as the address of the first array element. This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first array element. Arguments passed in this manner are said to be passed by reference rather than by value. When a reference is made to an array element within the function, the value of the element's subscript is added to the value of the pointer to indicate the address of the specified array element. Therefore, any array element can be accessed from within the function. If an array element is altered within the function, the alteration will be recognized in the calling portion of the program. With the return statement array cannot be used. If the elements of an array are to be passed back to the calling portion of the program, the array must either be defined as an external array or it must be passed to the function as a formal argument.

Way-1

Formal parameters as a pointer –

```
void myFunction(int *param) {
  .
  .
  .
}
```

Way-2

Formal parameters as a sized array −

```
void myFunction(int param[10]) {
  .
  .
  .
}
```

Way-3

Formal parameters as an unsized array –

```
void myFunction(int param[]) {
  .
  .
```

```
  .
}
```

Example

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows –

```
double getAverage(int arr[], int size) {

  int i;
  double avg;
  double sum = 0;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }

  avg = sum / size;

  return avg;
}
```

6.3.3. Multidimensional Arrays

The arrays we have used so far have been one dimensional. The elements of the array could be represented either as a single column or as a single row. A two dimensional array is a grid containing rows and columns, in which each element is uniquely specified by means of its row and column coordinates. Multidimensional arrays are defined in much the same manner as one dimensional array, except that a separate pair of square brackets is required for each subscript. Thus a two dimensional array will require two pairs of square brackets, a three dimensional array will require three pairs of square brackets and so on. A multidimensional array definition can be written as

storage-class data-type array-name[expression1][expression 2] — —[expression n];

The two dimensional array is like a matrix where position of each element is specified by both the column and row numbers.

The row subscript generally is specified before the column subscript. In C, each subscript be written within its own separate pair of brackets.

For example: Definition of Multidimensional arrays are given below.

```
float table[10][10];
char string[10]20];
```

If a multidimensional array definition includes the assignment of initial values, then care must be given to the order in which the initial values are assigned to the array element. The rule is that the last subscript increases most rapidly, and the first subscript increases least

rapidly. Thus, the elements of a two dimensional array will be assigned by rows, that is, the elements of the first row will be assigned, then the elements of the second row, and so on.

Suppose int sample [3][4]={1,2,3,4,5,6,7,8,9,10,11,12}

Thus sample [0][0]=1 sample[0][1]=2 sample [0][2]=3 sample [0][3]=4 sample[1][0]=5 sample[1][1]=6 sample[1][2]=7 sample[1][3]=8 sample[2][0]=9 sample[2][1]=10 sample[2][2]=11 sample[2] [3]=12

Multidimensional arrays are processed in the same manner as one dimensional array, on an element-by-element basis.

Let us understand the concept of multi-dimensional array using the following example

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10][10], b[10][10],i,j,n=3;
clrscr();
printf("Enter the First array values");
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Enter the Second array values");
for (i=0;i<n;i++)
{
for (j=0;j<n;j++)
{
scanf("%d",&b[i][j]);
}
}
printf("Displaying Array Values");
for (i=0;i<n;i++)
{
for (j=0;j<n;j++)
{
scanf("%d",a[i][j]);
}
}
for (i=0;i<n;i++)
{
for (j=0;j<n;j++)
{
scanf("%d",b[i][j]);
}
}
getch();
```

}

6.4. STRINGS

Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C− the C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

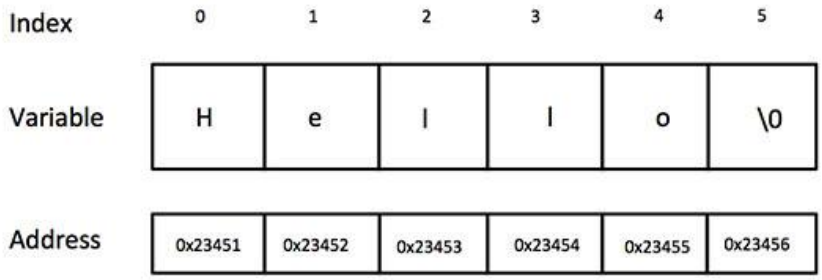

Figure 14: - Strings Memory Representation

```
#include <stdio.h>

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o',
'\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

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

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

Sr. No.	Function & Purpose

1.	**strcpy(s1, s2); -** Copies string s2 into string s1.
2.	**strcat(s1, s2); -** Concatenates string s2 onto the end of string s1.
3.	**strlen(s1); -** Returns the length of string s1.
4.	**strcmp(s1, s2); -** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5.	**strchr(s1, ch); -** Returns a pointer to the first occurrence of character ch in string s1.
6.	**strstr(s1, s2); -** Returns a pointer to the first occurrence of string s2 in string s1.

Table 11: - String Functions

Similarly, strcmp() function compares two strings to find out whether they are same or different. 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 are not, it returns the numeric difference between the ASCII values of the non-matching character.

It will print the same string as entered from the user. The length of string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays. scanf() is not capable of receiving multi word strings. The way to get around this limitation is by using the function gets().

Let us understand the concept of string functions using the following example.

1. strcat() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
char source[ ] = " fresh2refresh" ;
char target[ ]= " C tutorial" ;
printf ( "\nSource string = %s", source ) ;
printf ( "\nTarget string = %s", target ) ;
strcat ( target, source ) ;
printf ( "\nTarget string after strcat( ) = %s", target ) ;
}
```

2. strcpy() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
char source[ ] = "fresh2refresh" ;
char target[20]= "" ;
```

```
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
strcpy ( target, source ) ;
printf ( "\ntarget string after strcpy( ) = %s", target ) ;
return 0;
}
```

3. strlen() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
int len;
char array[20]="fresh2refresh.com" ;
len = strlen(array) ;
printf ( "\string length  = %d \n" , len ) ;
return 0;          }
```

4. strcmp() function

```
#include <stdio.h>
#include <string.h>
int main( )
{
char str1[ ] = "fresh" ;
char str2[ ] = "refresh" ;
int i, j, k ;
i = strcmp ( str1, "fresh" ) ;
j = strcmp ( str1, str2 ) ;
k = strcmp ( str1, "f" ) ;
printf ( "\n%d %d %d", i, j, k ) ;
return 0;
}
```

6.5. LET US SUM UP

In this unit, you have learnt about the arrays and strings of C language. This knowledge would make you understand the arrays, different types of arrays, arrays vs variables and string and string functions. Thus, the arrays and strings unit would have brought you to closer to know the concept and usage of arrays and strings supported by C.

6.6. UNIT – END QUESTIONS

1. Define Array? Explain about the types of array with example.
2. Examine the use of strings and its various functions with example.

6.7. ANSWER TO CHECK YOUR PROGRESS

1. Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a

collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Array can be classified into following types

a. One Dimensional Array
b. Two Dimensional Array
c. Multi-Dimensional Array

2. Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C− the C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

1. strcpy. 2.strcat. 3. strlen. 4. strcmp. 5. strstr 6. strchr

6.8. SUGGESTED READINGS

1. "C Primer Plus", Stephen Prata, Addison-Wesley Professional, Sixth Edition, 2013.
2. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
3. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.

UNIT VII – POINTERS - I

Structure

7.1. INTRODUCTION

As you know by now, variables are stored in memory. Each memory location has a numeric address, in much the same way that each element of an array has its own subscript. Variable names in C and other high-level languages enable the programmer to refer to memory locations by name, but the compiler must translate these names into addresses. A pointer is a variable that represents the location of a data item, such as a variable or an array element.

7.2. OBJECTIVES

After going through this lesson you will be in a position to

- Explain address operator.
- Define pointer declarations describe pointers and one-dimensional arrays.
- Define pointers and multidimensional arrays.
- Explain arrays of pointers.

7.3. BASICS OF POINTERS

Variables are stored in memory. Each memory location has a numeric address, in much the same way that each element of an array has its own subscript. Variable names in C and other high-level languages enable the programmer to refer to memory locations by name, but the compiler must translate these names into addresses. A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are used frequently in C. Pointers can be used to pass information back and forth between a function and its reference point.

The Address Operator

Within a computer's memory, every stored data item occupies one or more contiguous memory cells. The number of memory cells required to store a data item depends on the type of data item. For example, a single character will typically be stored in 1 byte of memory; an integer requires two contiguous bytes. A floating point number may require four contiguous bytes, and a double precision quantity may require eight contiguous bytes. Suppose 'a' is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can be accessed if we know the location of the first memory cell. The address of a's memory location can be determined by the expression &v, where & is a unary operator called the address operator, that evaluates the address of its operand. Now let us assign the address of 'a' to another variable,

Thus

pa= &a;

This new variable is called a pointer to 'a', since it "points" to the locations where 'a' is stored in memory. However, pa represents a's address, not its value. Thus, pa is referred to as a pointer variable.

Address of a → value of a

pa a

The data item represented by 'a' (i.e. the data item stored in a's memory cells) can be accessed by the expression *pa, where * is a unary operator, called the indirection operator, that operates only on a pointer variable. Therefore, *pa and 'a' both represent the same data item (ie the contents of the same memory cells). Further more if we write pa= &a and b = *pa, then 'a' and 'b' will both represents the same value i.e the value of a will indirectly be assigned to 'b

The unary operators & and * are members of the same precedence group as the other unary operators i.e. -, ++, —, ! , size of and(type). The address operator (&) must act upon operands associated with unique addresses, such as ordinary variables or single array elements.

The Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration is somewhat different than the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer, i.e the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as datatype *ptr;

Where ptr is the name of the pointer variable and data-type refers to the data type of the pointer's object.

For example,

```
float a,b;
float *pb;
```

The first line declares a and b to be floating-point variables. The second line declares pb to be a pointer variable whose object is a floating-point quantity i.e. 'pb' point to a floating point quantity. 'pb' represents an address, not a floating-point quantity.

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable.

The Pointers as Arguments

Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form.

When an argument is passed by value, the data item is copied to the function. Thus any alteration made to the data item within the function is not carried over into the calling routine when an argument is passed by reference; however the address of a data item is passed to the function. The contents of that address can be accessed freely, either within the function or within the calling routine. Moreover, any change that is made to the data item will be recognized in both the function and the calling routine. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function.

If formal arguments are pointers, then each must be preceded by an asterisk. Also, if a function declaration is included in the calling portion of the program, the data type of each argument that corresponds to a pointer must be followed by an asterisk.

```
#include main ()
{
int i,j;
i=2;
j=5;
printf("i=%d and j=%d\n", i,j);
swap(&i,&j);
printf("\nnow i=%d and j=%d\n", i,j);
}
swap(i,j)
int *i,*j;
{
int temp=*i;
*i=*j;
*j=temp;
}
```

If the formal parameters i and j of the swap function were declared merely as integers, and the main function passed only i and j rather than their addresses, the exchange made in swap would have no effect on the variables i and j in main. The variable temp in swap function is of type int, not int*. The values being exchanged are not the pointers, but the integers being pointed to by them. Also temp is initialized immediately upon declaration to the value pointed to by i.

7.4. POINTERS AND ONE-DIMENSIONAL ARRAYS

NOTES

Array name is really a pointer to the first element in that array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or x. Similarly, the address of the second array element can be written as either & x [1] or as (x+1) and so on. Here we should keep it in mind that the expression (x+1) is a symbolic representation for an address specification rather than an arithmetic expression. When writing the address of an array element in the form(x+i), there is no need to concern with the number of memory cells associated with each type of array element, the C compiler adjusts for this automatically. The programmer must specify only the address of the first array element and the number of array elements beyond the first. Since &x[i] and (x+i) both represent the address of the ith element of x then x [i] and *(x+i) both represent the contents of that address. The two terms are interchangeable.

When assigning a value to an array element such as x[i] the left side of the assignment statement may be written as either x[i]or *(x+i).

Thus a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element. Thus expression such as x,(x+i) and &x [i] cannot appear on the left side of an assignment statement. Numerical array elements cannot be assigned initial values if the array is defined as a pointer variable. Therefore a conventional array definition is required if initial values will be assigned to the elements of a numerical array. Let us assume that x is to be defined as a one-dimensional, 10 element array of integers. It is possible to define x as a pointer variable rather than as an array. Thus we can write int *x; instead of int x[10]; However x is not automatically assigned a memory block when it is defined as a pointer variable, To assign sufficient memory for x ,we can make use of the library function malloc as follows :

x= malloc (10*size of (int));

This function reserves a block of memory whose size is equivalent to the size of an integer quantity. The function returns a pointer to a character. If the declaration is to include the assignment of initial values, then x must be defined as an array rather then as a pointer variable for example

int x[10]= {1,2,3,4,5,6,7,8,9,10};

or int x[]= {1,2,3,4,5,6,7,8,9,10};

For character arrays you can write them in the form of pointer as char*x="this is string";

7.5. POINTER ARITHMETIC

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

- Increment
- Decrement

- Addition
- Subtraction
- Comparison

7.4.1. Pointer Subtraction and Comparison

An integer value can be added to or subtracted from a pointer variable. For example, px is a pointer variable representing the address of some variable x. We can write expressions such as a ++ px ,- - px (px +3),(px+i) and (px – i) where i is an integer variable. Each expression will represent an address located some distance from the original address represented by px. The exact distance will be the product of the integer quantity and the number. of bytes or words associated with the data item to which px points. One pointer variable can be subtracted from another provided both variables points to elements of the same array. The resulting value indicates the number of words or bytes separating the corresponding array elements.

Pointer variables can be compared provided both variables point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The following points must be kept in mind about operations on pointers:-

- A pointer variable can be assigned the address of an ordinary variable.
- A pointer variable can be assigned the value of another pointer variable provided both pointers point to objects of the same data type.
- An integer quantity can be added to or subtracted from a pointer variable.
- A pointer variable can be assigned a null (zero) value.
- One pointer variable can be subtracted from another provided both pointers point to elements of the same array.
- Two pointer variables can be compared provided both pointers point to objects of the same data type.

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

7.6. POINTER SUBTRACTION AND COMPARISON

As we've seen, you can add an integer to a pointer to get a new pointer, pointing somewhere beyond the original (as long as it's in the same array). For example, you might write

```
ip2 = ip1 + 3;
```

Applying a little algebra, you might wonder whether

```
ip2 - ip1 = 3
```

and the answer is, yes. When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them. You can also ask (again, as long as they point into the same array) whether one pointer is greater or less than another: one pointer is ``greater than'' another if it points beyond where the other one points. You can also compare pointers for equality and inequality: two pointers are equal if they point to the same variable or to the same cell in an array, and are

(obviously) unequal if they don't. (When testing for equality or inequality, the two pointers do not have to point into the same array.)

One common use of pointer comparisons is when copying arrays using pointers. Here is a code fragment which copies 10 elements from array1 to array2, using pointers. It uses an end pointer, ep, to keep track of when it should stop copying.

```
int array1[10], array2[10];
int *ip1, *ip2 = &array2[0];
int *ep = &array1[10];
for(ip1 = &array1[0]; ip1 < ep; ip1++)
        *ip2++ = *ip1;
```

As we mentioned, there is no element array1[10], but it is legal to compute a pointer to this (nonexistent) element, as long as we only use it in pointer comparisons like this (that is, as long as we never try to fetch or store the value that it points to.)

7.7. SIMILARITIES BETWEEN POINTERS & ONE-DIMENSIONAL ARRAYS

There are a number of similarities between arrays and pointers in C. If you have an array

```
int a[10];
```

you can refer to a[0], a[1], a[2], etc., or to a[i] where i is an int. If you declare a pointer variable ip and set it to point to the beginning of an array:

```
int *ip = &a[0];
```

you can refer to *ip, *(ip+1), *(ip+2), etc., or to *(ip+i) where i is an int.

There are also differences, of course. You cannot assign two arrays; the code

```
int a[10], b[10];
a = b;                                  /* WRONG */
```

is illegal. As we've seen, though, you *can* assign two pointer variables:

```
int *ip1, *ip2;
ip1 = &a[0];
ip2 = ip1;
```

Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right does. We haven't copied the data pointed to (there's still just one copy, in the same place); we've just made two pointers point to that one place.

The similarities between arrays and pointers end up being quite useful, and in fact C builds on the similarities, leading to what is called ``the equivalence of arrays and pointers in C.'' When we speak of this ``equivalence'' we do not mean that arrays and pointers are the same thing (they are in fact quite different), but rather that they can be used in related ways, and that certain operations may be used between them.

The first such operation is that it is possible to (apparently) assign an array to a pointer:

```
int a[10];
int *ip;
ip = a;
```

What can this mean? In that last assignment ip = a, aren't we mixing apples and oranges again? It turns out that we are not; C defines the result of this

assignment to be that ip receives a pointer to the first element of a. In other words, it is as if you had written

```
ip = &a[0];
```

The second facet of the equivalence is that you can use the ``array subscripting" notation [i] on pointers, too. If you write

```
ip[3]
```

it is just as if you had written

```
*(ip + 3)
```

So when you have a pointer that points to a block of memory, such as an array or a part of an array, you can treat that pointer ``as if" it *were* an array, using the convenient [i] notation. In other words, at the beginning of this section when we talked about *ip, *(ip+1), *(ip+2), and *(ip+i), we could have written ip[0], ip[1], ip[2], and ip[i]. As we'll see, this can be quite useful (or at least convenient).

The third facet of the equivalence (which is actually a more general version of the first one we mentioned) is that *whenever* you mention the name of an array in a context where the ``value" of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written &array[0]. When you write something like

```
int a[10];
int *ip;
ip = a + 3;
```

it is as if you had written

```
ip = &a[0] + 3;
```

which (and you might like to convince yourself of this) gives the same result as if you had written

```
ip = &a[3];
```

For example, if the character array

```
char string[100];
```

contains some string, here is another way to find its length:

```
int len;
char *p;

for(p = string; *p != '\0'; p++)
        ;
len = p - string;
```

After the loop, p points to the '\0' terminating the string. The expression p - string is equivalent to p - &string[0], and gives the length of the string. (Of course, we could also call strlen; in fact here we've essentially written another implementation of strlen.)

7.8. LET US SUM UP

In this unit, you have learnt about the pointers, pointers with array and pointers arithmetic of C language. This knowledge would make you understand the basics of pointers; the concept of array is implemented in pointer and arithmetic operations of pointers. Thus, the pointers unit would

have brought you to closer to know the concept of pointers and array of pointers.

NOTES

7.9. UNIT – END QUESTIONS

1. Discuss about the basic concepts of pointer with example.
2. Examine the operation of pointer arithmetic with example.

7.10. ANSWER TO CHECK YOUR PROGRESS

1. Variables are stored in memory. Each memory location has a numeric address, in much the same way that each element of an array has its own subscript. Variable names in C and other high-level languages enable the programmer to refer to memory locations by name, but the compiler must translate these names into addresses. A pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are used frequently in C. Pointers can be used to pass information back and forth between a function and its reference point.
2. We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:
   1. Increment. 2. Decrement. 3. Addition. 4. Subtraction. 5. Comparison.

7.11. SUGGESTED READINGS

1. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.
2. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.
3. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
4. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.
5. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.

NOTES

UNIT VIII – POINTERS - II

Structure

8.1. INTRODUCTION

Pointers are used frequently in C. Pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.

8.2. OBJECTIVES

After going through this lesson you will be in a position to

- Explain Null Pointer.
- Define pointers and Strings.
- Define pointers and Two-dimensional arrays.
- Explain arrays of pointers.

8.3. NULL POINTERS

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>
int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
```

```
  return 0;
}
```

When the above code is compiled and executed, it produces the following result − The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

NULL Pointer is a pointer which is pointing to nothing. NULL pointer points the base address of segment. In case, if you don't have address to be assigned to pointer then you can simply use NULL. Pointer which is initialized with NULL value is considered as NULL pointer.

NULL is macro constant defined in following header files –

- stdio.h
- alloc.h
- mem.h
- stddef.h
- stdlib.h

NOTES

8.4. POINTER AND STRINGS

A String is a sequence of characters stored in an array. A string always ends with null ('\0') character. Simply a group of characters forms a string and a group of strings form a sentence.

Creating a pointer for the string

The variable name of the string str holds the address of the first element of the array i.e., it points at the starting memory address.

So, we can create a character pointer ptr and store the address of the string str variable in it. This way, ptr will point at the string str. In the following code we are assigning the address of the string str to the pointer ptr.

char *ptr = str;

We can represent the character pointer variable ptr as follows.

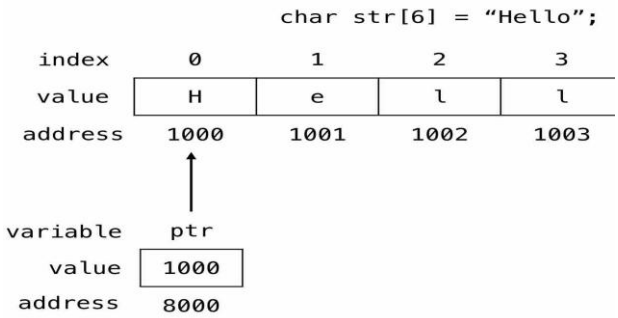

Figure 15: - Character Pointer

A pointer to array of characters or string can be looks like the following:

NOTES

```
#include <stdio.h>
int main() {
char *cities[] = {"Iran", "Iraq"};
int i;
for(i = 0; i < 2; i++)
printf("%s\n", cities[i]);
return 0; }
```

In the above pointer to string program, we declared a pointer array of character datatypes and then few strings like "Iran", "Iraq" where initialized to the pointer array (*cities[]). Note that we have not declared the size of the array as it is of character pointer type. Coming to the explanation, cities[] is an array which has its own address and it holds the address of first element (I (Iran)) in it as a value. This address is then executed by the pointer, i.e) pointer start reading the value from the address stored in the array cities[0] and ends with '\0' by default. Next cities[1] holds the address of (I (Iraq).This address is then executed by the pointer, i.e) pointer start reading the value from the address stored in the array cities[1] and ends with '\0' by default. As a result Iran and Iraq is outputted.

8.5. POINTER AND TWO DIMENSIONAL ARRAYS

A two dimensional array, is actually a collection of one dimensional arrays. Therefore, we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two dimensional array declaration can be written as

data type (*ptr) [expression2];

Notice the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present. Without them we would be defining an array of pointers rather than a pointer to a group of arrays, since these particulars symbols (i.e. The square brackets and asterisk) would normally be evaluated right-to-by.

For example, if x is a two dimensional integer array having 10 rows and 20 columns, then we can declare x as

int(*x)[20];

rather than int x(10) (20);

Similarly, three dimensional integer array can be written as int (*x) (20) (30)

An individual array element within a multi-dimensional array can be accessed by repeatedly using the indirection operator. If n is a 2D array having 10 rows and 20 columns, then item in row 2, column 5 can be accessed by writing either

x[2][5]

or

((x+2)+5)

Here, (x+2) is a pointer to row 2. Therefore, the object of this pointer *(x+2), refers to the entire row. Since row 2 is a one-dimensional array, *(x+2) is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence,(*(x+2)x5) is a pointer to element 5 in row 2. The object

of this pointer, (*(x+2)+5) therefore refers to the item in column 5 of row2 which is x[2][5].

8.6. ARRAYS OF POINTERS

NOTES

A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such cases the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n-1) dimensional array. In general terms, a 2D array can be defined as a one dimensional array of pointers by writing.

data type *array [expression 1];

In this type of declaration, array name and its preceding asterisk are not enclosed in parentheses. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

For example x is a 2D integer array having 10 rows and 20 columns. We can define x as a one-dimensional array of pointers by writing

int *x[10];

An individual array element, such as x[2][5], can be accessed by writing *(x[2]+5).

In this expression, x[2] is a pointer to the first element in row 2, so that (x[2] +5) points to element 5 within row 2. The object of this pointer, *(x[2]+5), therefore refers to x[2][5].

Pointer arrays offer a particularly convenient method for storing strings. In this situation, each array element is a character-type pointer that indicates the beginning of a separate string. Each individual string can be accessed by referring to its corresponding pointer.

An advantage to use arrays of pointers is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional array. If some of the strings are particularly long, there is no need to worry about the possibility of exceeding some maximum specified string length. Arrays of this type are often referred to as ragged arrays.

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[MAX];

It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows –

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];
  for ( i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; /* assign the address of integer. */
  }
  for ( i = 0; i < MAX; i++) {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
```

NOTES

```
  }
  return 0;
}
```

An advantage to use arrays of pointers is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional array. If some of the strings are particularly long, there is no need to worry about the possibility of exceeding some maximum specified string length. Arrays of this type are often referred to as ragged arrays.

8.7. LET US SUM UP

In this unit, you have learnt about the pointers with array and pointers with strings supported by C language. This knowledge would make you understand the concept of array is implemented in pointer and understand the strings to be implemented in pointer too. Thus, the pointers unit would have brought you to closer to know the concept of pointers with array and pointers with strings.

8.8. UNIT – END QUESTIONS

1. Analyze the concept of pointers with array and explain with example.
2. Write about the pointers with strings using a simple example.

8.9. ANSWER TO CHECK YOUR PROGRESS

1. A multi-dimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays. In such cases the newly defined array will have one less dimension than the original multi-dimensional array. Each pointer will indicate the beginning of a separate (n-1) dimensional array. In general terms, a 2D array can be defined as a one dimensional array of pointers by writing. A two dimensional array, is actually a collection of one dimensional arrays. Therefore, we can define a two dimensional array as a pointer to a group of contiguous one-dimensional arrays.

   .

2. A String is a sequence of characters stored in an array. A string always ends with null ('\0') character. Simply a group of characters forms a string and a group of strings form a sentence. The variable name of the string str holds the address of the first element of the array i.e., it points at the starting memory address.

   So, we can create a character pointer ptr and store the address of the string str variable in it. This way, ptr will point at the string str. In the following code we are assigning the address of the string str to the pointer ptr. **char *ptr = str;**

8.10. SUGGESTED READINGS

1. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
2. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.

3. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006

NOTES

BLOCK III: ARRAY CONCEPTS, POINTERS & FUNCTION

UNIT IX – STRUCTURES AND UNIONS

Structure

9.1. INTRODUCTION

We studied earlier that array is a data structure whose element are all of the same data type. Now we are going towards structure, which is a data structure whose individual elements can differ in type. Thus a single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. This lesson is concerned with the use of structure within a 'c' program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationship between structures and pointers, arrays and functions will also be examined. Closely associated with the structure is the union, which also contains multiple members.

9.2. OBJECTIVES

After going through this lesson you will be able to

- Explain the basic concepts of structure
- Process a structure
- Use typedef statement
- Explain the between structures and pointers relate structure to a function
- Explain the concept of unions

9.3. STRUCTURES

Array is a data structure whose element is all of the same data type. Now we are going towards structure, which is a data structure whose individual elements can differ in type. Thus a single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

9.3.1. Basics of Structures

Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types. Structure definition is as follows,

General format:

```
struct tag_name
{
data type member1;
data type member2;
…
…
}
```

In this declaration, struct is a required keyword; tag is a name that identifies structures of this type. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distict from one another, though a member name can be same as the name of a variable defined outside of the structure. A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure-type declaration.

For example:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
```

we can now declare the structure variable s1 and s2 as follows:

struct student s1, s2;

s1 and s2 are structure type variables whose composition is identified by the tag student.

It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

storage- class struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
} variable 1, variable 2 --------- variable n;

The tag is optional in this situation.

struct student
{
char name [80];
int roll_no;
float marks;
}s1,s2;

The s1, s2, are structure variables of type student. Since the variable declarations are now combined with the declaration of the structure type, the tag need not be included. As a result, the above declaration can also be written as struct{ char name [80]; int roll_no; float marks ; } s1, s2, ; A structure may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure. The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

storage-class struct tag variable = { value1, value 2,-------, value m};

A structure variable, like an array can be initialized only if its storage class is either external or static.

Processing a Structure

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable.member name.

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.

e.g. if we want to print the detail of a member of a structure then we can write as

printf(“%s”,st.name); or printf(“%d”, st.roll_no) and so on. More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing.

variable.member.submember.

Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

st.d1.month

The use of the period operator can be extended to arrays of structure, by writing

array [expression]. member

Structures members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions. They can be passed to functions and they can be returned from functions, as though they were ordinary single-valued variables.

e.g. suppose that s1 and s2 are structure variables having the same composition as described earlier. It is possible to copy the values of s1 to s2 simply by writing

s2=s1;

It is also possible to pass entire structure to and from functions though the way this is done varies from one version of 'C' to another.

9.3.2. Array of Structures

It is possible to define an array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can also make array of structures. In the first example in structures, we stored the data of 3 students. Now suppose we need to store the data of 100 such children. Declaring 100 separate variables of structure is definitely not a good option. For that, we need to create an array of structures.

For example, we are storing employee details such as name, id, age, address and salary. Normally we group them as employee structure with the above mentioned members. We can create the structure variable to access or modify the structure members. A company may have 10 to 100 employees, how about storing the same for 100 employees?

In C Programming, We can easily solve the above mentioned problem by combining 2 powerful concepts Arrays and Structures in C. We can create the employee structure with the above mentioned members and then instead of creating the structure variable, we create the array of structure variable.

```
.       /* Array of Structures in C Initialization */
struct Employee
{
 int age;
 char name[50];
 int salary;
} Employees[4] = {
            {25, "Suresh", 25000},
            {24, "Tutorial", 28000},
            {22, "Gateway", 35000},
            {27, "Mike", 20000}
            };
```

Here, Employee structure is used for storing the employee details such as age, name and salary. We created the array of structures variable Employees [4] (with size 4) at the declaration time only. We also initialized the values of each and every structure member for all the 4 employees.

9.3.3. Pointers to Structures

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if variable represents a structure type variable, then & variable represents the starting address of that variable. We can declare a pointer variable for a structure by writing

type *ptr;

Where type is a data type those identities the composition of the structure, and ptr represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

ptr= &variable;

Let us take the following example:

```
typedef struct
{
char name [ 40];
int roll_no;
float marks;
}student; student s1,*ps;
```

In this example, s1 is a structure variable of type student, and ps is a pointer variable whose object is a structure variable of type student. Thus, the beginning address of s1 can be assigned to ps by writing.

ps = &s1;

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

ptr →member

Where ptr refers to a structure- type pointer variable and the operator → is comparable to the period (.) operator. The associativity of this operator is also left-to-right.

The operator → can be combined with the period operator (.) to access a submember within a structure. Hence, a submember can be accessed by writing

ptr → member.submember

9.3.4. Self-referential Structures

An array is a collection of homogeneous elements but in the real world, we may need to include different types of logically related data.

Types of Self Referential Structures

- Self Referential Structure with Single Link
- Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members.

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time.

To store different data such as name, age, contact, etc. associated with a single entity, we make use of structures. A structure is, therefore,

capable of storing heterogeneous data under a single name and the data elements are called as members**.** A structure that contains pointers to a structure of its own type is known as self-referential structure.

In other words, a self-referential C structure is the one which includes a pointer to an instance of itself.

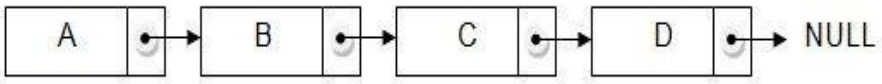

Figure 16: - C Linked List

Syntax of Self-Referential Structure in C Programming

```
struct demo
{
        datatype member1, member2;
        struct demo *ptr1, *ptr2;
}
```

As you can see in the syntax, ptr1 and ptr2 are structure pointers that are pointing to the structure demo, so structure demo is a self referential structure. These types of data structures are helpful in implementing data structures like linked lists and trees.

Self Referential Structure Example

```
struct node
{
    int data;
    struct node *link;
};
```

The concept of linked lists, stacks, queues, trees and many others works on the principle of self-referential structures.

9.4. UNIONS

Union, like structures, contains members whose individual data types may differ from one another. However, the members that compose a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory.

In general terms, the composition of a union may be defined as

```
union tag
{
member1;
member 2;
- - -
member m };
```

Where union is a required keyword and the other terms have the same meaning as in a structure definition. Individual union variables can then be declared as storage-class union tag variable1, variable2, -----, variable n; where storage-class is an optional storage class specifier, union is a required

keyword, tag is the name that appeared in the union definition and variable 1, variable 2, variable n are union variables of type tag.

The two declarations may be combined, just as we did in the case of structure. Thus, we can write.

Storage-class union tag
{
member1;
member 2;
- - -
member m }variable 1, varibale2, ..., variable n;

The tag is optional in this type of declaration. Let us take a 'C' program which contains the following union declaration:

union code
{
char color [5];
int size ;
}purse, belt;

Here we have two union variables, purse and belt, of type code. Each variable can represent either a 5–character string (color) or an integer quantity (size) of any one time.

A union may be a member of a structure, and a structure may be a member of a union. An individual union member can be accessed in the same manner as an individual structure members, using the operators (→) and. Thus if variable is a union variable, then varibale.member refers to a member of the union. Similarly, if ptr is a pointer variable that points to a union, then ptr→ member refers to a member of that union.

Unions are processed in the same manner, and with the same restrictions as structures. Thus, individual union members can be processed as though they were ordinary variables of the same data type and pointers to unions can be passed to or from functions.

9.5. LET US SUM UP

In this unit, you have learnt about the basics of structures, arrays of structures, pointers to structures and union supported by C language. This knowledge would make you understand the basic concept of structure; array is implemented with structure and understands the concept of union. Thus, the structures and unions unit would have brought you to closer to know the concept of structures and unions.

9.6. UNIT – END QUESTIONS

1. Describe about the structures and its implementation with example.
2. Describe about the union and differentiate the structure and union.

9.7. ANSWER TO CHECK YOUR PROGRESS

1. Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types. The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if variable represents a structure type variable, then & variable represents the starting address of that variable.
.
2. Union, like structures, contains members whose individual data types may differ from one another. However, the members that compose a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. Unions are processed in the same manner, and with the same restrictions as structures. Thus, individual union members can be processed as though they were ordinary variables of the same data type and pointers to unions can be passed to or from functions.

9.8. SUGGESTED READINGS

1. "C Primer Plus", Stephen Prata, Addison-Wesley Professional, Sixth Edition, 2013.
2. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
3. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.
4. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.
5. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.
6. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.
7. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
8. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.

UNIT X – FUNCTIONS

Structure

10.1. INTRODUCTION

In the earlier lessons we have already seen that C supports the use of library functions, which are used to carry out a number of commonly used operations or calculations. C also allows programmers to define their own functions for carrying out various individual tasks. In this lesson we will cover the creation and utilization of such user defined functions.

10.2. OBJECTIVES

After going through this lesson you will be able to

- Explain of function.
- Describe access to function.
- Define parameters data types' specification.
- Explain function prototype and recursion.
-

10.3. FUNCTIONS

10.3.1. Function Philosophy

The use of user-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Thus a C program can be modularized through the intelligent use of such functions. There are several advantages to this modular approach to program development. For example many programs require a particular group of instructions to be accessed repeatedly from several different places within a program. The repeated instruction can be placed within a single function, which can then be accessed whenever it is needed. Moreover, a different set of data can be

transferred to the function each time it is accessed. Thus, the use of a function avoids the need for redundant (repeating) programming of the same instructions. The decomposition of a program into individual program modules is generally considered to be an important part of good programming.

10.3.2. Function Basics

Function is a self-contained program segment that carries out some specific well-defined task. Every C program consists of one or more functions. The most important function is main. Program execution will always begin by carrying out the instruction in main. The definitions of functions may appear in any order in a program file because they are independent of one another. A function can be executed from anywhere within a program. Once the function has been executed, control will be returned to the point from which the function was accessed. Functions contain special identifiers called parameters or arguments through which information is passed to the function and from functions information is returned via the return statement. It is not necessary that every function must return information, there are some functions also which do not return any information for example the system defined function printf. Before using any function it must be defined in the program. Function definition has three principal components: the first line, the parameter declarations and the body of the functions. The first line of a function definition contains the data type of the information return by the function, followed by function name, and a set of arguments or parameters, separated by commas and enclosed in parentheses. The set of arguments may be skipped over. The data type can be omitted if the function returns an integer or a character. An empty pair of parentheses must follow the function name if the function definition does not include any argument or parameters.

The general term of first line of functions can be written as:

data-type function-name (formal argu 1, formal argu 2…formal argument n)

The formal arguments allow information to be transferred from the calling portion of the program to the function. They are also known as parameters or formal parameters. These formal arguments are called actual parameters when they are used in function reference. The names of actual parameters and formal parameters may be either same or different but their data type should be same. All formal arguments must be declared after the definition of function. The remaining portion of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the body of the function. This compound statement can contain expression statements, other compound statements, control statements etc. Information is returned from the function to the calling portion of the program via the return statement. The return statement also causes control to be returned to the point from which the function was accessed.

A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions.

```
main( )
```

```
{
message( ) ;
printf ( "\nCry, and you stop the monotony!" ) ;
}
message( )
{
printf ( "\nSmile, and the world smiles with you..." ) ;
}
```

And here's the output...

```
Smile, and the world smiles with you...
Cry, and you stop the monotony!
```

Here, main( ) itself is a function and through it we are calling the function message( ). What do we mean when we say that main( ) 'calls' the function message( )? We mean that the control passes to the function message( ). The activity of main( ) is temporarily suspended; it falls asleep while the message( ) function wakes up and goes to work. When the message( ) function runs out of statements to execute, the control returns to main( ), which comes to life again and begins executing its code at the exact point where it left off. Thus, main( ) becomes the 'calling' function, whereas message( ) becomes the 'called' function.

In general terms, the return statement is written as

return expression;

The value of the expression is returned to the calling portion of the program. The return statement can be written without the expression. Without the expression, return statement simply causes control to revert back to the calling portion of the program without any information transfer. The point to be noted here is that only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return. But a function definition can include multiple return statements, each containing a different expression. Functions that include multiple branches often require multiple returns.

It is not necessary to include a return statement altogether in a program. If a function reaches the end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information.

If you have grasped the concept of 'calling' a function you are prepared for a call to more than one function.

From this program a number of conclusions can be drawn:

- Any C program contains at least one function.
- If a program contains only one function, it must be **main( )**.
- If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because program execution always begins with **main( )**.
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in **main( )**.
- After each function has done its thing, control returns to **main( )**.When **main( )** runs out of function calls, the program ends.

As we have noted earlier the program execution always begins with **main( ).** Except for this fact all C functions enjoy a state of perfect equality. No precedence, no priorities, nobody is nobody's boss. One function can call another function it has already called but has in the meantime left temporarily in order to call a third function which will sometime later call the function that has called it.

Why use Function?

Two reasons:

(a) Writing functions avoids rewriting the same code over and over.

(b) Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

Passing Values between Functions

The functions that we have used so far haven't been very flexible. In short, now we want to communicate between the 'calling' and the 'called' functions.

The mechanism used to convey information to the function is the 'argument'. You have unknowingly used the arguments in the printf( ) and scanf( ) functions; the format string and the list of variables used inside the parentheses in these functions are arguments. The arguments are sometimes also called 'parameters'.

10.3.3. Function Prototypes

A function prototype is a declaration that indicates the types of the function's parameters as well as its return value. For example:

```
double pow( double, double );    // prototype of pow()
```

This prototype informs the compiler that the function pow() expects two arguments of type double, and returns a result of type double. Each parameter type may be followed by a parameter name. This name has no more significance than a comment, however, since its scope is limited to the function prototype itself. For example:

```
double pow( double base, double exponent );
```

Functions that do not return any result are declared with the type specifier void. For example:

```
void func1( char *str ); // func1 expects one string
                         // argument and has no return
                         // value.
```

Functions with no parameters are declared with the type specifier void in the parameter list:

```
int  func2( void );     // func2 takes no arguments and
                        // returns a value with type int.
```

Function declarations should always be in prototype form. All standard C functions are declared in one (or more) of the standard header files. For example, *math.h* contains the prototypes of the mathematical functions, such as sin(), cos(), pow(), etc., while *stdio.h* contains the prototypes of the standard input and output functions.

Function Definitions

The general form of a function definition is:

```
[storage_class] [type] name(
[parameter_list] ) // function declarator
{
   /* declarations, statements */      // function body
}
```

storage_class

One of the storage class specifiers extern or static. Because extern is the default storage class for functions, most function definitions do not include a storage class specifier.

type

The type of the function's return value. This can be either void or any other type, except an array.

name

The name of the function.

parameter_list

The declarations of the function's parameters. If the function has no parameters, the list is empty.

Here is one example of a function definition:

```
long sum( int arr[], int len )// Find the sum of the first
{                             // len elements of the array arr
  int i;
  long result = 0;
  for( i = 0;  i < len;  ++i )
    result += (long)arr[i];
  return result;
}
```

Because by default function names are external names, the functions of a program can be distributed among different source files, and can appear in any sequence within a source file.

Functions that are declared as static, however, can only be called in the same translation unit in which they are defined. But it is not possible to define functions with block scope—in other words, a function definition cannot appear within another function.

The parameters of a function are ordinary variables whose scope is limited to the function. When the function is called, they are initialized with the values of the arguments received from the caller.

The statements in the function body define what the function does. When the flow of execution reaches a return statement or the end of the function body, control returns to the calling function.

A function that calls itself, directly or indirectly, is called recursive. C permits the definition of recursive functions, since variables with automatic storage class are created anew—generally in stack memory—with each function call.

The function declarator shown above is in prototype style. Today's compilers still support the older Kernighan-Ritchie style, however, in which

the parameter identifiers and the parameter type declarations are separate. For example:

```
long sum( arr, len )  // Parameter identifier list
int arr[], len;       // Parameter declarations
{ ... }               // Function body
```

In ANSI C99, functions can also be defined as inline. The inline function specifier instructs the compiler to optimize the speed of the function call, generally by inserting the function's machine code directly into the calling routine. The inline keyword is prefixed to the definition of the function:

```
inline int max( int x, int y )
{  return  ( x >= y ? x : y ); }
```

If an inline function contains too many statements, the compiler may ignore the inline specifier and generate a normal function call.

An inline function must be defined in the same translation unit in which it is called. In other words, the function body must be visible when the inline "call" is compiled. It is therefore a good idea to define inline functions—unlike ordinary functions—in a header file.

Inline functions are an alternative to macros with parameters. In translating a macro, the preprocessor simply substitutes text. An inline function, however, behaves like a normal function—so that the compiler tests for compatible arguments, for example—but without the jump to and from another code location.

10.3.4. Passing Arguments

Arguments can be passed to a function by two methods, they are called passing by value and passing by reference. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Let us consider an example

```
#include main()
{
int x=3;
printf(“\n x=%d(from main, before calling the function”),x);
change(x);
printf(“\n\nx=%d(from main, after calling the function)”,x);
}
change(x) int x;
{
x=x+3;
printf(“\nx=%d(from the function, after being modified)”,x);
return;
}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum

up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from change.

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

Arrays are passed differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine.

10.3.5. Passing parameter by value and Passing parameter by Reference

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
void swap(int x, int y) {
  int temp;
  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put temp into y */
  return; }
```

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

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
void swap(int *x, int *y) {
  int temp;
  temp = *x;    /* save the value at address x */
  *x = *y;      /* put y into x */
  *y = temp;    /* put temp into y */
  return; }
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

10.3.6. Passing string to function

A string is a sequence of characters enclosed in double quotes. For example, "Hello World" is a string and it consists of a sequence of English letters in both uppercase and lowercase and the two words are separated by a white space. So, there are total 11 characters.

We know that a string in C programming language ends with a NULL \0 character. So, in order to save the above string we will need an array of size 12. The first 11 places will be used to store the words and the space and the 12th place will be used to hold the NULL character to mark the end.

Function declaration to accept one dimensional string

We know that strings are saved in arrays so, to pass an one dimensional array to a function we will have the following declaration.

returntype fuctionname(char str[ ])

Example

void displayString(char str[ ])

In the above example we have a function by the name displayString and it takes an argument of type char and the argument is an one dimensional array as we are using the [] square brackets.

Passing one dimensional string to a function

To pass a one dimensional string to a function as an argument we just write the name of the string array variable.

In the following example we have a string array variable message and it is passed to the displayString function.

10.3.7. Passing array to function

In C, there are various general problems which require passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

functionname(arrayname);//passing array

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

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

First way:

return_type function(type arrayname[])

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

Second way:

return_type function(type arrayname[SIZE])

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

Third way:

return_type function(type *arrayname)

Returning array from the function

As we know that, a function cannot return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a,b,c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

```
int * Function_name() {
//some statements;
return array_type;
}
```

To store the array returned from the function, we can define a pointer which points to that array. We can traverse the array by increasing that pointer since pointer initially points to the base address of the array.

10.3.8. Structures and Functions

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

Passing Structure to Function In C:

It can be done in below 3 ways.

- Passing structure to a function by value
- Passing structure to a function by address(reference)
- No need to pass a structure – Declare structure variable as global

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.

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.

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

10.3.9. Recursion

Many C compilers permits each of the argument data types within a function declaration to be followed by an argument name, that is

data-type function name (type1 argument 1, type 2 argument2… type n argument n);

Function declarations written in this form are called function prototypes.

Function prototypes are desirable, however, because they further facilitate error checking between the calls to a function and the corresponding function definition. Some of the function prototypes are given below:

int example (int, int); or int example (int a, int b);

void example 1(void); or void example 1(void);

void fun (char, long); or void fun (char c, long f);

The names of the arguments within the function declaration need not be declared elsewhere in the program, since these are "dummy" argument names recognized only within the declaration. "C" language also permits the useful feature of 'Recursion'.

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a precious result. In order to solve a problem recursively, two conditions must be satisfied. The problem must be written in a recursive form, and the problem statement must include a stopping condition. The best example of recursion is calculation of factorial of a integer quantity, in which the same procedure is repeating itself. Let us consider the example of factorial:

```
#include main()
{       int number;
long int fact(int number);
printf("Enter number");
scanf("%d", & number);
printf("Factorial of number is % d\n", fact(number)); }
        long int fact(int number)
{       if(number <=1) return(1);
else return(number *fact(number-1)); }
```

The point to be noted here is that the function 'fact' calls itself recursively, with an actual argument (n-1) that decrease in value for each successive call. The recursive calls terminate the value of the actual argument becomes equal to 1.

When a recursive program is executed, the recursive function calls are not executed immediately. Instead of it, they are placed on a stack until the condition that terminates the recursion is encountered. The function calls are then executed in reverse order, as they are popped off the stack. The use of recursion is not necessarily the best way to approach a problem, even though the problem definition may be recursive in nature.

10.4. LET US SUM UP

In this unit, you have learnt about the basics of functions, function prototypes, function arguments, passing parameters and recursion function of C language. This knowledge would make you understand the basic concept of functions; structure of a function and type of arguments used, and understands the recursion function also. Thus, the functions unit would have brought you to closer to know the concept of functions of C language.

10.5. UNIT – END QUESTIONS

1. Explain the basic concept of functions and its prototype with example.
2. Discuss about the types of passing parameters with example.
3. Define recursion function with example.

10.6. ANSWER TO CHECK YOUR PROGRESS

1. Function is a self-contained program segment that carries out some specific well-defined task. Every C program consists of one or more functions. The most important function is main. Program execution will always begin by carrying out the instruction in main. The definitions of functions may appear in any order in a program file because they are independent of one another. A function can be executed from anywhere within a program. Once the function has been executed, control will be returned to the point from which the function was accessed. Functions contain special identifiers called parameters or arguments through which information is passed to the function and from functions information is returned via the return statement..
2. Arguments can be passed to a function by two methods, they are called passing by value and passing by reference. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.
3. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a precious result. In order to solve a problem recursively, two conditions must be satisfied. The problem must be written in a recursive form, and the problem statement must include a stopping condition.

10.7. SUGGESTED READINGS

1. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.

2. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
3. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.
4. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.

NOTES

UNIT XI – STORAGE CLASSES

Structure

11.1. INTRODUCTION

We learn through this lesson as Storage class in C decides the part of storage to allocate memory for a variable; it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable.

11.2. OBJECTIVES

After going through this lesson you will be able to

- Explain of storage classes and visibility.
- Describe access to Pointers and Strings.
- Define Global and Static Variables.
- Explain external variables.

11.3. STORAGE CLASSES AND VISIBILITY

Storage class in C decides the part of storage to allocate memory for a variable; it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable. There are four storage classes in C those are *automatic*, *register*, *static*, and *external*.

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.

Thus a storage class is used to represent the information about a variable.

A variable is not only associated with a data type, its value but also a storage class. There are total four types of standard storage classes. The table below represents the storage classes in 'C'.

Storage Class Specifiers

There are four storage class Specifiers in C as follows, typedef specifier does not reserve storage and is called a storage class specifier only for syntactic convenience. It is not a storage class specifier in the common meaning.

- auto
- register
- extern
- static
- typedef

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses a storage class is shown here:

storage_class_specifier data_type variable_name;

At most one storage class specifier may be given in a declaration. If no storage class specifier is specified then following rules are used:

- Variables declared inside a function are taken to be auto.
- Functions declared within a function are taken to be extern.
- Variables and functions declared outside a function are taken to be static, with external linkage.

Variables and functions having external linkage are available to all files that constitute a program. File scope variables and functions declared as static (described shortly) have internal linkage. These are known only within the file in which they are declared. Local variables have no linkage and are therefore known only within their own block.

Types of Storage Classes

There are four storage classes in C they are as follows:

- Automatic Storage Class
- Register Storage Class
- Static Storage Class
- External Storage Class

11.4. AUTOMATIC OR LOCAL POINTERS AND STRINGS

A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is

mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.
The following C program demonstrates the visibility level of auto variables.

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

Automatic variables are always declared within a function and are local to the function in which they are declared, that is their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another. The location of the variable declarations within the program determine the automatic storage class, the keyword auto is not required at the beginning of each variable declaration.

These variables can be assigned initial value by including appropriate expressions within the variable declarations. An automatic variable does not retain its value once control is transferred out of its defining function. It means any value assigned to an automatic variable within a function will be lost once the function is exited. The scope of an automatic variable can be smaller than an entire function. Automatic variables can be declared within a single compound statement.

In above example program you see three definitions for variable i. Here, you may be thinking if there could be more than one variable with the same name. Yes, there could be if these variables are defined in different blocks. So, there will be no error here and the program will compile and execute successfully. The printf in the inner most block will print 3 and the variable i defined in the inner most block gets destroyed as soon as control exits from the block. Now control comes to the second outer block and prints 2 then comes to the outer block and prints 1. Here, note that automatic variables must always be initialized properly, otherwise you are likely to get unexpected results because automatic variables are not given any initial value by the compiler.

11.5. GLOBAL VARIABLES

If a variable is defined outside all functions, then it is called a global variable.

The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration.

Global variables: are variables which are declared above the main( ) function. These variables are accessible throughout the program. They can be accessed by all the functions in the program. Their default value is zero.

Example:

```
#include <stdio.h>
int x = 0;/*Variable x is a global variable. It can be accessed throughout
the program */void increment(void) {
x = x + 1;
printf("\n value of x: %d", x);}
int main(){printf("\n value of x: %d", x);
increment();
return 0;
}
```

11.6. STATIC VARIABLES

Static variables are defined within individual functions and therefore have a same scope as automatic variables, i.e. they are local to the functions in which they are defined. Static variables retain their values throughout the program. Thus, if a function is exited and reentered later, the static variables defined within that function will retain their former values. Static variables are defined within a function in the same manner as automatic variables, but its declaration must begin with the static storage class designation. They cannot be accessed outside of their defining function. Initial values can be included in static variable declarations. The initial value must be expressed as constants, not expression, the initial values are assigned to their respective variables at the beginning of program, execution. The variables retain these values throughout the program, unless different values are assigned during the program. This is all for storage classes auto, extern and static.

Let us consider and example of static variables:

```
static int a;
```

If the keyword static is replaced with the keyword auto, the variable is declared to be of storage class auto. If a static local variable is assigned a value the first time the function is called, that value is still there when the function is called second time.

```
display_number()
{
static int number=2;
printf("number=%d\n", number); number++;
}
```

When the first time display_number is called, it prints the value 2, to which number is initialized. Then number is incremented to 3, and terminates. The second time display_number is called, it prints the value of 3. On the third call, the value printed is 4 and so on. Point to be noted here is that the initialization is not performed after the first call. An initialization used in a declaration occurs only once-when the variable is allocated. Since a static variable is allocated only once, the initialization occurs only during the entire program; no matter how many times the function is called. When the display_number function in the example is called the second time, the value

NOTES

found in the variable number is the value left there by the previous call to the function.

11.7. EXTERNAL VARIABLES

The extern specifier gives the declared variable external storage class. The principal use of extern is to specify that a variable is declared with *external linkage*elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A declaration declares the name and type of a variable or function. A definition causes storage to be allocated for the variable or the body of the function to be defined. The same variable or function may have many declarations, but there can be only one definition for that variable or function.

When extern specifier is used with a variable declaration then no storage is allocated to that variable and it is assumed that the variable has already been defined elsewhere in the program. When we use extern specifier the variable cannot be initialized because with extern specifier variable is declared, not defined.

External variables are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. External variable are recognized globally, that means they are recognized throughout the program, they can be accessed from any function that falls within their scope. They retain their assigned values within their scope. Therefore, an external variable can be assigned a value within one function and this value can be used within another function. With the use of external variables one can transfer the information between functions.

External variable definitions and external variable declarations are not the same thing. An external variable definition is written in the same manner as an ordinary variable declaration. The storage-class specifier extern is not required in an external variable definition, because these variables will be identified by the location of their definition within the program. An external variable declaration must begin with the storage class specifier extern. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside of the function. The declaration of external variables cannot include the assignment of initial values. External variables can be assigned initial values as a part of the variable definitions, but the initial values must be expressed as constants rather than as expression. These initial values will be assigned only once, at the beginning of the program. If an initial value is not included in the definition of an external variable, the variable will automatically be assigned a value of zero.

11.8. LET US SUM UP

In this unit, you have learnt about the storage classes and visibility of C language. This knowledge would make you understand the types of storage classes and its visibility. Thus, the storage class unit would have

brought you to closer to know the concept of storage class and its visibility of C language.

11.9. UNIT – END QUESTIONS

1. What is the use of storage class?
2. Explain the following.
   i) Static Variable ii) Automatic variable
3. Explain the following.
   i) External Variable ii) Global variable

11.10. ANSWER TO CHECK YOUR PROGRESS

1. Storage class in C decides the part of storage to allocate memory for a variable; it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable. There are four storage classes in C those are automatic, register, static, and external.
   .
2. Static variables are defined within individual functions and therefore have a same scope as automatic variables, i.e. they are local to the functions in which they are defined. Static variables retain their values throughout the program. Thus, if a function is exited and reentered later, the static variables defined within that function will retain their former values. Static variables are defined within a function in the same manner as automatic variables, but its declaration must begin with the static storage class designation.
   A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

3. The extern specifier gives the declared variable external storage class. The principal use of extern is to specify that a variable is declared with *external linkage* elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A declaration declares the name and type of a variable or function. A definition causes storage to be allocated for the variable or the body of the function to be defined. The same variable or function may have many declarations, but there can be only one definition for that variable or function.
   If a variable is defined outside all functions, then it is called a global variable.

The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration. Global variables: are variables which are declared above the main( ) function. These variables are accessible throughout the program. They can be accessed by all the functions in the program. Their default value is zero.

11.11. SUGGESTED READINGS

1. "C Primer Plus", Stephen Prata, Addison-Wesley Professional, Sixth Edition, 2013.
2. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
3. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.
4. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.
5. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.
6. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.

BLOCK IV: STORAGE CLASSES & FILE MANAGEMENT

UNIT XII – THE PREPROCESSOR

Structure

12.1. INTRODUCTION

We have already seen many features provided by C language. Yet another unique feature of the C language is the preprocessor. The C preprocessor provides several tools that are not available in other high–level languages. The programmer can use these tools to make his program more efficient in all respect.

12.2. OBJECTIVES

After going through this lesson you will be able to

- Explain preprocessor working
- Explain the # define Directive
- Define constants
- Explain macros
- Write the various directions such as # undef, #include, #fdef, #ifdef, #ifndef, # else, #if

12.3. FILE INCLUSION

When you issue the command to compile a C program, the program is run automatically through the preprocessor. The preprocessor is a program that modifies the C source program according to directives supplied in the program. An original source program usually is stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. Some compilers enable the programmer to run only the preprocessor on the source program and to view the results of the

preprocessor stage. All preprocessor directives begin with the number or sharp sign (#). They must start in the first column, and no space is required between the number sign and the directive. The directive is terminated not by a semicolon, but by the end of the line on which it appears. The C preprocessor is a collection of special statements called directives that are executed at the beginning of the compilation process. The #include and # define statements are preprocessor directives. The job of C preprocessor is to process the source code before it is passed to the compiler. The Pre-processor accepts source code as input and is responsible for

- Removing comments.
- Interpreting special pre-processor directives denoted by #.

#include causes the contents of another file to be compiled as if they actually appeared in place of the #include directive. The way this substitution is performed is simple, the Preprocessor removes the directive and substitutes the contents of the named file. Compiler allows two different types of #include's, first is standard library header include, syntax of which is as follows,

#include <filename.h>

Here, filename.h in angle brackets causes the compiler to search for the file in a series of standard locations specific to implementation. For example, gcc compiler on Linux, searches for standard library files in a directory called /usr/include.

Other type of #include compiler supports is called local include, whose syntax is as follows,

#include "filename.h"

filename in double quotes "" causes compiler to search for the file first in the current directory and if it's not there it's searched in the standard locations as usual. Nevertheless, we can write all our #include in double quotes but this would waste compiler's some time while trying to locate a standard library include file. Though, this doesn't affect runtime efficiency of program, however, it slows down compilation process.

For Example,

#include <stdio.h>

Notice here that stdio.h is a standard library file which compiler, first, should required to search in the current directory before locating it in standard location. A better reason why library header files should be used with angle brackets is the information that it gives the reader. The angle brackets make it obvious that

#include <string.h>

references a library file. With the alternate form

#include "string.h"

12.4. MACRO DEFINITION AND SUBSTITUTION

In Macro Substitution an identifier in a program is replaced by a predefined string composed of one or more tokens. We can use the #define directive for this purpose. The definition should start with the keyword **#define** and should follow by identifier and a token with at least one blank space between them. The token may be any text and identifier must be a valid C name. The pre-processor replaces

every occurrence of the **identifier** in the source code by **token**. There are different forms of macro substitution. The most common form is:

- Simple macro substitution.
- Argument macro substitution.

Simple token replacement is commonly used to define constants.

The #define directive is used to define a symbol to the preprocessor and assign it a value. The symbol is meaningful to the preprocessor only in the lines of code following the definition. For example, if the directive #define NULL 0 is included in the program, then in all lines following the definition, the symbol NULL is replaced by the symbol. If the symbol NULL is written in the program before the definition is encountered, however, it is not replaced. The # define directive is followed by one or more spaces or tabs and the symbol to be defined. The syntax of a preprocessor symbol is the same as that for a C variable or function name. It cannot be a C keyword or a variable name used in the program; if it is so a syntax error is detected by the compiler. For example, suppose a program contains the directive

#define dumb 52

Which in turn is followed by the declaration

int dumb;

This would be translated by the preprocessor into

int 52;

It merely substitutes symbols where it finds them. The symbol being defined is followed by one or more spaces or tabs and a value for the symbol. The value can be omitted, in which case the symbol is defined but not given a value. If this symbol is used later in the program, it is deleted without being replaced with anything. If a # define directive does not fit on a single line, it can be continued on subsequent lines. All lines of the directives except the last must end with a backslash(\ ) character. A directive can be split only at a point where a space is legal.

Syntax part	Explanation
#define	It is preprocessor directive used to define constant
macro_identifier	It is constant used in the program which we wish to declare using #define
value	It is value of the constant

Table: - Macro definition

#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.

The # undef Directive:-

If a preprocessor symbol has already been defined, it must be undefined before being redefined. This is accomplished by the #undef directive, which specifies the name of the symbol to be undefined. It is not necessary to perform the readefinition at the beginning of a program. A symbol can be redefined in the middle of a program so that it has one value

NOTES

in the first part and another value at the end. A symbol need not be redefined after it is undefined.

12.5. MACROS WITH ARGUMENTS

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept "arguments". Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition. A macro that accepts arguments is called a "function-like macro" because the syntax for using it looks like a function call.

To define a macro that uses arguments, you write a `#define' directive with a list of "argument names" in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

#define min(X, Y) ((X) < (Y) ? (X) : (Y))

To use a macro that expects arguments, you write the name of the macro followed by a list of "actual arguments" in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro `min' include `min (1, 2)' and `min (x + 28, *p)'.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro `min' defined above, `min (1, 2)' expands into

((1) < (2) ? (1) : (2))

where `1' has been substituted for `X' and `2' for `Y'.

Likewise, `min (x + 28, *p)' expands into

((x + 28) < (*p) ? (x + 28) : (*p))

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

macro (array[x = y, x + 1])

passes two arguments to `macro': `array[x = y' and `x + 1]'. If you want to supply `array[x = y, x + 1]' as an argument, you must write it as `array[(x = y, x + 1)]', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, `min (min (a, b), c)' expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
? (((a) < (b) ? (a) : (b)))
: (c))
```

If a macro `foo' takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: `foo ( )'. Just `foo ()' is providing no arguments, which is an error if `foo' expects an argument. But `foo0 ()' is the correct way to call a macro defined to take zero arguments, like this:

#define foo0() ...

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function.

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named `min' in the same source file that defines the macro.

If you write `&min' with no argument list, you refer to the function. If you write `min (x, bb)', with an argument list, the macro is expanded. If you write `(min) (a, bb)', where the name `min' is not followed by an open-parenthesis, the macro is not expanded, so you wind
up with a call to the function `min'. You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

#define FOO(x) - 1 / (x)

or this:

#define BAR (x) - 1 / (x)

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

12.6. NESTING OF MACROS

A macro body may also contain further macro definitions. However, these nested macro definitions aren't valid until the enclosing macro has been expanded! That means, the enclosing macro must have been called, before the nested macros can be called.

Example:

A macro, which can be used to define macros with arbitrary names, may look as follows:

```
DEFINE MACRO MACNAME
MACNAME MACRO
DB 'I am the macro &MACNAME.'
ENDM
ENDM
```

In order not to overload the example with "knowhow", the nested macro only introduces itself kindly with a suitable character string in ROM. The call

```
DEFINE Obiwan
```

would define the macro

```
Obiwan MACRO
DB 'I am the macro Obiwan.'
ENDM
```

and the call

```
DEFINE Skywalker
```

would define the following macro:

```
Skywalker MACRO
DB 'I am the macro Skywalker.'
ENDM
```

12.7. CONDITIONAL COMPILATION

Removing statements by hand would be quite tedious and could also lead to error. For this reason, the preprocessor provides directives for selectively removing section of code. This process is known as conditional compilation.

#define RECORD–FILE

If the # ifdef directive tests whether a particular symbol has been defined before the #ifdef is encountered. It does not matter what value has been assigned to the symbol. In fact, a symbol can be defined to the preprocessor without a value.

If the #ifdef directive is encountered after this definition it produce a true result. If, however, the directive #undef RECORD–FILE is encountered before the directive #ifdef RECORD–FILE then the preprocessor considers the symbol RECORD–FILE to be undefined and the #ifdef directive returns a false value.

If an #ifdef returns a true value, all the lines between the #ifdef and the corresponding #endif directive are left in the program. If those lines contain preprocessor directives, the directives are processed. In this way, conditional compilation directives can be nested. If the #ifdef evaluates as false, the associated lines are ignored, including any preprocessor directives that are included.

The statements need not all be grouped in one place. The #ifdef and #fendif directives can be used as many times as required. The preprocessor also provides the directive #ifndef, which produces a true result if a symbol is not defined. This makes it possible to use a single symbol to switch between two versions.

Conditional compilation can be used to select preprocessor directives as well as C code. For example suppose a header file is included in a program. and a certain preprocessor symbol (say, FLAG) may or may not be

defined in that header file. If the programmer wants FLAG never to be defined, then the #include directive can be followed by

#ifdef FLAG
#undef FLAG
#endif

This ensures that, even if the symbol is defined in the header file, its definition is removed. It is not sufficient merely to work on

#undef FLAG

because if FLAG is not defined, the directive is erroneous. If FLAG should always be defined, then we would write

#ifndef FLAG
#define FLAG
#endif

We could, of course, give FLAG a value. We cannot simply write

#define FLAG

Since if FLAG is already defined, an error probably will result.

THE #ELSE DIRECTIVE

This directive functions in much the same way as the else clause of an if statement. All lines between an #ifdef or an #ifndef directive and the corresponding # else clause are included if the #ifdef or #ifndef is true. Otherwise, the lines between the #else and the corresponding #endif are included.

The # else directive and the lines that follow it can be omitted, but never the #endif directive. No other text can be included on the same line as the #else or #endif directive.

THE # IF DIRECTIVE

The #if directive tests an expression. This expression can be of any form used in a C program, with virtually any operators, except that it can include only integer constant values No variables, or function calls are permitted, nor are floating point, character or string constants. The #if directive is true, if the expression evaluates to true (non–zero). Any undefined preprocessor symbol used in the #if expression is treated as if it has the value φ. Using a symbol that is defined with no value does not work with all preprocessors, and an attempt to do so might result in an error.

12.8. LET US SUM UP

In this unit, you have learnt about the preprocessor directive of C language and macros of C language. This knowledge would make you understand the file inclusion of preprocessor, macro definition, and nesting of macros. Thus, the preprocessor unit would have brought you to closer to know the concept of preprocessor directive and macros used in program development.

12.9. UNIT – END QUESTIONS

1. Identify the use of preprocessor directive with example.
2. Discuss about the macros and its implementation

12.10. ANSWER TO CHECK YOUR PROGRESS

1. The preprocessor is a program that modifies the C source program according to directives supplied in the program. An original source program usually is stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. Some compilers enable the programmer to run only the preprocessor on the source program and to view the results of the preprocessor stage. All preprocessor directives begin with the number or sharp sign (#). They must start in the first column, and no space is required between the number sign and the directive. The directive is terminated not by a semicolon, but by the end of the line on which it appears. The C preprocessor is a collection of special statements called directives that are executed at the beginning of the compilation process. The #include and # define statements are preprocessor directives..
2. In Macro Substitution an identifier in a program is replaced by a pre defined string composed of one or more tokens. We can use the #define directive for this purpose. The definition should start with the keyword **#define** and should follow by identifier and a token with at least one blank space between them. The token may be any text and identifier must be a valid C name. The pre-processor replaces every occurrence of the **identifier** in the source code by **token**. There are different forms of macro substitution.

12.11. SUGGESTED READINGS

1. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.
2. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.
3. "The C Programming Language", Brain W. Kernighan / Dennis Ritchie, Pearson Publications, 2015.
4. "C: The Complete Reference", Herbert Schildt, McGraw Hill Publications, Fourth Edition, 2017.
5. "C Primer Plus", Stephen Prata, Addison-Wesley Professional, Sixth Edition, 2013.
6. "Sams Teach Yourself C Programming in One Hour a Day", Bradley L. Jones, Sams Publishing, Seventh Edition, 2013.
7. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.
8. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.
9. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.
10. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.
11. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
12. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.

UNIT XIII – DYNAMIC MEMORY ALLOCATION AND LINKED LIST

Structure

13.1. INTRODUCTION

Dynamic memory allocation is an aspect of allocating and freeing memory according to your needs. Dynamic memory is managed and served with pointers that point to the newly allocated space of memory in an area which we call the heap**.** Now you can create and destroy an array of elements at runtime without any problems. A linked list is a collection of objects linked together by references from an object to another object. By convention these objects are names as nodes. So the basic linked list, or commonly called singly linked list consists of nodes where each node contains one or more data fields AND a reference to the next node. The last node contains a null reference to indicate the end of the list.

13.2. OBJECTIVES

After going through this lesson you will be able to

- Explain of dynamic memory allocation.
- Describe access to malloc(), calloc() & realloc().
- Define parameters of linked list.
- Explain singly linked list.

13.3. DYNAMIC MEMORY ALLOCATION

Using array in programming, we allocate a fixed size for our data. This size can't be increased or decreased while execution of the

program. We can't change it even if the size allocated is more or less than our requirement. This type of allocation of memory is called **Static Memory Allocation**. This leads to wastage or shortage of memory.

Dynamic memory allocation is an aspect of allocating and freeing memory according to your needs. Dynamic memory is managed and served with pointers that point to the newly allocated space of memory in an area which we call the **heap**. Now you can create and destroy an array of elements at runtime without any problems.

C allows programmer to allocate memory dynamically i.e. during run time and this process is called dynamic memory allocation. By allocating memory dynamically, we can use only the amount of memory required for us.

For this, C has four built in functions under **"stdlib.h"** header files for allocating memory dynamically. They are:

- malloc()
- calloc()
- realloc()
- free()

malloc() : - Allocates the memory of requested size and returns the pointer to the first byte of allocated space.

calloc() : - Allocates the space for elements of an array. Initializes the elements to zero and returns a pointer to the memory.

realloc() : - It is used to modify the size of previously allocated memory space.

free() : - Frees or empties the previously allocated memory space.

13.3.1. Allocating Memory with malloc

The malloc() function stands for memory allocation. It is a function which is used to allocate a block of memory dynamically. It reserves memory space of specified size and returns the null pointer pointing to the memory location. The pointer returned is usually of type void. It means that we can assign malloc function to any pointer.

Syntax

ptr = (cast_type*) malloc (byte_size);

Here,

- ptr is a pointer of cast_type.
- The malloc function returns a pointer to the allocated memory of byte_size.

For example,

Example: ptr = (int*) malloc (50);

When this statement is successfully executed, a memory space of 50 bytes is reserved. The address of the first byte of reserved space is assigned to the pointer ptr of type int.

The malloc function allocates a memory block of size n bytes (size_t is equivalent to an unsigned integer) The malloc function returns a pointer (void*) to the block of memory. That void* pointer can be used for any pointer type. malloc allocates a contiguous block of memory. If enough

contiguous memory is not available, then malloc returns NULL. Therefore always check to make sure memory allocation was successful by using

```
void* p;
if ((p=malloc(n)) == NULL)
return 1;
else
{ /* memory is allocated */}
```

13.3.2. Allocating Memory with calloc

The calloc function stands for contiguous allocation. This function is used to allocate multiple blocks of memory. It is a dynamic memory allocation function which is used to allocate the memory to complex data structures such as arrays and structures.

Malloc function is used to allocate a single block of memory space while the calloc function is used to allocate multiple blocks of memory space. Each block allocated by the calloc function is of the same size.

Syntax,

Ptr = (cast_type*) calloc (n, size)

- The above statement is used to allocate n memory blocks of the same size.
- After the memory space is allocated, then all the bytes are initialized to zero.
- The pointer which is currently at the first byte of the allocated memory space is returned.

Whenever there is an error allocating memory space such as the shortage of memory, then a null pointer is returned.

For example,

void *calloc(size_t nmemb, size_t size);

allocates memory for an array of nmemb elements each of size and returns a pointer to the allocated memory. Unlike malloc the memory is automatically set to zero.

calloc(n, sizeof(int))

is equivalent to

malloc(n*sizeof(int))

except for the fact that calloc block is already initialized. Calloc is appropriate when allocating a dynamic array of ints.

13.3.3. Freeing Memory

The memory for variables is automatically deallocated at compile time. In dynamic memory allocation, you have to deallocate memory explicitly. If not done, you may encounter out of memory error.

The free() function is called to release/deallocate memory. Will cause the program to give back the block to the heap (or free memory). The argument to free is any address that was returned by a prior call to malloc. If free is applied to a location that has been freed before, a double free memory error may occur. We note that malloc returns a block of void* and therefore can be assigned to any type.

```
double* A = (double*)malloc(n);
int* B = (int*)malloc(n);
char* C = (char*)malloc(n);
```

In each case however, the addresses A+i, B+i, C+i are calculated differently.

- A + i is calculated by adding 8i bytes to the address of A (assuming sizeof(double) = 8)
- B + i is calculated by adding 4i bytes to the address of B
- C + i is calculated by adding i bytes to the address of C

calloc vs. malloc: Key Differences

The calloc function is generally more suitable and efficient than that of the malloc function. While both the functions are used to allocate memory space, calloc can allocate multiple blocks at a single time. You don't have to request for a memory block every time. The calloc function is used in complex data structures which require larger memory space.

The memory block allocated by a calloc function is always initialized to zero while in malloc it always contains a garbage value.

13.3.4. Reallocating Memory Blocks

Using the **realloc()** function, you can add more memory size to already allocated memory. It expands the current block while leaving the original content as it is. realloc stands for reallocation of memory. realloc can also be used to reduce the size of the previously allocated memory.

Syntax

ptr = realloc (ptr, newsize)

The above statement allocates a new memory space with a specified size in the variable newsize. After executing the function, the pointer will be returned to the first byte of the memory block. The new size can be larger or smaller than the previous memory. We cannot be sure that if the newly allocated block will point to the same location as that of the previous memory block. This function will copy all the previous data in the new region. It makes sure that data will remain safe.

realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

13.4. POINTER SAFETY

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

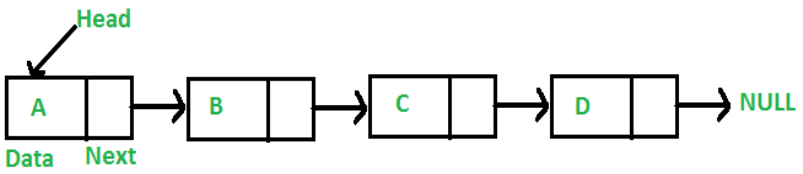

Figure 17: - Linked List

Static arrays are structures whose size is fixed at compile time and therefore cannot be extended or reduced to fit the data set. A dynamic array can be extended by doubling the size but there is overhead associated with the operation of copying old data and freeing the memory associated with the old data structure. One potential problem of using arrays for storing data is that arrays require a contiguous block of memory which may not be available, if the requested contiguous block is too large. However the advantages of using arrays are that each element in the array can be accessed very efficiently using an index. However, for applications that can be better managed without using contiguous memory we define a concept called "linked lists".

A linked list is a collection of objects linked together by references from one object to another object. By convention these objects are named as nodes. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node. The last node points to a NULL reference to indicate the end of the list.

The entry point into a linked list is always the first or head of the list. It should be noted that head is NOT a separate node, but a reference to the first Node in the list. If the list is empty, then the head has the value NULL. Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be contiguous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle and will be discussed in the next lesson. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

13.5. THE CONCEPT OF LINKED LIST

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1) Dynamic size

2) Ease of insertion/deletion

Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. **2)** Extra memory space for a pointer is required with each element of the list. **3)** Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node.

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

```
struct Node {
   int data;
   struct Node* next;
};
```

Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node

13.5.1. Inserting a node by using Recursive Programs

There are three different possibilities for inserting a node into a linked list. These three possibilities are:

1. Insertion at the beginning of the list.
2. Insertion at the end of the list
3. Inserting a new node except the above-mentioned positions.

Insertion at the beginning of the list: -

Step-1 : Get the value for NEW node to be added to the list and its position.
Step-2 : Create a NEW, empty node by calling malloc(). If malloc() returns no error then go to
or else say "Memory shortage".
Step-3 : insert the data value inside the NEW node's data field.
Step-4 : Add this NEW node at the desired position (pointed by the "location") in the LIST.
Step-5 : Go to step-1 till you have more values to be added to the LIST.

In Link list We can Insert new Node at First position , at Last position or in middle of the list at given position. Let's assume that no. of node in Link list is N
So we need three different Functions to insert node at different position.
Functions:

void addFront()

Function adds node to the first position in the list. Takes O(1) constant time to set next of new node to head and head to new node.

Void addRear()

Function adds node to the last position in the list. Takes O(N) time because we need to locate last node to insert. And in link list we don't have reference to last node so we need to traverse list.

Void addMiddle()

Function adds node after given node. Takes O(1) time , because we have given reference to the node after which we have insert node. In this way we can use Singly Link list, when number of node is not know priori.

But in Singly Link list, we can traverse List in only one direction. If we want to insert node before given node(p) instead of after given node (as we did in addMiddle() function) then it takes O(N) time to locate one previous node of the node p because we need to traverse List up to previous node of p.

13.5.2. Sorting and Reversing a Linked List

Sorting a List is one of the common operations that are performed. Sorting an array of object can easily be done using sorting algorithms like bubble, insertion or quick sort. However, sorting a linked list is not that trivial since linked list nodes cannot be randomly accessed. Only useful way to sort a linked list is to remove a node from the old list and insert into a new list in order. However for large lists, this is not very practical as insertion sort requires $O(n^2)$ operations. An alternative method is to define a toArray method for the linked list class that allows the linked list to be converted to an array first and then apply a more efficient algorithm like quick sort to sort. Once sorted the array can be converted into a linked list again. This is still more efficient in theory as toArray, sorting using qsort, constructing a linked list from an array takes O(n), O(n log n) and O(n) operations respectively. Asymptotically this is still better than $O(n^2)$ performance of a slow sort.

One of the useful operations that can be performed on linked lists is to reverse a linked list. A naive algorithm for doing this is to remove the first node of the list, insert to the beginning of a new list, then remove the second node (which now has become first node) and insert to the beginning of the new list etc. If the list has n nodes, then after performing n of these operations, we would have reversed the list.

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved,.

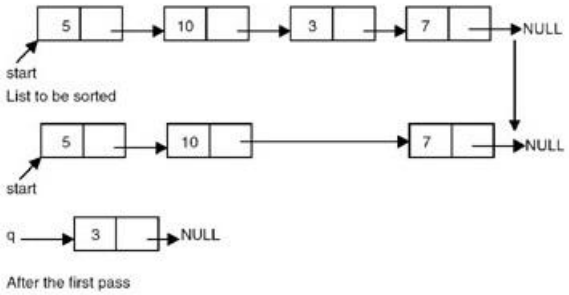

Figure 18: Sorting of a linked list.

To reverse a list, we maintain a pointer each to the previous and the next node, then we make the link field of the current node point to the

previous, make the previous equal to the current, and the current equal to the next,.

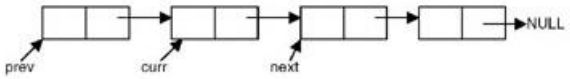

Figure 19: A linked list showing the previous, current, and next nodes at some point during reversal process.

Therefore, the code needed to reverse the list is

```
Prev = NULL;
While (curr != NULL)
{
  Next = curr->link;
  Curr -> link = prev;
  Prev = curr;
  Curr = next;
}
```

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
  int data;
  struct node *link;
};
struct node *insert(struct node *p, int n)
{
  struct node *temp;
  if(p==NULL)
  {
    p=(struct node *)malloc(sizeof(struct node));
    if(p==NULL)
    {
        printf("Error\n");
      exit(0);
    }
    p-> data = n;
    p-> link = NULL;
  }
  else
  {
    temp = p;
    while (temp-> link!= NULL)
    temp = temp-> link;
    temp-> link = (struct node *)malloc(sizeof(struct node));
    if(temp -> link == NULL)
    {
        printf("Error\n");
      exit(0);
    }
```

```
      temp = temp-> link;
      temp-> data = n;
      temp-> link = null;
     }
     return(p);
}

void printlist ( struct node *p )
{
     printf("The data values in the list are\n");
     while (p!= NULL)
     {
          printf("%d\t",p-> data);
       p = p-> link;
     }
}

/* a function to sort reverse list */
struct node *reverse(struct node *p)
{
  struct node *prev, *curr;
  prev = NULL;
  curr = p;
  while (curr != NULL)
  {
    p = p-> link;
    curr-> link = prev;
    prev = curr;
    curr = p;
  }
  return(prev);
}
/* a function to sort a list */
struct node *sortlist(struct node *p)
{
  struct node *temp1,*temp2,*min,*prev,*q;
  q = NULL;
  while(p != NULL)
  {
       prev = NULL;
    min = temp1 = p;
    temp2 = p -> link;
    while ( temp2 != NULL )
    {
          if(min -> data > temp2 -> data)
      {
                min = temp2;
                prev = temp1;
      }
      temp1 = temp2;
```

```
            temp2 = temp2-> link;
          }
          if(prev == NULL)
               p = min -> link;
          else
               prev -> link = min -> link;
          min -> link = NULL;
          if( q == NULL)
          q = min; /* moves the node with lowest data value in the list
        pointed to by p to the list
          pointed to by q as a first node*/
               else
               {
                 temp1 = q;
                 /* traverses the list pointed to by q to get pointer to its
        last node */
                 while( temp1 -> link != NULL)
                          temp1 = temp1 -> link;
                 temp1 -> link = min; /* moves the node with lowest data
value
        in the list pointed to
          by p to the list pointed to by q at the end of list pointed by
          q*/
            }
          }
          return (q);
        }
        void main()
        {    int n;
          int x;
          struct node *start = NULL ;
          printf("Enter the nodes to be created \n");
          scanf("%d",&n);
          while ( n- > 0 )
          {         printf( "Enter the data values to be placed in a
        node\n");
            scanf("%d",&x);
            start = insert ( start,x);
          }
          printf("The created list is\n");
          printlist ( start );
          start = sortlist(start);
          printf("The sorted list is\n");
          printlist ( start );
          start = reverse(start);
          printf("The reversed list is\n");
          printlist ( start );  }
```

Explanation

The working of the sorting function

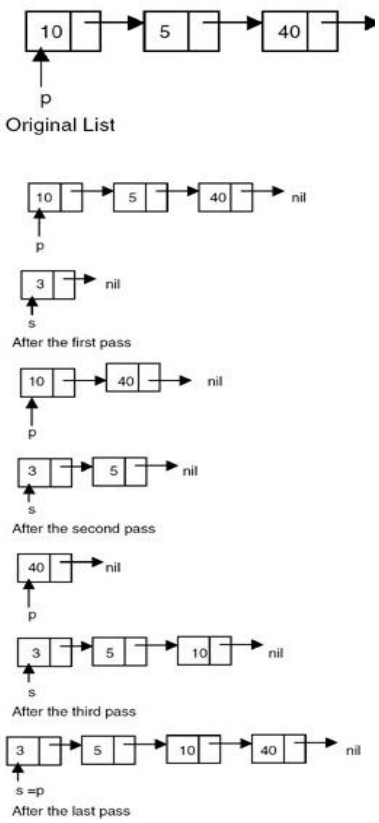

Figure 20: - Sorting of a linked list.

The working of a reverse function

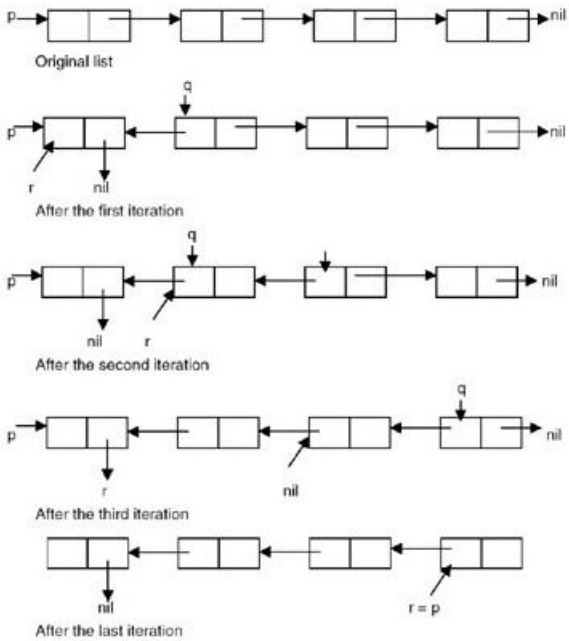

Figure 21: - Reversal of a list.

13.5.3. Deleting the Specified Node in a Singly Linked List

To delete a node from linked list, we need to do following steps.

1) Find previous node of the node to be deleted.

2) Change the next of previous node.
3) Free memory for the node to be deleted.

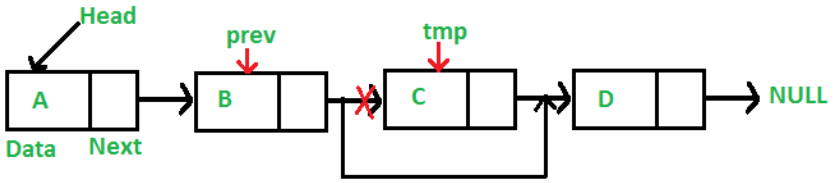

Figure 22: - Delete Node

Since every node of linked list is dynamically allocated using malloc() in C, we need to call free() for freeing memory allocated for the node to be deleted.

13.6. LET US SUM UP

In this unit, you have learnt about the concept of dynamic memory allocation and linked list of C language. This knowledge would make you understand the dynamic memory allocation and its types and to understand the concept of linked list and to know the implementation of linked list in C language. Thus, the dynamic memory allocation and linked list unit would have brought you to closer to know the concept of dynamic memory allocation procedures and use of linked list.

13.7. UNIT – END QUESTIONS

1. How to allocate the memory using the concept of dynamic memory allocation in C language? 2. Write about the functionality of linked list.

13.8. ANSWER TO CHECK YOUR PROGRESS

1. Dynamic memory allocation is an aspect of allocating and freeing memory according to your needs. Dynamic memory is managed and served with pointers that point to the newly allocated space of memory in an area which we call the heap. Now you can create and destroy an array of elements at runtime without any problems. C allows programmer to allocate memory dynamically i.e. during run time and this process is called dynamic memory allocation. For this, C has four built in functions under "stdlib.h" header files for allocating memory dynamically. They are:
   1. malloc() 2. calloc() 3. realloc() 4. free()

2. A linked list is a collection of objects linked together by references from one object to another object. By convention these objects are named as nodes. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node. The last node points to a NULL reference to indicate the end of the list.

To sort a linked list, first we traverse the list searching for the node with a minimum data value. Then we remove that node and append it to another list which is initially empty. We repeat this process with the remaining list until the list becomes empty, and at the end, we return a pointer to the beginning of the list to which all the nodes are moved

13.9. SUGGESTED READINGS

1. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.
2. "The C Programming Language", Brain W. Kernighan / Dennis Ritchie, Pearson Publications, 2015.
3. "C: The Complete Reference", Herbert Schildt, McGraw Hill Publications, Fourth Edition, 2017.
4. "C-How to Program", Paul Deitel & Harvey Deitel, Prentice-Hall Publications, Seventh Edition, 2013.
5. "Programming with ANSI and Turbo C", Ashok Kamthane, Pearson Education India, 2006.
6. "C Programming Absolute Beginners Guide", Greg Perry & Dean Miller, Que Publishing, Third Edition, 2013.

NOTES

UNIT XIV – FILE MANAGEMENT

Structure

14.1. INTRODUCTION

Many applications require that information be written to or read from an auxiliary storage device. Such information is stored on the storage device in the form of data file. Thus, data files allow us to store information permanently and to access later on and alter that information whenever necessary. In C, a large number of library functions is available for creating and processing data files. There are two different types of data files called stream-oriented (or standard) data files and system oriented data files.

14.2. OBJECTIVES

After going through this lesson you will be able to

- Open and close a data file.
- Create a data file.
- Process a data file.

14.3. DEFINING AND OPENING A FILE

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

We must open the file before we can write information to a file on a disk or read it. Opening a file establishes a link between the program and the operating system. The link between our program and the operating system is a structure called FILE which has been defined in the header file “stdio.h”. Therefore, it is always necessary to include this file when we do high level disk I/O. When we use a command to open a file, it will return a pointer to the structure FILE.

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

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values –

r Opens an existing text file for reading purpose.

w Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.

a Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.

r+ Opens a text file for both reading and writing.

w+ Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.

a+ Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Therefore, the following declaration will be there before opening the file, FILE *fp each file will have its own FILE structure. The FILE structure contains information about the file being used, such as its current size, its location in memory etc. Let us consider the following statements,

FILE *fp;

fp=fopen("Sample.C," "r");

fp is a pointer variables, which contains the address of the structure FILE which has been defined in the header file "stdio.h". fopen() will oepn a file "sample.c" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Here, "r" is a string and not a character.

When fopen() is used to open a file then, it searches on the disk the file to be opened. If the file is present, it loads the file from the disk into memory. But if the file is absent, fopen() returns a NULL. It also sets up a character pointer which points to the first character of the chunk of memory where the file has been loaded.

Reading a file:

To read the file's contents from memory there exists a function called fgetc(). This is used as:

s=fgetc(fp);

fgetc() reads the character from current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable s. This fgetc() is used within an indefinite while loop, for end of file. End of file is signified by a special character, whose ascii value is 26.

While reading from the file, when fgetc() encounters this Ascii special character, instead of returning the characters that it has read, it returns the macro EOF. The EOF macro has been defined in the file "stdio.h".

14.4. CLOSING A FILE

When we finished reading from the file, there is need to close it. This is done using the function fclose() through the following statement:

fclose(fp);

NOTES

NOTES

This command deactivates the file and hence it can no longer be accessed using getc(). 'C' provides many different file opening modes which are as follows:

1. "r" Open the file for reading only.
2. "w" Open the file for writing only.
3. "a" Open the file for appending (or adding) data to it.
4. "r+" The existing file is opened to the beginning for both reading and writing.
5. "w+" Same as "w" except both for reading and writing.
6. "a+" Same as "a" except both for reading and writing.

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

The fclose function takes a file pointer as an argument. The file associated with the file pointer is then closed with the help of fclose function. It returns 0 if close was successful and EOF (end of file) if there is an error has occurred while file closing.

14.5. INPUT / OUTPUT OPERATIONS ON FILE

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream –

int fputc(int c, FILE *fp);

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file.

int fputs(const char *s, FILE *fp);

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

int fgetc(FILE * fp);

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

char *fgets(char *buf, int n, FILE *fp);

Let's see a little more in detail about what happened here. First, **fscanf()** read just This because after that, it encountered a space,

second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

fprintf(file_pointer, str, variable_lists): It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.

fscanf(file_pointer, conversion_specifiers, variable_adresses): It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable_adresses using conversion specifiers.

```
#include <stdio.h>
int main() {
        FILE * file_pointer;
        char buffer[30], c;

        file_pointer = fopen("fprintf_test.txt", "r");
        printf("----read a line----\n");
        fgets(buffer, 50, file_pointer);
        printf("%s\n", buffer);

        printf("----read and parse data----\n");
        file_pointer = fopen("fprintf_test.txt", "r"); //reset the
pointer
        char str1[10], str2[2], str3[20], str4[2];
        fscanf(file_pointer, "%s %s %s %s", str1, str2, str3,
str4);
        printf("Read String1 |%s|\n", str1);
        printf("Read String2 |%s|\n", str2);
        printf("Read String3 |%s|\n", str3);
        printf("Read String4 |%s|\n", str4);

        printf("----read the entire file----\n");

        file_pointer = fopen("fprintf_test.txt", "r"); //reset the
pointer
        while ((c = getc(file_pointer)) != EOF) printf("%c", c);

        fclose(file_pointer);
        return 0;
    }
```

1. In the above program, we have opened the file called "fprintf_test.txt" which was previously written using fprintf() function, and it contains "Learning C with Guru99" string. We read it using the fgets() function which reads line by line where the buffer size must be enough to handle the entire line.
2. We reopen the file to reset the pointer file to point at the beginning of the file. Create various strings variables to handle each word separately. Print the variables to see their contents. The fscanf() is mainly used to extract and parse data from a file.

NOTES

3. Reopen the file to reset the pointer file to point at the beginning of the file. Read data and print it from the file character by character using getc() function until the EOF statement is encountered
4. After performing a reading operation file using different variants, we again closed the file using the fclose function.

14.6. PREDEFINED STREAMS

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

Besides the file pointers which we explicitly open by calling fopen, there are also three predefined streams. stdin is a constant file pointer corresponding to standard input, and stdout is a constant file pointer corresponding to standard output. Both of these can be used anywhere a file pointer is called for; for example, getchar() is the same as getc(stdin) and putchar(c) is the same as putc(c, stdout). The third predefined stream is stderr. Like stdout, stderr is typically connected to the screen by default. The difference is that stderr is not redirected when the standard output is redirected. For example, under Unix or MS-DOS, when you invoke

```
program > filename
```

anything printed to stdout is redirected to the file filename, but anything printed to stderr still goes to the screen. The intent behind stderr is that it is the ``standard error output''; error messages printed to it will not disappear into an output file. For example, a more realistic way to print an error message when a file can't be opened would be

```
if((ifp = fopen(filename, "r")) == NULL)
{
fprintf(stderr, "can't open file %s\n", filename);
exit or return
}
```

where filename is a string variable indicating the file name to be opened. Not only is the error message printed to stderr, but it is also more informative in that it mentions the name of the file that couldn't be opened.

14.7. ERROR HANDLING DURING I/O

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred

during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main () {
  FILE * pf;
  int errnum;
  pf = fopen ("unexist.txt", "rb");

  if (pf == NULL) {

    errnum = errno;
    fprintf(stderr, "Value of errno: %d\n", errno);
    perror("Error printed by perror");
    fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
  } else {

    fclose (pf);
  }

  return 0;
}
```

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1.

NOTES

14.8. RANDOM ACCESS TO FILES

Random access means you can move to any part of a file and read or write data from it without having to read through the entire file. Years ago, data was stored on large reels of computer tape. The only way to get to a point on the tape was by reading all the way through the tape. Then disks came along and now you can read any part of a file directly.

Individual records of a *random-access file* are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.

This makes random-access files appropriate for:

- airline reservation systems
- banking systems
- point-of-sale systems
- Other kinds of transaction-processing systems that require rapid access to specific data.

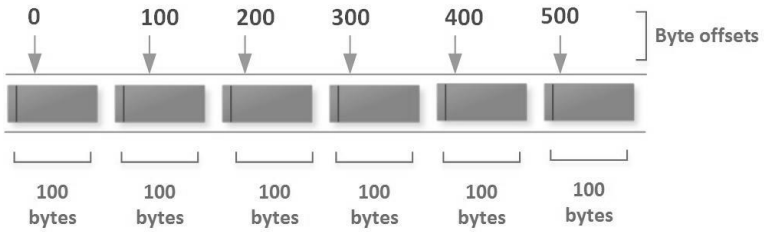

Figure 19: Random Access File Operation

Fixed-length records enable data to be inserted in a random-access file without destroying other data in the file. Data stored previously can also be updated or deleted without rewriting the entire file.

Creating a Random-Access File

Functions fwrite and fread are capable of reading and writing arrays of data to and from disk. The third argument of both fread and fwrite is the number of elements in the array that should be read from or written to disk.

There is no need to read each record sequentially, if we want to access a particular record.C supports these functions for random access file processing.

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

The rewind() function places the pointer to the beginning of the file, irrespective of where it is present right now. The syntax is

rewind(fp);

Where fp is the file pointer.

Pointer movement is of utmost importance since fread() always reads that record where the pointer is currently placed. Similarly , fwrite() always writes the record where the pointer is currently placed.

The fseek() function move the pointer from one record to another. The syntax is

fseek(fp,no-of-bytes, position);

Here, fp is the file pointer, no-of-bytes is the integer number that indicates how many bytes you want to move & position is the position of the

pointer from where you want to move e.g. it may be current position, beginning of file position, & End of file position.
e.g. to move the pointer to the previous record from its current position, the function is

fseek(fp, -recsize, SEEK-CUR);

Here no-of-bytes is stored in variable recsize which itself is a record size, SEEK_CUR is a macro defined in "stdio.h". Similarly SEEK_END, SEEK_SET can be used for end of file and beginning of file respectively If we wish to know where the pointer is positioned right now, we can use the function ftell(). It returns this position as a long int which is an offset from the beginning of the file. The value returned by ftell()can be used in subsequent calls to fseek(). The sample call to ftell() is show below:

p=ftell(fp);

14.9. COMMAND LINE ARGUMENTS

If any input value is passed through command prompt at the time of running of program is known as command line argument. It is a concept to passing the arguments to the main() function by using command prompt.

When Use Command Line Argument

When you need to developing an application for DOS operating system then in that case command line arguments are used. DOS operating system is a command interface operating system so by using command we execute the program. With the help of command line arguments we can create our own commands.

In Command line arguments application main() function will takes two arguments that is;

- argc
- argv

argc : argc is an integer type variable and it holds total number of arguments which is passed into main function. It take Number of arguments in the command line including program name.

argv[] : argv[] is a char* type variable, which holds actual arguments which is passed to main function.

Compile and run CMD programs

Command line arguments are not compile and run like normal C programs, these programs are compile and run on command prompt. To Compile and Link Command Line Program we need TCC Command.

First open command prompt.

Follow you directory where your code saved.

For compile -> C:/TC/BIN>TCC mycmd.c.

For run -> C:/TC/BIN>mycmd 10 20.

```
include <stdio.h>

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

  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
```

NOTES

```
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

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

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

Command line arguments related programs are not execute directly from TC IDE because arguments cannot be passed. To Edit the Command Line Argument Program uses edit Command.

- Whenever the program is compiled and link we will get .exe file and that .exe file itself is command.
- In above example program name is mycmd.c so executable file is mycmd.exe and command name is mycmd.
- To load the application in to the memory we required to use program name and command name.

Access data from outside of main

- argc and agrv are local variables to main function because those are the main function parameters.
- According to the storage classes of C argc and argv are auto variable to main function, so we can not extend the range of auto variable.
- By using argc and argv we can not access command from data outside of the main function.
- In implementation when we required to access command from data outside of the main function then use _argc, _argv variables.
- _argc and _argv are global variable which is declared in dos.h.

14.10. LET US SUM UP

In this unit, you have learnt about the concept of file operations and command line arguments of C language. This knowledge would make you understand the file, file operations, file management functions and command line arguments of C language. Thus, the file management unit would have brought you to closer to know the concept of various file operations and function of C language and also know the use of command line argument.

14.11. UNIT – END QUESTIONS

1. Define file? Explain about the file operations and functions of C language.
2. What is the use of command line argument.

NOTES

14.12. ANSWER TO CHECK YOUR PROGRESS

1. A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure. To read the file's contents from memory there exists a function called fgetc(). When we finished reading from the file, there is need to close it.
   Random access means you can move to any part of a file and read or write data from it without having to read through the entire file. Years ago, data was stored on large reels of computer tape. The only way to get to a point on the tape was by reading all the way through the tape. Then disks came along and now you can read any part of a file directly.
   Individual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.

2. If any input value is passed through command prompt at the time of running of program is known as command line argument. It is a concept to passing the arguments to the main() function by using command prompt. When you need to developing an application for DOS operating system then in that case command line arguments are used. DOS operating system is a command interface operating system so by using command we execute the program. With the help of command line arguments we can create our own commands. In Command line arguments application main() function will takes two arguments that is;
   argc : argc is an integer type variable and it holds total number of arguments which is passed into main function. It take Number of arguments in the command line including program name.
   argv[] : argv[] is a char* type variable, which holds actual arguments which is passed to main function.

14.13. SUGGESTED READINGS

1. "Programming in C", Stephen G. Kochan, Addison-Wesley Professional, Fourth Edition, 2014.
2. "Programming in ANSI C", E. Balagurusamy, McGraw Hill Publications, Eighth Edition, 2019.
3. "Let Us C", Yashavant Kanetkar, BPB Publications, Sixteenth Edition, 2017.
4. "Head First C: A Brain-Friendly Guide", David Griffiths & Dawn Griffiths, O'Reilly Publications, 2012.
5. "The C Programming Language", Brain W. Kernighan / Dennis Ritchie, Pearson Publications, 2015.
6. "C: The Complete Reference", Herbert Schildt, McGraw Hill Publications, Fourth Edition, 2017.

DISTANCE EDUCATION – CBCS – (2018-2019 Academic Year Onwards)

Question Paper Pattern (ESE) – Theory

Programme: - B.Com (CA)
Subject: - DBMS

Time: - 3 Hours
Maximum: - 75 Marks

Part – A (10 X 2 = 20 Marks)

1. What is entity relationship model?
2. Expand DDL and DML.
3. Write the use of magnetic disk.
4. What is SQL?
5. What are the command uses in the transaction control?
6. What is called EQUI join?
7. Define normalization.
8. What is called external shorting?
9. What is mean by recursive relationship?
10. Define aggregative operation.

Part – B (5 X 5 = 25 Marks)

11. a) Write about database languages?

(Or)

b) Write about storage manager.

12. a) Briefly explain triggers in SQL?

(Or)

b) Write about database architecture?

13. a) Briefly explain about distributed systems.

(Or)

b) What is Weak entity set? Explain with an example

14. a) Write about transaction concept.

(Or)

b) What is the role of the DBA? Discuss.

15. a) How to remove transitive dependence.

(Or)

b) Write about lock-based protocols.

Part – C (3 X 10 = 30 Marks)

16. Describe the structure of relational databases?
17. Discuss about triggers?
18. Discuss briefly about Entity-relationship model?
19. Explain Transaction isolation.
20. Explain briefly about Server system architectures?

NOTES