DIPLOMA IN COMPUTER APPLICATION

DCA-11-T

PROBLEM SOLVING THROUGH C

Centre for Distance and Online Education
Guru Jambheshwar University of Science & Technology, HISAR-125001

CONTENTS

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 1	
INTRODUCTION TO COMPUTER PROGRAMMING LANGUAGE	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

1.1 OBJECTIVES

The main Objective of this lesson is to introduce about the programming language. today computer programmer has many languages to choose from, but what's the difference between them?, what are these languages used for?, how can we categorize them in useful way?, why programming language is needed? , different types of programming language, how one language is translated into other, history of

high level languages, algorithm, and flowcharts.

1.2 INTRODUCTION

Computer science has made huge strides since it became recognized as a distinct academic discipline in the 1960s. One of the fundamental problems it has addressed is how to translate the problems that people need solved into a format that computers can process. A variety of powerful and ingenious solutions have been applied to this problem. Programming languages are human-engineered languages developed to convey instructions to machines; they are based on rules of syntax and semantics. Thousands of different programming languages have been developed, used, and discarded.

A Programming Language is an artificial language that can be used to create programs that controls the behavior of a machine, particularly a computer. A Programming language is designed to communicate instructions to a machine. A program written in some programming language involves some kind of computation for e.g. a program written to compute sum of two numbers or a program written to control hardware devices. Programming languages belong to a group of formal languages. They have been invented as intermediate between human and computer. Generally a programming language is different from natural languages that are only used for interaction between people, while programming language allows human to communicate with machines.

Why do not use natural languages as programming language?

Programming language are designed to prevent problem occurring with natural language.

a) Ambiguity- Natural languages are full of ambiguity and we need the context of a word in order to choose the appropriate meaning. For e.g. “minute” is used as a unit of time as a noun but means tiny as an adjective, only the context would differentiate the meaning.

b) Redundancy- Natural languages are full of redundancy helping to solve ambiguity problems and to minimize misunderstandings. For e.g. in the sentence “We are playing tennis at the moment”, “at the moment” is not really necessary but underlines that is happening now.

c) Literacy- Natural languages are full of idioms and phrases. The most popular in English is probably “It rains cats and dogs”. This can be very complicated even if u speak a foreign language very well.

Any programming language is usually described by two components -:

A) **Syntax-** The syntax of language defines the form of language. It is the set of rules that defines the combinations of symbol that is needed to make a correctly structured program in that particular language. In other words, syntax refers to the ways symbols may be combined to create well formed sentences or programs in the language. Syntax defines the formal relation between the constituent of language, thereby providing a structural description of the various expressions that make up legal strings in the language. Syntax deals solely with the form and structure of symbols in a language without any consideration given to their meaning. To perform same computation, different programming languages have their own syntax to write programs. Programs that are not written in proper syntax will show syntax error during compilation.

For e.g. syntax of for loop in C language is –

```
for ( initialization; end condition; increment/decrement)
{
    //Code for program body
}
```

B) **Semantics-** The semantics of language defines the meaning of language. Semantics describes the processes a computer follow when executing a program in that specific language. Semantics help to better understand what a program is doing or how to predict the outcome of program. In other words, semantics reveals the meaning of syntactically valid strings in a language. For natural language, this means correlating sentences and phrases with objects, thoughts and feelings of our experiences. For programming language semantics describes the behavior that a computer follows when executing a program in the language. We might disclose this behavior by describing the relationship between the input and output of a program or by step by step explanation of how a program will execute on a real or an abstract machine. In computer programming language semantics analysis is done after syntax analysis.

1.3 NEED FOR PROGRAMMING LANGUAGES

Generally a computing machine like a computer is designed to perform some specific tasks but in order to achieve this they need an accurate description of task. These tasks may ranges from calculating sum of two numbers to constructing the database of employees. A computer does not have capability to

design a strategy to solve problems. A computer will not perform all these tasks on its own unless it is instructed to do so. So to instruct on how to perform all these kind of tasks, human need to interact with machines too. Instructions are given to machines by human intervention with the help of programming languages. We can also say that programming languages is way to interact with machines. Thousands of programming languages are available to provide instructions to machines.

A programming language is a language designed to describe a set of actions to be executed on a computer. Programming language is thus a practical way for humans to give instruction to computers. The language used by processor is machine language. The code that reaches the processor consists of series of 0's and 1's known as binary data. Machine codes are difficult to understand, which is why intermediary languages have been developed. The code written in this type of language is transformed into machine code so that processor can process it. The "Assembler" was the first programming language ever used. This is very similar to machine code but can be understood by the programmer. But such a language is so similar to machine code that it strictly depends upon the type of processor used. Each processor type may have its own machine code. Thus a program developed for one machine may not be ported on another machine. The term portability describes the ability to use a software program on different tyoes of machine. A program written in assembler code may sometime have to be completely re written to work on another type of computer. A programming language has therefore several advantages:

- It is much understandable than machine code.
- It allows greater portability i.e. it can be easily adapted to run on different types of computer.

Many languages support more than one programming style, which brings out some of the basic design decision behind languages. Some of the basic design types are discussed below:

- **Procedural:** Procedural languages execute a sequence of statements that led a result. In essence, a procedural language expresses the procedure to be followed to solve the problem. Procedural languages typically use many variables and have heavy use of loops and other elements of state which distinguishes them from functional programming language. Procedural programming specifies a list of operations that the program must complete to get the desired output. Each program has a starting state, a list of operations to complete, and an ending point. This approach is also known as imperative programming.

A procedure is effectively a list of computations to be carried out. Procedural programming can be compared to unstructured programming, where all of the code resides in a single large block. By splitting the programmatic tasks into small pieces, procedural programming allows a section of code to be re-used in the program without making multiple copies. It also makes it easier for programmers to understand and maintain program structure. Two of the most popular procedural programming languages are FORTRAN and BASIC.

- **Functional:** in functional programming, programs are executed by evaluating expressions, in contrast with procedural programming where programs are composed of statement which change global state when executed. functional programming requires that functions are first class, which means that they are treated like any other values that can be passed as arguments to other functions or to be return as a result of a function. It is also possible to manipulate and define function from within other function.
- **Structured Programming:** Structured programming is a special type of procedural programming. It provides additional tools to manage the problems that larger programs were creating. Structured programming requires that programmers break program structure into small pieces of code that are easily understood. It also frowns upon the use of global variables and instead uses variables local to each subroutine. The most popular structured programming languages include C, Ada, and Pascal.
- **Object Oriented**: Object oriented programming views the world as a collection of objects that have internal data and external means of accessing parts of that data. The main goal of object oriented programming is to think about the problem by dividing it into collection of objects that provide services that can be used to solve a particular problem.

Object-oriented programming is one the newest and most powerful paradigms. In object oriented programs, the designer specifies both the data structures and the types of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object. A program thus becomes a collection of cooperating objects, rather than a list of instructions. Objects can store state information and interact with other objects.

One of the main features of object oriented programming is encapsulation that means that

everything an object will need should be inside the object. OOP is popular in larger software projects; because objects or groups of objects can be divided among teams and developed in parallel. Object-oriented programming seems to provide a more manageable foundation for larger software projects. The most popular object-oriented programming languages include Java, Visual Basic, C#, C++, and Python.

The evolution of programming languages was driven by a quest for efficient translation of human language to machine code. This produced languages with high levels of abstraction, which hide the hardware and used representations that are convenient and comfortable to human programmers. At some point, another critical aspect of language design complexity of programs.

As programs became larger and more sophisticated, developers realized that some language types were easier to support in large systems. This has lead to greater use of objected-oriented and event driven programming languages.

1.3 CLASSIFICATIONS OF PROGRAMMING LANGUAGE

A Programming language is a set of grammatical rules for instructing a computer to perform a specific task. Today's we have thousands of programming languages to chose from, but what's the difference between them? What are these languages used for? How can we categorize in useful way? Early programming languages were designed for specific kinds of tasks. In any case, each language has its own characteristics, vocabulary, and syntax. These days programming languages are becoming more and more general and all purpose but they still have their specialization and each language has its own advantage and disadvantage. Languages are generally being divided into few basic types as shown in Figure.1.3.1 below:-

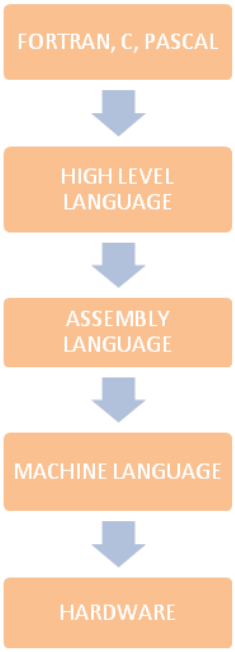


Figure.1.3.1: Types of Languages

Machine level language: It is instructive to try to communicate with a computer in its own language. Computers do not understand any of natural language such as English, French or German. The very lowest possible level at which you can program a computer is in its own native machine code, consisting of strings of 1's and 0's and stored as binary numbers. The main problems with using machine code directly are that it is very easy to make a mistake, and very hard to find it once you realize the mistake has been made. The only language understood by computer is *machine language* that is composed of only two symbols "0" and "1" or power "on" and "off". Each statement in a machine language program is a sequence of bits. Series of bits represents instruction that a computer can understand. For e.g. number 455 is represented by the bit sequence 111000111. Machine language is low level programming language. Each computer understands only its machine language. It is easily understand by computers but difficult to understand for humans. That is why people use high level language. Programs written in high level language are compiled and/or interpreted into machine language so that computers can execute them. They have a sort of dictionary containing all valid words of that language. These words are basic instructions for e.g. machine language code for add instruction is 0001 for some machine. It means that machine will interpret 0001 as to add two numbers. However it

is difficult for human being to interact machine with combinations of 0's and 1's for different instructions. Every CPU has its own machine language. It means the same instruction in one machine may have different meaning in some other machine i.e. it is machine dependent. Machine language code can't be run directly on any computer also machine language are tedious and time consuming. Thus, a program written in a low-level language can be extremely efficient, making optimum use of both computer memory and processing time. However, to write a low-level program takes a substantial amount of time, as well as a clear understanding of the inner workings of the processor itself. Therefore, low-level programming is typically used only for very small programs, or for segments of code that are highly critical and must run as efficiently as possible.

Assembly language: Now because human have difficulties to understand, analyze, and extract information from sequence of 0's and 1's, an language known as *assembly language* is written ,that maps instruction words to synonyms that give an idea of what the instruction does. We can also say that an assembly language contains the same instructions as a machine language or machine language is a representation of machine language, but instruction and variables have names instead of just combination of 0's and 1's. In other words, each assembly language instruction translates to a machine language instruction. No matter how close assembly language is to machine code, the computer still cannot understand it. Computer manufacturers started providing English-like words abbreviated as mnemonics that are similar to binary instruction in machine language. The advantage of assembly language is that its instructions are readable. The designer uses easy symbols that can be easily remembered by programmer, so that programmer can easily develop the program in assembly language. For instance 0001 code in machine language became ADD instruction in assembly language, other e.g. are SUB for subtraction, MUL for multiplication, DIV for division. Assembly language can be run directly run on computers processor. This type of language is most appropriate in writing operating system and desktop applications. Every processor has its own assembly language. Though assembly language statements are readable, statements are still low-level. Another disadvantage of assembly language is that it is not portable. In other words, assembly language programs are specific to a particular hardware. Assembly language program for a Mac will not work on a PC.

But this can be advantage for programmers who are targeting a specific platform and need full control over the hardware.

High level language: In order to write algorithms for solving more complex problems there was a need for machine independent *higher level language* with more understandable instruction set. The first ones high level language is C and FORTRAN. High level languages are what most programmers use. One advantage of high level language is that they are very readable. Statements in these languages are user friendly and somewhat English-like languages. High level language is easier to learn and write. You can gain a basic understanding of what a C program is doing by simply reading the program source code. Loops in C programs are indicated by the words for, while, and do. Another advantage of high level language is that they are less tedious to use. A single statement in a high level language can translate into many machine language statements. Also, high level languages are usually portable.

A disadvantage of high level language is that they are usually less powerful and less efficient. Since statements are high level, you cannot code at the bit level the way you can do with assembly language. High level languages also need to be compiled and/or interpreted into machine language before execution.

This is basic description of three basic types of programming languages. Until people can process information like computers, we will leave machine language to computers and use high-level programming languages instead.

1.4 LANGUAGE TRANSLATORS

Programming languages are foreign languages for computer. Therefore there is a need to translate a program of code of instructions given to computer into a language that a computer can understand i.e. machine language. A program can be written in assembly language as well as high level language. This written program is called as source program. This source program needs to be converted into machine language, which is called as object program. A translator translates source code of programming language into machine language instruction code. Generally, source program is written in languages such as, C, PASCAL, COBOL and ASSEMBLY LANGUAGE, which should be translated into machine language before execution. Programming language translators are classified as follows:

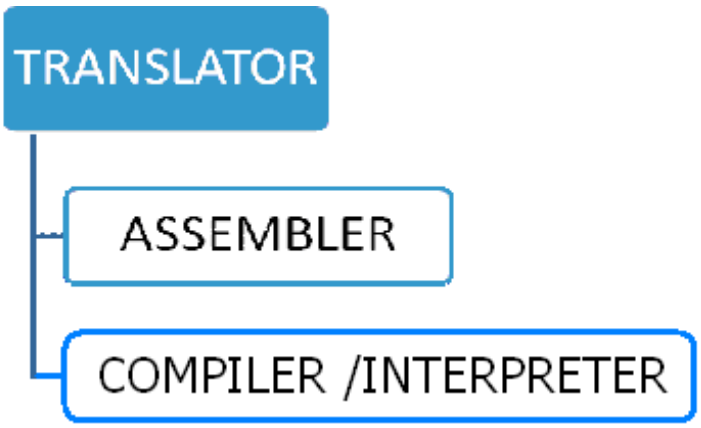


Figure 1.4.1 Translators

Assemblers are needed to transform symbolic codes (mnemonics) of assembly language into machine language instructions. Each assembly language mnemonic has its own machine language code in the form of 0's and 1's. This translation job is performed either manually or with the help of program called as assembler.In case of manual translation, the programmer uses the set of instructions supplied by the manufacturer. The hexadecimal code for mnemonic instruction is searched from code sheet. This procedure is somewhat tedious and time consuming. Alternate solution is by using assembler program which provides the codes of mnemonics. This process is fast and facilitates the programmer in developing the program fast.

Figure 1.4.2

With assembly-language programs, updating or correcting a compiled program requires that the original (source) program be modified appropriately and then recompiled to form a new machine-language (object) program in Figure 1.4.2.

Compilers are the translators program which translates high level language into machine codes, which can be used again and again. The source code is input to the compiler and the object code is output. The

entire source code will be read by the compiler first and then generates the object code. Some high level languages such as C, C++, and JAVA have compiler employed. The compiler displays the error list and warning for the statements violating the syntax of programming language. Compilers also have the ability of linking subroutines to the programs.

Figure 1.4.3

Compiler languages are the high-level equivalent of assembly language. Each instruction in the compiler language can correspond to many machine instructions. Once the program has been written, it is translated to the equivalent machine code by a program called a compiler as shown in Figure 1.4.3. Once the program has been compiled, the resulting machine code is saved separately, and can be run on its own at any time. Typically, the compiled machine code is less efficient than the code produced when using assembly language. This means that it runs a bit more slowly and uses a bit more memory than the equivalent assembled program. To offset this drawback, however, we also have the fact that it takes much less time to develop a compiler-language program, so it can be ready to go sooner than the assembly-language program.

Interpreter also comes in the group of translators. Interpreter works somewhat similar to compilers with few differences. The source program is English-like both for compiler and interpreter. Interpreter reads source program line by line whereas in compiler entire source program is read. Interpreter directly executes the program from its source code due to which every time source code is inputted to the interpreter, unlike in compiler where once the source code is translated into object code then it can be used again and again without inputting the source code again into compiler. In other words, in interpreter each line is converted into object code. It takes very less time for execution because no intermediate object code is generated.

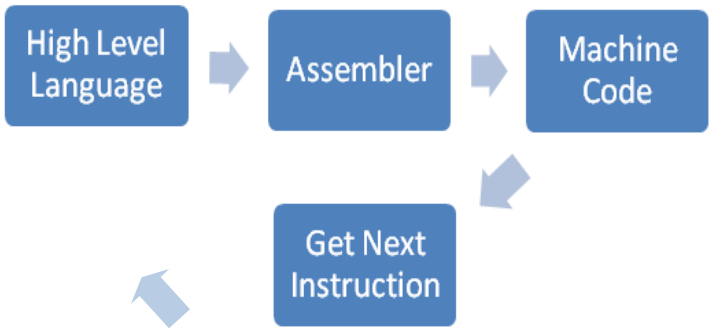


Figure 1.4.4

An interpreter language, like a compiler language, is considered to be high level. However, it operates in a totally different manner from a compiler language. Rather, the interpreter program

resides in memory, and directly executes the high-level program without preliminary translation to machine code. This use of an interpreter program to directly execute the user's program has both advantages and disadvantages. The primary advantage is that you can run the program to test its operation, make a few changes, and run it again directly. There is no need to recompile because no new machine code is ever produced. This can enormously speed up the development and testing process.

On the down side, this arrangement requires that both the interpreter and the user's program reside in memory at the same time. In addition, because the interpreter has to scan the user's program one line at a time and execute internal portions of itself in response; execution of an interpreted program is much slower than for a compiled program.

1.7 EVOLUTION OF HIGH LEVEL LANGUAGE

The lack of portability between different computers led to the development of high-level languages –so called because they permitted a programmer to ignore many low-level detail of the computer's hardware. Further, it was recognized that the closer the syntax, rules, and mnemonics of the programming language could be to "natural language" the less likely it became that the programmer would inadvertently introduce errors (called 'bugs") into the program. Hence, in the mid-1950s a third generation of languages came into use. These algorithmic or symbolic languages are designed for solving a particular type of problem. Unlike machine or symbolic languages, they vary little between computers. They must be translated into machine code by a program called a compiler or interpreter.

Early computers were used almost exclusively by scientists, and the first high-level language,

FORTRAN [Formula translation], was developed (1953-57) for scientific and engineering application by John Backus at the IBM Corp. A program that handled recursive algorithms better, LISP [LISt Processing], was developed by John McCarthy at the Massachusetts Institute of Technology in the early 1950s; implemented in 1959, it has become the standard language for the artificial intelligence community. COBOL [COmmon Business Oriented Language], the first languages intended for commercial application, is still widely used; it was developed by a committee of computer manufacturers and users under the leadership of Grace Hopper, a U.S. Navy programmer, in 1959. ALGOL [ALGorithmic Language], developed in Europe about 1958, is used primarily in mathematics and science, as is APL [A Programming Language], published in the United States in 1962 by Kenneth Iverson. PL/1 [Programming Language 1], developed in the late 1960s by the IBM Corp., and ADA [for Ada Augusta, countess of Lovelace, biographer of Charles Babbage], developed in 1981 by the U.S. Det. Of Defense, are designed for both business and scientific use.

BASIC [Beginner's All-purpose symbolic Instruction Code] was developed by two Dartmouth College professors, John Kemeny and Thomas Kurtz, as a teaching tool for undergraduates (1966); it subsequently became the primary language of the personal computer revolution. In 1971, Swiss professor Nicholas Wirth developed a more structured language for teaching that he named Pascal (for French mathematician Blaise Pascal, who built the first successful mechanical calculator). Modula 2, a Pascal like language for commercial and mathematical application, was introduced by Wirth in 1982.Ten years before that, to implement that the UNIX operating system, Dennis Ritchie of Bell Laboratories produced a language that he called C; along with its extension, called C++, developed by Bjarne Stroustrup of Bell laboratories, it has perhaps become the most widely used general-purpose language among professional programmers because of its ability of deal with used the rigors of object-oriented programming. Java is an object-oriented language similar to C++ but simplified to eliminate features that are prone to programming errors. Java was developed specifically as a network-oriented language, for writing programs that can be safely downloaded through the internet and immediately run without fear of computer viruses. Using small Java programs called, applets, World Wide Web pages can be developed that include a full range of multimedia functions.

Fourth- generation languages are nonprocedural –they specify what is to be accomplished without describing how. The first one, Forth, developed in 1970 by American astronomer Charles Moore, is

used in scientific and industrial control application. Most fourth-generation languages are written for specific purpose. Fifth-generation languages, which are still in their infancy, are an outgrowth of artificial intelligence research. PROLOG [PROgrammng LOGic], developed by French computer scientist Alain Colmerauer and logician Philippe Roussel in the early 1970s, is useful for programming logical processes and making deductions automatically. Many others languages have been designed to meet specialized needs. GPSS [General Purpose System Simulator] is used for modeling physical and environmental events, and SNOBOL [S

tring- Oriented Symbolic Language]is designed for pattern matching and list processing. LOGO, a version of LISP, was developed in the 1960s to help children learn about computers. PILOT

[Programmed Instruction Learning, Or Testing] is used in writing instructional software, and Occam is a non-sequential language that optimizes the execution of a program's instructions in parallel-processing system.

There are also procedural languages that operate solely within a larger program to customize it to a user's particular needs. These include the programming language of several database and statistical programs, the scripting languages of communications programs, and the macro languages of word-processing programs.

1.8 ALGORITHM

Since a computer program consists of a series of instructions for the computer to carry out, plus any necessary fixed data required to perform those instructions, the process of programming consists of defining those instructions and that data. Thus, in order to design a program for a computer, you must determine three basic elements:

- The instructions that must be performed.
- The order in which those instructions are to be performed.
- The fixed data required to perform the instructions.

To make computer to do specific task we have to write a program. To write a computer program, we have to tell the computer, step by step, exactly what programmer wants it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal. When you are telling the computer what to do, you also get to choose how it's going to do it. That's where computer

algorithms come in.

In computer science, an algorithm is a step-by-step procedure for performing a specific task. In programming, algorithms are the set of well defined instruction in sequence to solve a program.

Qualities of a good algorithm:

- Inputs and outputs should be defined precisely.
- Each step in algorithm should be clear and unambiguous.
- Algorithm should be most effective among many different ways to solve a problem.
- An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

How to write an algorithm?

1) Begin
2) Include variable and their usages.
3) Try to give go back to step number if condition fails.
4) Use jump statement to jump from one statement to another.
5) Define expression
6) Use break and terminate to stop the process.

Some of the points that should keep in mind while creating an algorithm:

- To write an algorithm, keep in mind that algorithm is step by step process.
- Depending upon programming language, include syntax wherever necessary.
- Make it efficient
- If there are any loops try to give sub number lists
- Try to avoid unwanted comments.
- Try to avoid unwanted raw data in algorithm.
- Use proper logic.
- Make in small and concise

- Make a clear plan before writing an algorithm
- Use fast calculating iterations

Example - Algorithm for adding two numbers.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

Example2: Algorithm to find out number is odd or even.

Step 1: Start

Step 2: Input number

Step 3: Rem=number mod2

Step 4: if rem=0 then

print "number even"

else

print "number odd"

endif

Step 5: Stop

Most computer programmers spend a large percentage of their time creating algorithms. (The rest of their time is spent debugging the algorithms that don't work properly. The goal is to create efficient algorithms that do not waste more computer resources (such as RAM and CPU time) than necessary. This can be difficult, because an algorithm that performs well on one set of data may perform poorly on other data.

1.9 FLOWCHART

Flowchart is a graphical or symbolic representation of algorithm. It is the diagrammatic representation of the step-by-step solution to a given problem. Flowcharts are very helpful in writing program and explaining program to others. Each step in the process is represented by a

different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction. A flowchart typically shows the flow of data in a process, detailing the operations/steps in a pictorial format which is easier to understand than reading it in a textual format.

A flowchart describes what operations (and in what sequence) are required to solve a given problem. A flowchart can be likened to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. Flowcharts are a pictorial or graphical representation of a process. The purpose of all flow charts is to communicate how a process works. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts often facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

For example, consider that we need to find the sum, average and product of 3 numbers given by the user.

Algorithm for the given problem is as follows:

Read X, Y, Z

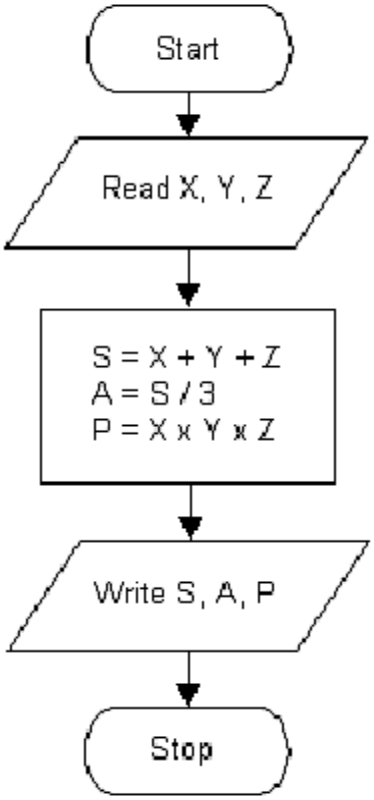
Compute Sum (S) as X + Y + Z

Compute Average (A) as S / 3

Compute Product (P) as X x Y x Z

Write (Display) the Sum, Average and Product

Flowchart for the above problem will look like



The *benefits of flowcharts* are as follows:

- *Communication*: Flowcharts are better way of communicating the logic of a system to all concerned.
- *Effective analysis*: With the help of flowchart, problem can be analysed in more effective way.
- *Proper documentation*: Program flowcharts serve as a good program documentation, which is needed for various purposes.
- *Efficient Coding*: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- *Proper Debugging*: The flowchart helps in debugging process.
- *Efficient Program Maintenance*: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

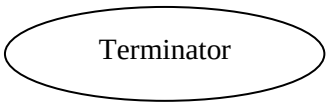
Although a flowchart is a very useful tool, there are a few limitations in using flowcharts which are listed below:

- *Complex logic*: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
- *Alterations and Modifications*: If alterations are required the flowchart may require re-drawing completely.
- *Reproduction*: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

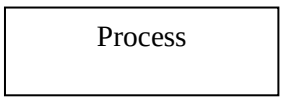
Flowchart symbol

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs are shown.

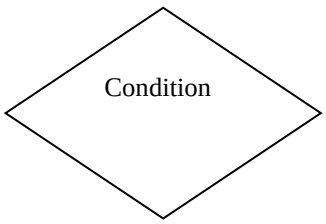
Terminator: An oval flow chart shape indicates the start or end of the process, usually containing the word “Start” or “End”.



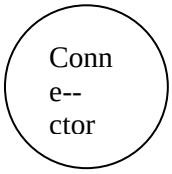
Processing steps: shows a process or action step. This is the most common symbol and is represented as rectangles.



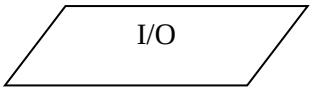
Conditional: (or decision), represented as a diamond (rhombus). These typically contain a Yes/No question or True/False test. This symbol is unique in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the "pre-defined process" symbol.



Connector: A small, labeled, circular flow chart shape used to indicate a jump in the process flow. Connectors are generally used in complex or multi-sheet diagrams



Data (I/O): the data flowchart shape indicates input to and output from a process. As such, the shape is more often referred to as I/O shape rather than data. Parallelogram shape is used.



Arrow: used to show the flow of control in a process. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.

These are the basic symbols used generally. Now, the basic guidelines for drawing a flowchart with the above symbols are that:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
- The flowchart should be neat, clear and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The flowchart is to be read left to right or top to bottom.
- A process symbol can have only one flow line coming out of it.
- For a decision symbol, only one flow line can enter it, but multiple lines can leave it to denote possible answers.
- The terminal symbols can only have one flow line in conjunction with them.

Consider an algorithm for calculating factorial of any number:

Step 1: Start

Step 2: Read the number n

Step 3: [initialize]

i=1, fact=1

Step 4: Repeat step 4 through 6 until i<=n

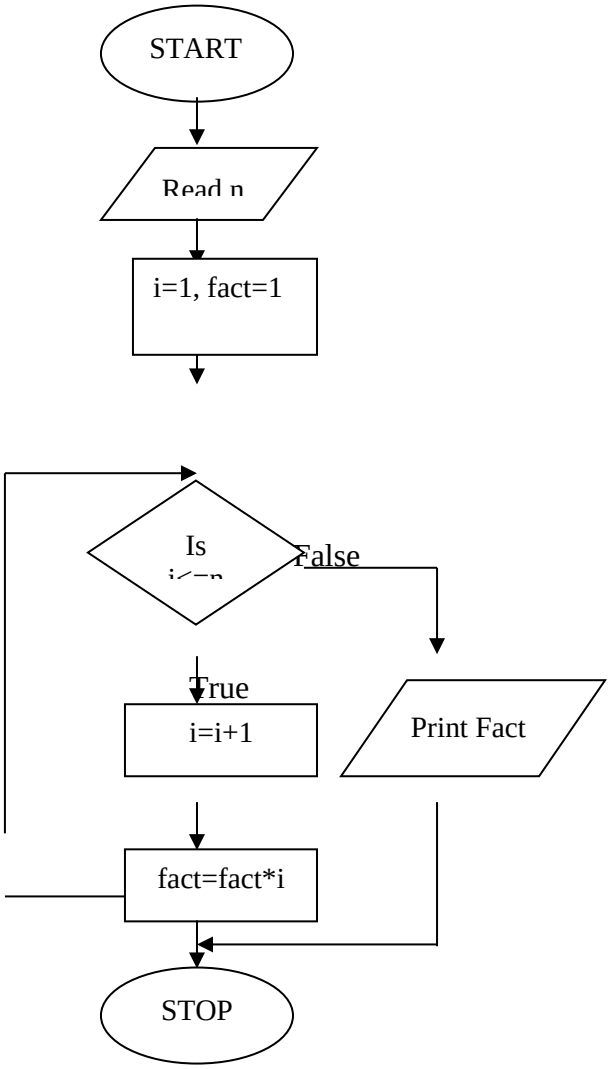
Step 5: fact=fact*i

Step 6: i=i+1

Step 7: Print fact

Step 8: Stop

Flowchart of above algorithm is:



1.8 SUMMARY

In this lesson we have studied:

- Need for programming languages to communicate with computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.
- Different type of programming languages: low level language, machine level language, high level language.
- Kinds of translators to translate between different kinds of languages so that computer can understand the user language.
- Evolution level language from outdated languages like FORTRAN, LISP to enhanced and user friendly languages JAVA and C++.
- Using Algorithm to build any program, Algorithm tells the computer how to do a particular task. Any program can be made very easily if user knows the basic working algorithm for that program.
- Flowchart is used to represent the working of algorithm or program by using various symbols.

1.9 KEYWORDS

Programming language- A programming Language is an artificial language that can be used to create programs that controls the behavior of a machine, particularly a computer.

Translator- A translator translates source code of programming languages into machine language instruction code.

Assembler- Assemblers are needed to transform symbolic codes (mnemonics) of assembly language into machine language instructions.

Compiler- Compiler are the translator program which translates high level language into machine codes.

Interpreter- Interpreter works somewhat similar to compilers with few differences. Interpreter reads source program line by line whereas in compiler entire source program is read.

Algorithm- Algorithms are the set of well defined instruction in sequence to solve a program

Flowchart- Flowchart is graphical view of algorithms.

1.10 REVIEW QUESTIONS

1. What do you mean by programming language?
2. What is necessity of programming language in today's world?
3. Explain different types of programming languages?
4. Define Translator. Explain different type of translator?
5. Define Algorithm. Write algorithm for following;

5.1 To find whether a number is even or not.

5.2 To calculate division of two numbers with condition that if denominator is zero then to display an error message.

5.3 To find average of marks obtained by student in four subjects and indicate whether it is pass or fail.

6. Define flowchart. Draw flowchart for following:

6.1 To read two sides of a rectangle and calculate its area and perimeter.

6.2 To read two value and find largest among two.

1.11 FURTHER READINGS

[1] E.Balagurusamy, "Programming in ANSI C", Tata McGraw Hill.

[2] T.D. Brown, "*C for Basic Programmers"*, Silicon Press.

[3] Brain W.Keringhan & Dennis M.Ritchie, "*The C Programming Language",* Prentice Hall.

[4] Peter Prinz, Tony Crawford, "*C in a Nutshell",* O' REILLY.

[5] V.Rajaraman, "*Computer Basics and C Programming",* PHI Learning.

[6] Anita Goel, "*Computer Fundamentals",* Pearson Education.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 2	
INTRODUCTION TO 'C'	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

2.1 OBJECTIVES

The main objective of this lesson is to make the students have introduction to the most popular and easy computer programming language C, why C is so popular, characteristics of C language, basic building blocks of C program, which words have specific meaning in C program, which part

is being ignored by while compiling the program and why, learn about the various types of data that can be operated using C program,

2.2 INTRODUCTION

C is a general purpose high level programming developed by Dennis Ritchie in 1972 at AT & T's Bell laboratories of USA for UNIX operating system (UNIX operating system and all applications of UNIX are written in C language only). It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972.

Features of C language

- C programs are easy to read, maintain, fast, efficient, effective, reusable, reliable, and modular.
- C has various applications like Operating System development, Database systems, word processors, Compilers and assemblers, spread sheets etc.
- C is a middle level language i.e. it combines the features of both low level language and high level language.
- C has a powerful feature of Portability i.e. C program can be compiled and executed in any operating system (Unix, DOS, Windows)
- It is also called as procedure oriented programming language.
- C has been coded in assembly language.
- C language allows manipulation of bits, bytes and addresses.
- C is a case sensitive language.

Various C standards

- **K&R C or Classic C- Developed by Brian Kernighan and Dennis Ritchie in 1978.**
- **C89/C90 standard** – First standardized specification for C language was developed by American National Standards Institute (ANSI) in 1989. C89 and C90 standards refer to the same programming language.

- **C99 standard** – Next revision was published in 1999 that introduced new futures like advanced data types and other changes.
- **C11 standard** - The new version adds some more features to C. It allows some portions of the existing C99 standard library optional, and improves compatibility with C++.
- **Embedded C** – This standard includes enhanced features that were not available in normal C like named address spaces, fixed-point arithmetic, and basic I/O hardware addressing.
- **Microsoft C, Quick C** – Released by Microsoft Corporation.
- **Turbo C**- Version by Borland International.
- **ANSI standard**- Version adapted by American National Standard.

In the further context we will talk about turbo C version. Here we will talk about C features that are included in the ANSI standard and almost all programs following ANSI standard can be run under turbo C version & where the converse is not true.

Advantages & Disadvantages of C

There are few advantages and disadvantages of C programming language these are:

Advantages:

1. **Easy to learn**- The main advantages of C language is that there is not much vocabulary to learn, and that the programmer can arrange for the program is very fast. C programming language is very easy to learn.
2. **More number of Libraries –** Many numbers of libraries is written in C language.
3. **Base of all languages –** C are a building block of many other currently known high level languages.
4. **Procedural Approach –** C is a procedural oriented language in which programmer creates procedures (functions) to execute their task. This approach is easy to learn as it follows algorithm (set of rules) to execute your statement.
5. **Fast Compilation –** C compiler produces machine code very fast as compared to other language compiler i.e. C compiler can compile around 1000 lines of code in a second or two. C compiler also optimizes code for faster execution.

6. **Portable –** C is easy to install and operate. C language setup is around 3-5 MB so it is easy to carry in your Pen Drive. C language output is exe file which can be executed in any computer without any framework / software.

Disadvantages:

1. Object Oriented Programming Feature – Object oriented programming feature is absent in C.

2. Namespace Feature – C does not provide namespace features i.e. with this feature we can use the same variable name again in one scope.

3. Run type checking is not available - There is no option of run time type checking in C. For example, if a float value is passed in integer type parameter then value will be changed accordingly; it will not give any kind of error message.

4. **Does not support Constructor & Destructor** – As C doesn't support the feature of object oriented programming, so it don't have constructor (for construction of object) and destructor (for destruction of object).

2.3 C TOKENS

A token is a keyword, an identifier, a constant, a string literal, or a symbol. C program is made up of various tokens.

Identifiers - The names of variables, functions, labels and various others user defined items are called as identifiers. Identifiers contain alphabets, digits and underscore.

Identifiers start with a letter Upper Case (A to Z) or Lower Case (a to z) or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. It can't include spaces.

Table2.3.1 Examples of Identifiers

Add()	percent	R	pi
maths1	Name	Class	Roll_no
Area_circle1	xyz	Factorial()	F1a2c3t

Variable –

A variable is a data name that may be used to store a data value. A variable may take different values at different times during the execution of a program. Variables must be defined before they are used in a program.

The rules for variables names are:

1. A variable may consist of letters, digits and underscore (_).
2. The first character must be a letter.
3. Special symbols and blanks are not allowed.
4. Keywords or Reserved words are not allowed.
5. Both uppercase and lowercase letters are allowed and are considered to be distinguishable; i.e. name and NAME are not same as C is case sensitive language.
6. A variable name may be as long as you wish but the first 8 characters of a variable are recognized by C Compilers.

Variables are tokens defined by user and used in programming. It must start with a letter or underscore (_), it can't include spaces, and it can contain digits.

Valid	**Invalid**
int x	int 3x
float area	float area + var
int a_b	int a.b

Difference between Identifier and Variable:

As we have studied in above section the names of variables, functions, labels and other user defined functions are called as identifiers**.**

Identifier	**Variable**
All identifiers are not variables.	All variables are identifier.
Identifier may not have any memory unless it is a variable.	All variables have memory.

Mentioning the type of an identifier is not needed unless it is a variable.	Type of variable must be defined.

Constants – Constants refers to the fixed values which never change during the execution of a program. A "constant" is a number, character, or character of string that can be used as a value in a program. There may be a situation in programming that the value of certain variables to remain constant during execution of a program. In doing so we can use a qualifier **const** at the time of initialization. Example

const int x=20;	const char name[]= “Asha”;	const float pi=3.14;	const char c= ‘g’;

Types of C Constants:

Primary Constants: Primary constants include

- Numeric Constants
  - → Integer Constant
  - → Real Constant
- Character Constant
  - → Single Character Constant
  - → String Constant

Secondary Constants: Secondary constants include

- Array
- Union
- Pointer
- Structure
- Enum. etc.

Numeric Constant-Numeric constant are made up of two parts i.e. Integer and Real constants. These are discussed below in detail.

Integer Constant – An integer constant refers to a sequence of digits without decimal point.

Rules for constructing integer constants are:

1. An integer constant must have at least one digit.
2. An integer constant contains neither a decimal point nor an exponent.
3. Commas and blank spaces cannot be included within the constant.
4. Sign (+ or -) must proceed the number i.e. it can be either positive or negative. Default sign is positive.
5. The allowable range for integer constants for a 16 bit computer is -32768 to +32767.

There are three types of integer constants, namely. Decimal, octal and hexadecimal.

i. **Decimal integers** consist of a set of digits 0 through 9.

Example-

Valid:

12 -78 +65 0 765 678 -56 +78 1

Not Valid:

52.0	Contains decimal point
12,250	Contains a comma
78-	Sign does not proceed
67 80	Blank spaces not allowed

ii. An **Octal integer** constant consists of any combination of digits from 0 through 7, with a leading 0.

Example-

Valid:

014 0567 0 08976 06 77 45 1234

Invalid: Octal must contain digits only from 0 through 7.

569 569 899 989 679 897 989 999 909

iii. A **Hexadecimal integer** constant consist of any combination of digits from 0 through 9 or alphabets a (or A) through f (or F), with a leading of 0x or 0X i.e. this constant must begin with either 0x or 0X.

Example-

Valid:

0x7 0x7ac 0x6f 0xabc 0x978 0xdcb 0xbcd 0xcde 0xdef

Invalid:

Def0X	It should contain 0X in leading i.e. in beginning.
0xfgh	Illegal character g
0xdfg	Illegal character g
ghi0X	Does not begin with 0X
0X78efg	Illegal character g

Real Constant - Real constant are also called **Floating point constant.** A real constant is a number that contains either a decimal point (Fractional form) or an exponent part (Exponential form) / or both. The real constants could be written in 2 forms: Fractional form and Exponential form.

Fractional form real constant-

Rules for real constants in Fractional forms are:

1. A real constant must have at least one digit.
2. A real contain must contain a decimal point.
3. Commas and blank spaces cannot be included within the number.
4. Sign (+ or -) must precede the number i.e. it could be either (+ve) or (–ve). By default sign is positive.

Example-

Valid:

65.5 0.231 0.1 6325.236 23.6 0.0 2345.575 3.26 5.6

Invalid:

65 5	A decimal point missing.
56+	Sign must proceed
36, 56	Illegal character (,)

Exponential form real constants – The general form of real constants expressed in exponential form is

Mantissa e Exponent

Rules for real constants in Exponential forms are:

1. The mantissa is either a real number expressed in decimal notation or an integer.
2. The exponent is always an integer number with a option plus or minus sign.
3. The mantissa part may have a (+) or (-) sign. Default sign is (+).
4. The mantissa part and the exponential part should be separated by a letter in an exponential form 'e'.
5. No commas or blank spaces are allowed.
6. Range of real constants expressed in exponential form is -3.4 e 38 to 3.4 e 38.

Example-

Valid:

0.6e2 -23.2e6 2.34e-8 2.3e9 23e-9 25e25

Invalid:

56,000	(,) are not allowed.
88	Either a decimal point or exponent must be present.
25e2.5	Exponent must be an integer.
@67.78e56	@ signs are not allowed.

Character Constant – Character constant consist of two parts i.e. Single character constant and string constants. Following section discuss in detail these two parts.

Single Character Constant – A single character constant is a single character, enclosed in single quotation mark. The character may be letter, number, or special character.

Example-

Valid:

's' 'd' 'h' 'j' 'k' 'l' 'o' 'p' 'g'

Invalid:

S	Single Quotation missing
'f	Closing quotes missing
k'	Opening quotes missing
'kk'	Single character is allowed

String Constants – A string constant is a sequence of characters enclosed in double quotes ("….."). A string constant may consist of any number of digits letters, escape sequences and spaces. Total number of characters enclosed within the double quotes will be called the length of string constant.

Example-

Valid:

"gh" "3098" "X" "U.S.A" "8+5-8" "Name"

NOTE: That character constant 'A' and the corresponding single character string constant "A" are not equivalent. The string constant "A" consists of character **A** and **\0.** It occupies two bytes, one for the ASCII code of **A** and another for the NULL character with a value **0**, which is used to terminate all strings. However, a single character string constant does not have an NULL character value.

Secondary Constants – It contain Arrays, Union, Enumeration, etc.

Arrays – set of similar data elements are called as arrays.

Ex: int a [10], char name [20];

Union - Group of deferent data items combined together are called as Union. All the data items can share unique memory location.

Enumeration - An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the "enumeration set," "enumerator constants," "enumerators," or "members"). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

2.4 KEYWORDS

These are reserved words in C compiler and each word has different meaning, these cannot be used as names of variables, functions or labels. Example of few keywords are given in following Table2.4.1

Table2.4.1 Example of Keywords

Auto	else	Long	switch
break	enum	Register	typedef
case	extern	Return	union
char	float	Short	unsigned
const	for	Signed	void
continue	goto	Sizeof	volatile
default	if	Static	while
do	int	Struct	double

Literals - The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Various types of literals are

- Integer Literal - Allows values of integer type to be used in expressions directly. An integer literal can be a decimal, octal, or hexadecimal constant.
- Floating-point literal - A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. We can represent floating point literals either in decimal form or exponential

form. While representing using decimal form, we must include the decimal point, the exponent, or both and while representing using exponential form; we must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

- Character Literals – Theses are enclosed in single quotes e.g., 'x' and can be stored in a simple variable of **char** type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t').

2.5 ESCAPE SEQUENCE

These Escape sequence are those certain characters in C when they are preceded by a back slash (\) they will provide a special meaning. Escape sequences are used in input and output functions such as printf and scanf. Following Table shows few Escape sequences and there meaning.

Table2.5.1 Escape Sequences

Escape Sequences	Meaning
\a	Alert or Bell
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

2.6 SYMBOLIC CONSTANT

A symbolic constant is name that substitute for a sequence of character that cannot be changed. The character may represent a numeric constant, a character constant, or a string. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. They are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc., that the symbolic constants represent.

Example – C program consists of the following symbolic constant definitions.

 # define PI 3.141593

It defines a symbolic constant PI whose value is 3.141593. When the program is pre-processed, all occurrences of the symbolic constant PI are replaced with the replacement text 3.141593.

Note - That the pre-processor statements begin with a #symbol, and are not end with a semicolon. By convention, pre-processor constants are written in UPPERCASE.

2.7 COMMENTS

Comment is a non-executable statement in C when a programmer includes comments the program can be easily understandable i.e. it increase the readability and understanding. Comments are like helping text in your C program and they are ignored by the compiler. There are two ways of writing comments:

Single Line- Comments on single line are added by using double slash "//".

Example- // Hello.

Multiple Line: Comments on multiple line are added by enclosing the lines in "/* and */".

Example-/* *Hello. This is to explain you about comments */*

As shown in above example comment start with /* and terminates with the */.

2.8 C INSTRUCTIONS

Before we study the detail instructions of C language let's have a C- basic structure.

C program is made up of

- Preprocessor directives
- Functions
- Variables
- Statements & Expressions
- Constants

Let us look at a simple code that would print the words "This is my first C program":

#include<stdio.h>

main( )

{

// my first C Program

printf ("This is my first C program \n");

return0;

}

Let us look various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a **Preprocessor command** which tells a C compiler to include stdio.h (Standard I/O)header file before going to actual compilation.

Preprocessor Commands –

Lines that begin with a pound sign (#). The C preprocessor is a program that process the source program before it is passed to the compiler. These preprocessor commands are often called as directives. The preprocessor works on the source code/program and creates 'Expanded Source Code'. It is the expanded source code that is sent to the compiler for compilation. Example-

2. The next line *main ()* is the main function where program execution begins. The two braces, {and}, signify the begin and end segments of the program. It contains statements in these braces.

   Statements - A statement is a specification of an action to be taken by the computer as the program executes. Compound Statements is a list of statements enclosed in braces, { }.

3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line *printf(...)* is another function available in C which causes the message "This is my first C program" to be displayed on the screen. These are called Input/ Output functions of C language.
5. /n after the message "This is my first C program \n "helps the cursor to move on next line and is known as Escape Sequence discussed in above section.

Example - const double CITY_TAX_RATE = 0.0175;

Named Constants - It is an identifier whose value is fixed and does not change during the execution of a program in which it appears. In C the declaration of a named constant begins with the keyword const. During execution, the processor replaces every occurrence of the named constant.

9. In C program, the semicolon (;) is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Example- *printf ("This is my first C program \n")**;***

Input/ Output functions of C language –

All these input and output functions are declared in <stdio.h>. Table2.8.1 shows the various Input/ Output functions available in C language.

Table2.8.1 Input and Output Functions

Function	Type	Meaning
printf	Output	Writes formatted text on the screen.
scanf	Input	Reads formatted text from keyboard.
getc	Input	Reads a character from stream.
putc	Output	Writes a character into a stream.
gets	Input	Reads a string from keyboard.
puts	Output	Writes string on to the screen.

Declaration of variables -

All variables must be declared before we use them in C program, although certain declarations can be made implicitly by content. A declaration specifies a 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, int, float, double, or any user defined data type etc., and *variable_list* may consist of one or more identifier names separated by commas.

Some valid variable declarations along with their definition are:

int i;

flot percentage;

float pi;

We can also declare at the time of declaration i.e.

type variable_name=value;

Example -

float pi=3.14;

An **extern declaration** is not a definition and does not allocate storage. In effect, it claims that a definition of the variable exists somewhere else in the program. A variable can be declared multiple times in a program, but it must be defined only once. Following is the declaration of a variable with **extern** keyword:

extern int i ;

Duplicate /although we can declare a variable multiple times in C program but it can be declared only once in a function or a block of code.

2.9 DATA TYPES

C is rich in data types. Data type defines the type of data a variable will hold. Data type refers to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. If a variable 'Z' is declared as 'int' it means Z can hold only integer values. Every variable which is being used in the program must be declared as what data-type it is. The verity of data types allow the programmer to select appropriate data type to satisfy the need of application as well as the needs of different machine.

A data type is used to

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

Variable's type determine how it is stored internally, what operations can be applied to it, How such operations are interpreted.

Data Types

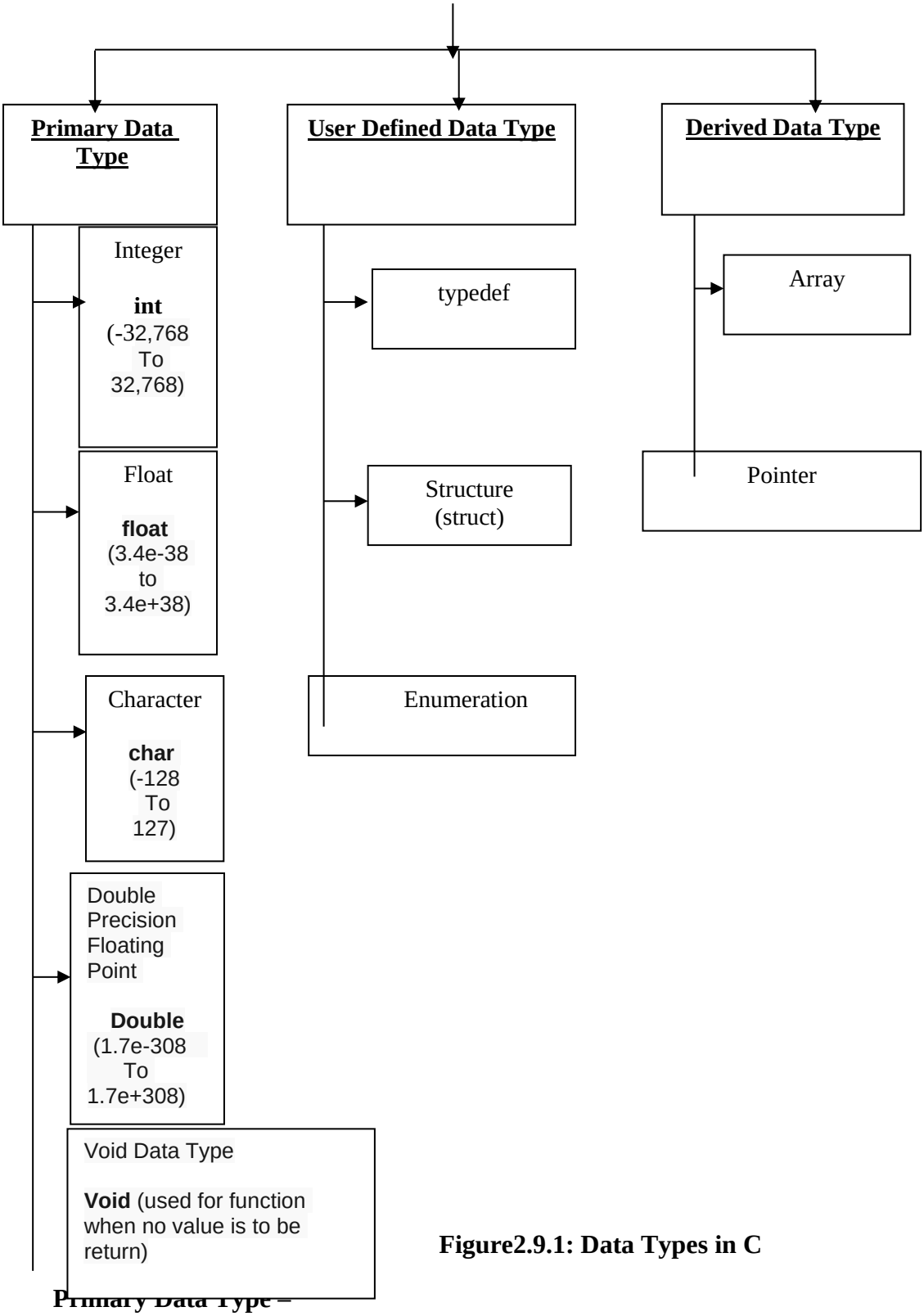


Figure2.9.1: Data Types in C

Primary Data Type –

Integer Data Type – The following Table2.9.1 shows the detail of standard integer data type and memory storage size and value ranges.

Table2.9.1 Integer Type

Type	Storage size	Value range
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating Data type - The following Table2.9.2 shows the detail of standard float data type and memory storage size, value ranges and Precision.

Table2.9.2 Float Type

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Note – Do not use increment (++) or decrement with floating point variables.

Character Data Type – The following Table2.9.3 shows the detail of standard character data type and memory storage size and value ranges.

Table2.9.3 Character Data Type

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127

Double Precision Floating Point Data type – The following Table2.9.4 shows the detail of standard Double Precision Floating Point Data Type and memory storage size and value ranges.

Table2.9.4 Double Precision Floating Point Data Type

Type	Storage size	Value range
double	64	1.7E - 308 to 1.7E + 308
long	80	3.4E _ 4932 to 1.1E + 4932

Note- To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions

sizeof (type);

yields the storage size of the object or type in bytes.

Void Data Type –

It does not return any value.

Table2.9.1 Data Type

Type	Description
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

User Defined Data Type –

Typedef –

By using a feature known as **"type definition"** that allows user to define an identifier that would represent a data type using an existing data type i.e. we can define a new name for an existing data type.

General form:

typedef type identifier;

Or to better understand

typedef existing_data_type new_user_define_data_type

Example –

typedef int number;

typedef long big_number;

typedef float decimal;

typedef double big_decimal;

/ Now we can use above user defined types to declare variables*/*

number visitors=25;

big_number population=12500000;

decimal radius = 2.5;

big_decimal pie=3.14;

Note: Main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.

Structure - A collection of related variables of the same or different data types.

Example -

```
struct student
{
char name [20];
int rollno;
float marks;
};        //Collection of different data types
struct student student = {“Asha”, 10011018, 98};
```

Enumeration – The *enumerated type* is a user-defined type based on the standard integer type.” We give each value in an enumerated type an identifier that we call an *enumeration constant*, which we can use as a symbolic name.

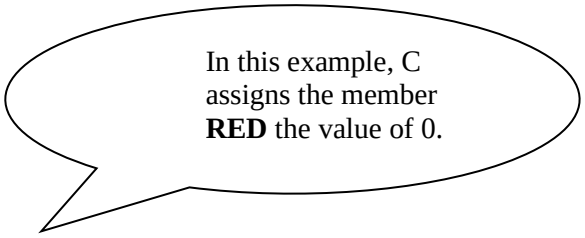
Syntax:

```
enum typeName
{
        Identifier list
};
```

If we don’t assign values to the members of the identifier list, C will assign values, beginning at 0.

Example –

```
enum color
{
```



Red,

Yellow,

Blue,

Green,

};

Derived Data Type – They include (a) Pointer types, (b) Array types, (c) Union types and (d) Function types.

Array - A finite collection of data of same types or homogenous data type.

Example -

int roll_no [20]; //Array to contain roll number of 20 students

Pointer - Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with /without pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.

A **pointer** is a variable whose value is the address of another variable i.e. direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

*type *var_name*

Example –

int *ip //Pointer to integer

This was to give you brief idea about all these Data Types; these will be discussed more in further sections.

Format Specifier or Conversion Character

Conversion characters or Format Specifiers are used to provide the format for the value to be print. It has a prefix (%) and followed by a specifier. Below Table2.9.1 shows some Format Specifier

Table2.9.1 Format Specifier

%d (%i)	Prints int signed integer
%f	Prints float
%e	Prints double exponential format
%c	Prints single character
%s	Prints string
%o	Prints octal
%u	Prints Unsigned Integer
%x	Prints Hexadecimal

Programming Example –

Program - Write a Program to print “Hello World”.

```
#include <stdio.h>
 #include<conio.h>
main()
{
  printf("HELLO WORLD\n");
}
```

Output:

HELLO WORLD

Program - Write a Program to first input an integer and then print that integer.

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

```
{
  int a;
  printf ("Enter an integer \n");
  scanf ("%d", &a);
  printf("Integer that you have entered is %d \n", a);
}
```

Output:

Enter an integer

90

Integer that you have entered is 90

Note - Input is done using scanf function and output on screen is given using printf function.

Program - Write a program that asks the user for 4 characters and displays them on the screen.

```
#include <stdio.h>
 #include<conio.h>
main)
{
 /* declarations */
char letter1, letter2, letter3, letter4;
printf ("Enter a name: ");
scanf ("%c%c%c%c", &letter1, &letter2, &letter3, &letter4);
printf ("You entered: %c%c%c%c", letter1, letter2, letter3, letter4);
printf (" / Backwards: %c%c%c%c\n", letter4, letter3, letter2, letter1);
}
```

Output:

Enter a name: Asha

You entered: Asha / Backwards: ahsA

Note: Ampersand(&) before the variables in scanf() function is must. It is an 'Address of' operator. It gives the location number used by the variable in memory. &letter1 tells scanf() at which memory location should it store the value supplied by the user. The detail is further explained in Lesson of Pointers.

Program - Write a program to add two integer numbers.

```
#include <stdio.h>
 #include<conio.h>
main()
{
 int a, b, c;
printf("Enter two numbers to add \n");
scanf("%d%d",&a,&b);
c = a + b;
printf("Sum of entered numbers = %d \n",c);
}
```

Output:

Enter two numbers to add

3

4

Sum of entered numbers = 7

Program - Write a program to perform basic arithmetic operations which are addition, subtraction, multiplication and division of two numbers.

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

```
{
int first, second, add, subtract, multiply;
float divide;
printf("Enter two integers\n");
scanf("%d %d",&first,&second);
add       = first + second;
subtract = first - second;
multiply = first * second;
divide    = first / (float)second;   //typecasting
printf("Sum = %d\n",add);
printf("Difference = %d\n",subtract);
printf("Multiplication = %d\n",multiply);
printf("Division = %f \n",divide);
}
```

Output:

Enter two integers

5 5

Sum = 10

Difference = 0

Multiplication = 25

Division = 1.0

***Typecasting** 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 values from one type to another explicitly using the cast operator as follows:

(type_name) expression

In the above example we want to store a int value into a float then we can type cast int to float.

Program - Write a program to calculate area and circumference of circle.

```
#include <stdio.h>
#define PI 3.1416
main ()
{
double diam, area, circ, r;    /* declarations */
printf ("Enter a value for the diameter: "); /* get diameter from user */
scanf ("%lf", &diam);
 /* do the computations */
r = diam / 2;
area = PI * r * r;
 circ = 2 * PI * r;
/* display the report on the screen */
printf ("\n A circle with a diameter of %3.1lf cm, ", diam);
printf ("has an area of %5.3lf cm\u00b2\n", area);
printf ("and a circumference of %4.2lf cm. \n", circ);
}
```

Output:

Enter a value for the diameter: 10

A circle with a diameter of 10.0cm, has an area of

78.540 cm2

and a circumference of 31.42 cm.

2.9 SUMMARY

In the above lesson we have seen

- The basic structure of C language.
- C programs are easy to read, maintain, fast, efficient, effective, reusable, reliable, and modular.
- Various data type available in C, C tokens, Keywords, Escape sequence, Symbolic Constant.
- Comments are non-executable statement of a C program.
- #include statement contains a reference to a special file called a header file.
- The location value of a variable is the address in memory at which its data value is stored.

2.10 KEYWORDS

Constant – A named data item whose value does not change throughout the execution of the program.

Data type – The characteristic of a data value or variable that determines the type of operations applicable to it and its size.

Whitespace – The characters that separate one identifier from the other like space, tab, new line, etc.

Keywords – Keywords are also referred to as verb or reserve word, is a sequence of characters that is reserved by the compiler and have special meaning to the C language.

Identifier – A string of characters representing the name to identify a variable, function etc.

2.11 REVIEW QUESTIONS

1. What do you mean by identifier and variable? What is the difference between them?
2. What do you mean by escape sequence?
3. Write a detail about various data type available in C?
4. What are comments? Explain it with the help of example?
5. What is a string constant?

2.12 FURTHER READINGS

[1] E.Balaguruswamy, “Programming in ANSI C”, Tata McGraw Hill.

[2] Brian W.Kernighan & Dennis M.Ritchie, “The C Programming Language”, Prentice Hall.

[3] R.Hutchison, “Programming in C”, Tata McGraw Hill.

[4] A.K.Sharma, “Fundamentals of Computers & Programming with C”, Ganpat Rai Publications.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 3	
OPERATORS AND EXPRESSIONS	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

3.1 OBJECTIVES

The main objective of this lesson is to make the students have introduction about the various operators available in C. All computer languages provides some operators to perform predefined operations as a progress required to perform a lot more than just simple input and output operations. An operator in general, is a symbol that operates on a certain data type. C language is very rich in operators. In this unit we'll examine the commonly used operators with example and special notes on their use.

3.2 INTRODUCTION

We can now create and initialize variables, but how do we modify the data? The answers to use operators, which as the name suggests, operate on the data. The built-in operators in C may be broadly subdivided into four classes' arithmetic, relational, logical and bitwise.

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators various type of operators. There are two terms in this context Operator and Operand. Operator is symbol that tell C compiler to perform some operation, action on one or more operands. An Operand is a data item over which an operator or operators can act upon.

As operators are the tools to perform some predefined operations, there are only five basic operations which can be performed by most computers as in,

- To store and retrieve data in binary form.
- To communicate not only with the outside world (I/O), but also with their separate components parts.
- To take logical decisions.
- To perform basic arithmetic operations.
- To perform bitwise logical operations, such as the NOT, AND, and OR operations.

C provides the basic operations i.e. last three that includes basic arithmetic, bitwise logical operation and in addition to this, C has many higher-level operators that correspond directly with memory allocation and retrieval. C has no high level operators concerned with I/O but it does have the register data class that encourages special memory to CPU communications.

There two approaches on the basis of which operators can be classified,

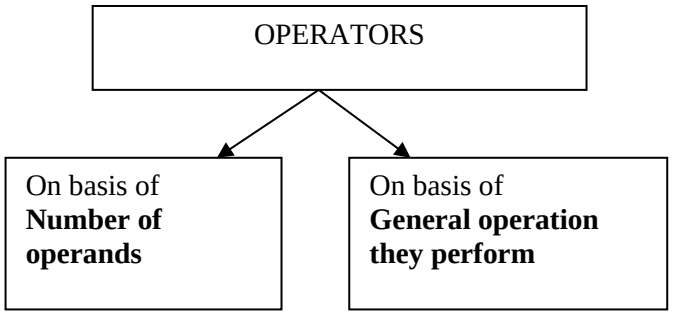


Figure3.2.1 Operators Classification

On the basis of **Number of operands** involved the classification can be done as:

1. **Unary operators**: These can be applied on only one operator.

Example - (++, --, size of)

2. **Binary operator**: These can be applied on two operands.

Example - (+, *)

3. **Ternary operator**: Theses can be applied on three operands.

   Example - (? :)

In the current context we will talk about the classification based on the **Operations** it performs. C provides various types of operators. These operators can be broadly categorized as refer Figure3.2.2

Figure3.2.2 Various Operators in C

3.2.1 Arithmetic Operators –

The Arithmetic operators perform Arithmetic operations. The arithmetic operators can operate on any built in data types. Following Table3.2.1.1 shows all the arithmetic operators supported by C language.

Table3.2.1.1 Arithmetic Operators

Operator	Meaning
+	Addition or Unary plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulo Division

This table shows the symbols of arithmetic, together with their duties. These operators allow you to write expressions whose evaluation is precisely the treatment of information that made the computer. Arithmetic operators, along with a wide range of features resident in the library of the language used make it possible to perform calculations of all kinds.

Assume variable A holds 10 and variable B holds 20. Several arithmetic expressions involving these variables are shown below in Table3.1.1.2, together with their resulting values.

Table3.2.1.2 Arithmetic Operators Example

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

Program – Write a program using arithmetic operator.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a = 21;
```

```
int b = 10;
int c ;
c = a + b;
printf("Line 1 - Value of c is %d \n", c );
c = a - b;
printf("Line 2 - Value of c is %d \n", c );
c = a * b;
printf("Line 3 - Value of c is %d \n", c );
c = a / b;
printf("Line 4 - Value of c is %d \n", c );
c = a % b;
printf("Line 5 - Value of c is %d \n", c );
}
```

Output –

Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

3.2.2 Relational Operators –

Relational operators are used to compare, logical, arithmetic and character expression. Each of these six relational operators takes two operands. Each of these operators compares their left side with their right side. The whole expression involving the relation operator then evaluate to an integer.

It evaluates to = 0 if the condition is false;

& = 1 if it is true.

Following Table3.2.2.1 shows all the relational operators supported by C language.

Table3.2.2.1 Relational Operators Description

Operator	Description
= =	Checks if the value of two operands is equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The concept of comparison certainly implies different meanings depending on the data type of the variables compared. Operators presented in this table are suitable for comparing numeric values or ordinal (individual characters or enumerated values), but not to compare strings or lists of any kind. String comparison requires the use of features designed specifically for this purpose, like strcmp (). (This topic is further explained in detailed in upcoming few chapters).

Assume variable A holds 10 and variable B holds 20 then:

Table3.2.2.2 Relation Operator Example

Operator	Interpretation	Value

= =	(A = = B) is not true.	0
!=	(A! = B) is true.	1
>	(A > B) is not true.	0
<	(A < B) is true.	1
>=	(A >= B) is not true.	0
<=	(A <= B) is true.	1

Program – Write a program to show example of relational operator.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a = 21;
int b = 10;
int c ;
if ( a == b )
{
printf("Line 1 - a is equal to b\n" );
}
else
{
printf("Line 1 - a is not equal to b\n" );
}
```

```
if ( a < b )
{
printf("Line 2 - a is less than b\n" );
}
else
{
printf("Line 2 - a is not less than b\n" );
}
if ( a > b )
{
printf("Line 3 - a is greater than b\n" );
}
else
{
printf("Line 3 - a is not greater than b\n" );
}
/* Lets change value of a and b */
a = 5;
b = 20;
if ( a <= b )
{
printf("Line 4 - a is either less than or equal to b\n" );
}
if ( b >= a )
{
```

printf("Line 5 - b is either greater than or equal to b\n");

}

}

Output –

Line 1 - a is not equal to b

Line 2 - a is not less than b

Line 3 - a is greater than b

Line 4 - a is either less than or equal to b

Line 5 - b is either greater than or equal to b

3.2.3 Logical Operators –

A Logical operator is used to compare or evaluate logical and relational expression. There are three logical operators in C language. They are shown in Table 3.2.3.1

Table3.2.3.1 Logical Operators Description

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

Logical operators are used to calculate the value of logical expressions i.e. the only values that can be taken by operands are (True) or (False) i.e. two versions of these operators. When you simply want to know if an expression is true or false; example: x>7, operators used are logical. These operators do not

consider the structure of bits; simply take the value 0 as false and any other as true.

Suppose that A and B are integer variables whose values are 100 and 4, respectively. Several arithmetic expressions involving these variables are shown in Table3.2.3.2, together with their resulting values.

Table3.2.3.2 Logical Operators Example

Example	Interpretation	Value
(a>b)&&(a=100)	True	1
(b>a)‖(a>b)	True	1
!(a>b)	False	0

There is also another version (the bitwise) imposed by the logical operations to the bits used to internally represent the operands. These are detailed in next section which is known as Bitwise Operators.

Program – Write to program to show example of Logical operator

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a = 5;
int b = 20;
int c ;
if ( a && b )
{
printf("Line 1 - Condition is true\n" );
}
if ( a || b )
```

```
{
printf("Line 2 - Condition is true\n" );
}
/* lets change the value of a and b */
a = 0;
b = 10;
if ( a && b )
{
printf("Line 3 - Condition is true\n" );
}
else
{
printf("Line 3 - Condition is not true\n" );
}
if ( !(a && b) )
{
printf("Line 4 - Condition is true\n" );
}}
```

Output –

Line 1 - Condition is true

Line 2 - Condition is true

Line 3 - Condition is not true

Line 4 - Condition is true

3.2.4 Assignment Operators –

An Assignment operator is used to form an assignment expression, which assigns the value to an identifier. The most commonly used assignment operator is (=). Assignment expressions that make use of this operator are written in the form:

Identifier=expression;

Example -

i=100;

j=200;

Other forms of assignment operator exist that are obtaining various operators such as +, - , *, /, % etc. with the = sign. Following Table3.2.4.1 shows the various operators and there description.

Table3.2.4.1 Assignment Operators Description with Example

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Program – Program to show assignment operator

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=21;
int c;
c = a;
printf (“Line1- = Operator Example, Value of c =%d\n”, c);
c+= a;
printf (“Line2- += Operator Example, Value of c = %d\n”, c);
c-=a;
printf (“Line3- -= Operator Example, Value of c = %d\n”, c);
c*=a;
```

```
printf ("Line4- *= Operator Example, Value of c = %d\n", c);
c/=a;
printf ("Line5- /= Operator Example, Value of c = %d\n", c);
c=200;
c%=a;
printf ("Line6- %= Operator Example, Value of c = %d\n", c);
c<<=2;
printf ("Line7- <<= Operator Example, Value of c = %d\n", c);
c>>=2;
printf ("Line8- >>= Operator Example, Value of c = %d\n", c);
c&=2;
printf ("Line9- <<= Operator Example, Value of c = %d\n", c);
c^=2;
printf ("Line10- ^= Operator Example, Value of c = %d\n", c);
c|= 2;
printf ("Line11- |= Operator Example, Value of c = %d\n", c);
}
```

Output –

```
Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
```

Line 8 - >>= Operator Example, Value of c = 11

Line 9 - &= Operator Example, Value of c = 2

Line 10 - ^= Operator Example, Value of c = 0

Line 11 - |= Operator Example, Value of c = 2

3.2.5 Increment and Decrement Operators

C offers two unusual and unique operators ++ and -- called increment and decrement operator respectively. The use of these two operators will results in the value of the variable being incremented or decremented by unity.

i=i+1 can be written as i++; and

j=j-1 can be written as j- -.

The operators can be place either before or after the operand.

If the operator is placed before the variable (++i and --i) it is known as **pre incrementing** and **pre decrementing** respectively.

If the operator is placed after the variable (I++ and I--) it is known as post incrementing and post decrementing respectively.

Pre incrementing or pre decrementing or post incrementing or post decrementing have different effect when used in expression. In the pre increment or pre decrement first the value of operand is incremented or decremented then it is assigned.

Example -

```
x=100;
y=++x;
```

After the execution the value of y will be 101 and the value of x will be 100.

In the post increment or post decrement first the value of operand is assigned to the concern variable and then the increment or decrements perform.

Example -

x=100;

y=x++;

After the execution the value of y will be 100 and the value of x will be 101.

Program – Write a program for Increment and Decrement Operator.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
int p,q,x,y;
clrscr();
printf("Enter the value of  x \n");
scanf("%d",&x);
printf("Enter the value of y \n");
scanf("%d",&y);
printf("x=%d\ny=%d\n",x,y);
p=x++;
q=y++;
printf("x=%d\ty=%d\n",x,y);
printf("p=%d\tq=%d\n",p,q);
p=--x;
q=--y;
printf("x=%d\ty=%d\n",x,y);
printf("p=%d\tq=%d\n",p,q);
```

getch();

}

Output –

Enter the value of x 10

Enter the value of y 20

x=10

y=20

x=11 y=21

p=10 q=20

x=10 y=20

p=10 q=20

3.2.6 Conditional Operator –

The conditional expression can be used as shorthand for some if-else statements. It is a ternary operator. This operator consists of two symbols: the question mark (?) and the colon (:).

The general syntax of the conditional operator is:
Identifier = (test expression)? Expression1: Expression2;

This is an expression, not a statement, so it represents a value. The operator works by evaluating test expression. If it is true (non-zero), it evaluates and returns expression1. Otherwise, it evaluates and returns expression2.

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of

if (x < y)
{

```
min = x;
}
else
{
min = y;
}
```

You just say...

```
min = (x < y) ? x: y;
```

Suppose that x and y are integer variables whose values are 100 and 4, respectively. After executing above statement, the value of min is 100.

This is the only operator in C that makes use of three operands. It tests condition corresponding to test expression, if it is true the value corresponding to Expression1 and it is false it corresponds to the value of Expression2.

Program - Write a program for Conditional operator.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,result,choice;
clrscr();
printf("Enter first number\n");
scanf("%d",&a);
printf("Enter second number\n");
scanf(%d",&b);
printf("Enter 1 for addition or 2 for multiplication\n");
scanf("%d",&choice);
result=(choice==1)? a+b : (choice==2)? a*b : printf("Invalid Input");
```

if(choice==1|| choice==2)

printf ("The result is %d \n\n",result);

getch();

}

Output –

Enter first number

10

Enter second number

3

Enter 1 for addition or 2 for multiplication

2

The result is 30

3.2.7 Bitwise Operators –

Bitwise operators work on bits and perform bit by bit operations. When bit wise operators are used on operands they convert the numbers into binary system and then apply bit wise operators. Various types of Bitwise Operators used by C compiler are described in below Table3.2.7.1.

Table3.2.7.1 Bitwise Operators Description

Operator	Description
&	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 Ones 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. (multiply by power of 2)
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. (divide by power of 2)

The truth tables for &, |, and ^ are as follows:

Table3.2.7.2 Bitwise Operators Example

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

For Example -

Bitwise Negation

p = 00000011

~p = 11111100

Bitwise AND

p = 00000011

q = 00010110

p & q = 00000010

Bitwise OR

p = 00000011

q = 00010110

p | q = 00010111

Bitwise Exclusive OR

p = 00000011

q = 00010110

p ^ q = 00010101

Right Shift

p = 0000 0011 = 3

(p<<=1) = 00000110 = 6

(p<<=2) = 00011000 = 24

(p<<=3) = 11000000 = 192

Left Shift

p = 11000000 = 192

(p>>=1) = 01100000 = 96

(p>>=2) = 00011000 = 24

(p>>=3) = 0000 0011 = 3

3.2.8 Special Operators –

There are few special operators that are available in C. These are

Unary Operators

The classes of operator that act upon a single operand to produce a new value are known as Unary operator. The frequently used Unary operators in C are

1. Unary minus
2. sizeof operator

Unary Minus

The operand of Unary minus operator must have arithmetic type, and the result is the –ve of its operand. The operand can be numerical constant, variable or expression. It is different from arithmetic subtraction operator (requires two operand).

Example:

-(p+q)

x=-y

-10*(p+q)

Program - Write a program for Unary Minus.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
clrscr();
a=-100;
printf("The value of a is %d",a);
a=-(10+20);
printf("The value of a is %d",a);
a=-(10+20*-2);
printf("The value of a is %d",a);
getch();
}
```

Output:

The value of a is -100

The value of a is -30

The value of is 30

sizeof Operator –

The sizeof operator returns the size of its operand in bytes. The sizeof operator always precedes its operand. The operand may be an expression or it may be a cast.

Example:

Suppose i is an integer variable and c is a character type variable then

printf("Size of integer : %d", sizeof(i));
printf("Size of character : %c", sizeof(c));

Might generate the following output:
size of integer: 2
size of character: 1

Program - Write a program to show example of Size of operator.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
float b;
double c;
char d;
clrscr();
printf("Size of integer :%d \n \n", sizeof(a));
printf("Size of Floating point :%d \n \n", sizeof(b));
printf("Size of Double :%d \n \n", sizeof(c));
printf("Size of Character :%d \n \n, sizeof(d));
```

getch();

}

Output:

Size of Integer: 2

Size of Floating point: 4

Size of Double: 8

Size of Character: 1

Comma Operators

A set of expression separated by comma is a valid constant in the C language. For example i and j are declared by the statements

int i , j;
i= (j=10, j+20);

In the above declaration, right hand side consists of two expressions separated by comma. The first expression is j=10 and second is j+20. These expressions are evaluated from left to right i.e. first the value 10 is assigned to j and then expression j+20 is evaluated so the final value of i will be 30.

Program – Write a program to illustrate example of Comma Operator.

#include<stdio.h>

#include<conio.h>

void main()

{

int i,j;

clrscr();

i =(j=10,j+20);

printf("i=%d \n j=%d \n "i, j);

getch();

}

Output –

i=30

j=10

Shorthand Assignment Operator –

Form of assignment operator is

Variable operator = expression

The assignment statement

Variable operator = expression

Is equivalent to:

Variable=variable operator (expression)

Example –

a+=10 is equivalent to a=a+10

a-=10 is equivalent to a=a-10

a*=10 is equivalent to a=a*10

a/=10 is equivalent to a=a/10

a%=10 is equivalent to a=a%10

3.3 ARITHMETIC EXPRESSIONS

Expressions in C can get rather complicated because of the number of different types and operators that can be mixed together. Expressions in C are built from combinations of *operators* and *operands*, so for example in this expression

x = a+b*(-c)

we have the operators =, + * and -.The operands are the variables x, a, b and c. You will also have noticed that parentheses can be used for grouping sub-expressions such as the -c. Most of C's unusually rich set of operators are either *binary operators*, which take two operands, or *unary operators*, which take only one. In the example, the - was being used as a unary operator, and is performing a different

task from the binary subtraction operator which uses the same - symbol.

Some of the operators have both a binary and a unary form where the two meanings bear no relation to each other; a good example would be the binary multiplication operator *, which in its unary form means indirection via a pointer variable.

Example - **variable name* i.e. *b means pointer to b variable.

A peculiarity of C is that operators may appear consecutively in expressions without the need for parentheses to separate them. The previous example could have been written as

x = a+b*-c;

and still have been a valid expression. An expression may consist of single entity or some combination of such entities interconnected by one or more operators. All expression represents a logical connection that's either true or false. Thus logical type expression actually represents numerical quantities.

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. Some examples of expressions are:

a+b;

3.14*r*r;

a*a+2*a*b+b*b;

Because of the number of operators that C has(which are already discussed in above sections), and because of the strange way that assignment works, the *precedence* of the operators (and their *associativity*) is of much greater importance to the C programmer than in most other languages. It will be discussed fully in the next section.

Program - Write a program to illustrate basic example of C arithmetic expressions.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int a=123,b;
```

b=a+2;

clrscr();

printf("Value of a =%d\n",a);

printf("Value of b=%d",b);

getch();

return 0;

}

Output -

Value of a=123

Value of b=125

3.4 EVALUATION EXPRESSIONS

After looking at the operators we have to consider the way that they work together. For things like addition it may not seem important; it hardly matters whether

a + b + c

is done as

(a + b) + c

or

a + (b + c)

does it? Well, yes in fact it does. If a+b would overflow and c held a value very close to -b, then the second grouping might give the correct answer where the first would cause undefined behavior. The problem is much more obvious with integer division:

a/b/c

gives very different results when grouped as

a/(b/c);

or

(a/b)/c;

Let's try it with a=10, b=2, c=3. The first gives 10/ (2/3); 2/3 in integer division gives 0, so we get 10/0 which immediately overflows.

The second grouping gives (10/2), obviously 5, which divided by 3gives 1.

The grouping of operators like that is known as *associativity*. The other question is one of *precedence*, where some operators have a higher priority than others and force evaluation of sub-expressions involving them to be performed before those with lower precedence operators. This is almost universal practice in high-level languages, so we 'know' that

a + b * c + d

groups as

a + (b * c) + d

indicating that multiplication has higher precedence than addition.

Here operators with the highest precedence appear at the top of the Table3.4.1; those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Table3.4.1. Operator Precedence

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

Program - Write a program to illustrate the example of evaluation expression.

```
#include <stdio.h>
#include<conio.h>
int main()
{
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
e = (a + b) * c / d; // (30 * 15 ) / 5
printf ("Value of (a + b) * c / d is : %d\n", e );
e = ((a + b) * c) / d; // (30 * 15) / 5
printf ("Value of ((a + b) * c) / d is : %d\n" , e );
e = (a + b) * (c / d); // (30) * (15/5)
printf ("Value of (a + b) * (c / d) is : %d\n", e );
```

*e = a + (b * c) / d; // 20 + (150/5)*

*printf ("Value of a + (b * c) / d is : %d\n" , e);*

return 0;

}

Output –

*Value of (a + b) * c / d is : 90*

*Value of ((a + b) * c) / d is : 90*

*Value of (a + b) * (c / d) is : 90*

*Value of a + (b * c) / d is : 50*

3.5 TYPE CONVERSION

Typecasting 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 values from one type to another

explicitly using the cast operator as follows:

(type_name) expression

The name of the data type to which the conversion is to be made is enclosed in parenthesis and placed directly to the left of the expression whose value is to be converted.

Example:

int i= (int)3.14;
float j= (float)10/2

Consider the following example with help of program where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

Program – Write a program to illustrate the example of cast operator.

#include <stdio.h>

#include<conio.h>

```
void main()
{
int sum = 17, count = 5;
double mean;
mean = (double) sum / count;
printf("Value of mean : %f\n", mean );
}
```

Output –

Value of mean: 3.400000

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character in an int.

Program - A program for adding a character in an integer.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int i = 17;
char c = 'c';          /* ascii value is 99 */
int sum;
sum = i + c;
```

printf ("Value of sum: %d\n", sum);

}

Output –

Value of sum: 116

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

Value of sum: 116. Here value of sum is coming as 116 because compiler is doing integer promotion and converting the value of 'c' to ASCII before performing actual addition operation.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values in a common type. Compiler first performs integer promotion; if operands still have different types then they are converted to the type that appears highest in the following hierarchy shown in Figure 3.5.1:

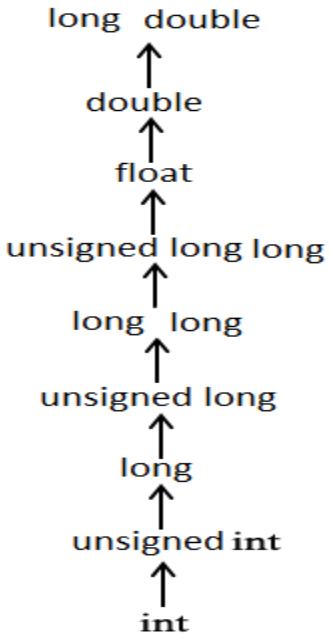


Figure 3.5.1: Hierarchy of integer types

The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take following example to understand the concept:

Program - A program for illustrate arithmetic conversions not performed for assignment operators

& not for logical operators.

```
#include <stdio.h>
#include<conio.h>
void main()
{
int i = 17;
char c = 'c'; /* ascii value is 99 */
float sum;
sum = i + c;
printf ("Value of sum : %f\n", sum );
}
```

Output –

Value of sum: 116.000000

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

Value of sum: 116.000000. Here it is simple to understand that first c gets converted to integer but because final value is double, so usual arithmetic conversion applies and compiler convert i and c into float and add them yielding a float result.

Program – Write a program to illustrate the concept of type conversion

```
#include <stdio.h>
#include<conio.h>
void main()
{
int a=100;
float b=3.1452;
double c=32.5264158;
```

```
char d='c';
clrscr();
printf("a=%d\nb=%f\nc=%lf\nd=%c\n"a,b,c,d);
printf("(float)a        :%f\n",(float)a);
printf("(float)b        :%f\n",(float)b);
printf("(float)c        :%f\n",(float)c);
printf("(float)d        :%f\n",(float)d);
printf("(int)a          :%f\n",(int)a);
printf("(int)b          :%f\n",(int)b);
printf("(int)c          :%f\n",(int)c);
printf("(int)d          :%f\n",(int)d);
printf("(char)a         :%f\n",(char)a);
printf("(char)b         :%f\n",(char)b);
printf("(char)c         :%f\n",(char)c);
printf("(char)d         :%f\n",(char)d);
getch();
}
```

Output –

```
a=100
b=3.1452
c=32.5264158
d=c
(float)a        :100.000000
(float)b        :3.145200
(float)c        :32.5264158
```

(float)d :99.000000

(int)a :100

(int)b :3

(int)c :32

int(d) :99

char(a) :d

char(b) :✓

char(c) :

char(d) :c

3.6 SUMMARY

- C supports many types of operators such as arithmetic, relational, logical etc.
- The binary arithmetic operators are +, -, *, /, and the modulus operator %.
- The relational operators are >,>=, <, <=. They all have the same precedence. Just below them in precedence are the equality operators: ==, ! =.
- Relational operators have lower precedence than arithmetic operators.
- Logical operators are &&.||and!
- Two most useful operators namely increment operator ++ and decrement operator --, these are further categorized as Pre-Increment & Post-Increment and Pre-decrement and Post-decrement.
- When used as standalone expression ++a and a++ are both equivalent to assignment a=a+1. And --a and a-- are both equivalent to assignment a=a-1.
- Bitwise operator includes &,|,^,~,<<,>>.
- Conversion of one data type to another type is called as Type Casting.

3.7 KEYWORDS

Operands - Operands are things the operators work on. They are usually constant values or the names of variables.

Operators - Operators are the things which do the work. They specify the operation to be performed on the operands. Most operators work on two operands, one each side.

Bitwise Operator – An operator that operates on the individual bits of the input value.

Type Casting – Specifying the data type before a value in order to convert one data type to another compatible data type.

Ternary Operator – An operator that takes three operators (i.e. conditional operator).

3.8 REVIEW QUESTIONS

1. What is an expression? What is its component?
2. What is an operator? Describe several different types of operator available in C?
3. What are unary operators? Describe the various unary operators?
4. What are Relational operators? With what type of operands can they be used?
5. Explain the order of precedence of operators.
6. What is the output of the following

a.
```
#include<stdio.h>
main()
{
int k=20;
int i=20, j=0;
j=i/++k;
printf("\n j=%d", j);
}
```

b.
```
#include<stdio.h>
main()
{
        printf("%d", >>10);
```

```
}
```

c.
```
#include<stdio.h>
main()
{
int a, b, c;
a=0;
b=4;
c=5;
a=+c+++b/c;
printf("\n a=%d", a);
}
```

3.9 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma,*Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press, 1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 4	
INPUT / OUTPUT IN 'C'	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

4.1 OBJECTIVE

The main objective of this lesson is to introduce about various input functions in C. Two type of I/O functions are discussed namely formatted I/O functions which manipulate the data in a particular format e.g. print(), scanf() and unformatted I/O functions which cannot control the data in a particular e.g. getch(), getchar() etc. Further classification of unformatted I/O functions is done as string I/O and character I/O.

4.2 INTRODUCTION

The computer will be of no use unless it is able to communicate with the outside world. A computer is

an machine which takes input, processes it and shows output as shown in Figure. 4.2.1. Input/Output devices are required for users to communicate with the computer.

In simple terms, input devices bring information INTO the computer and output devices bring information OUT of a computer system. These input/output devices are also known as peripherals since they surround the CPU and memory of a computer system. In computing, input/output or I/O is the communication between information processing system (such as a computer) and the outside world, possibly a human or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. I/O devices are used by a person (or other system) to communicate with a computer. For instance, a keyboard or a mouse may be an input device for a computer, while monitors and printers are considered output devices for a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

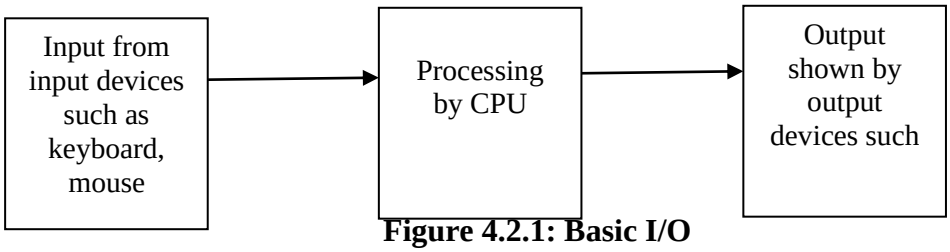


Figure 4.2.1: Basic I/O

Some commonly used Input/Output devices are listed in Table4.2.1.

Table 4.2.1: I/O devices

Input devices	**Output devices**
Keyboard	Monitor
Mouse	LCD
Joystick	Printer
Scanner	Plotter
Light pen	
Touch screen	

The peculiarity about C is that it has no provision for receiving data from input devices nor sending data

to output devices. So, all the input output operations in C are carried out with the help of inbuilt functions, which are defined in standard libraries. In this unit we will C the various types of I/O operations with the help of various programs written in C. We shall also learn about different library functions and header files included in these respective libraries.

4.3 TYPE OF I/O FUNCTIONS IN C

Any language that a programmer use to communicate with computer facilitates inputting the data, processing and outputting the desired result. In any programming language input means to feed some data into program. This can be given in the form of file or from command line.

C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement. On the other hand output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data. C programming language provides a set of built-iin functions to output required data. The various standard I/O functions are written by the designers of C in a library called as "C standard library". A library is just one or more files that contains a group of predefined functions such as printf(), scanf() etc.. This library get installs itself whenever you install C language in computer. These standard library functions can be accessed from library by different methods depending upon programming language. Whenever these function are defined in any program, compiler search for those functions in standard library. A file must be linked with the program whenever a function is used in program. In C, these files are of extension .h for e.g. stdio.h for standard I/O functions, math.h for standard maths function are included while writing program. These files are also called as header files. Recall the pre-processor directive statement:

#include < stdio.h >

#include directive tells the compiler to link stdio.h header file with the program. All the I/O functions such as printf(), scanf(), getchar(), puts() etc are defined in this header file. An I/O function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed in parenthesis. The argument represents data items that are sent to the function. Some input output does not require arguments, though the empty parenthesis must still appear.

The standard I/O library functions are categorized into three broad categories as shown in Figure.4.3.1:

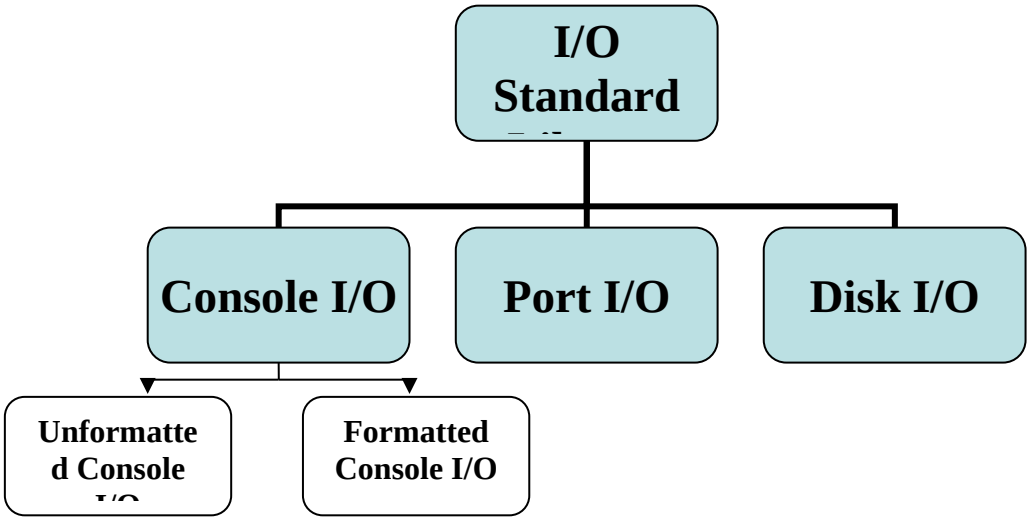


Figure.4.3.1: Standard Library I/O functions in C

1) **Console I/O-** Console refers to the standard input and output devices. Console I/O functions refers to the operations that occur at the keyboard and screen of your computer. These functions accept input from the keyboard and produce output to the monitor screen.

   C takes all input and output as stream of characters. A stream is nothing, but a series of bytes. Whether a program gets input from the keyboard, a file, C treats every stream as a stream of characters. In C there are three stream associated with console I/O operations-

   - Stdin- It is the stream which supplies the data from input source (i.e. keyboard) to program.
   - Stdout- It is the stream which is used to receive output of a program and supply it to the standard output device (i.e. monitor).
   - Stderr- it is the stream used to keep error messages separate from output of a program. i.e. standard error generally points to your terminal screen.

Further, console I/O is divided into:

a) Formatted console I/O- formatted I/O function accept or present data in a particular format. The standard C library consists of two I/O functions, printf() and scanf(). printf() function is used for outputting and scanf() function is used for inputting the data. These functions can format the information under the user's direction. It is highly desirable that the output is presented in such a way

that they are understandable and is in form easy to use. The printf() statement provides certain features through which the screen output is effectively controlled. scanf() allows formatted reading of data from the keyboard. scanf() function can read a character, string, numeric data from keyboard. The general format, examples and features of formatted console I/O functions are discussed in detail upcoming topics.

b) Unformatted console I/O- this function cannot control the format of reading and writing the data. All the unformatted console I/O functions are defined in stdio.h file. There are two types of unformatted console I/O functions:

- Character I/O- These are the functions which can handle I/O of one character at a time.

  These functions deal with the individual character value. For inputting following functions are used, getchar(), getch(), getche(). For outputting we use, putchar(), putch().

  The basic differences between these I/O functions are discussed in detail with examples in upcoming topics.

- String I/O- These functions can handle I/O of whole string. gets() function is used for inputting and puts() function for outputting the data. Detail study of these functions is done in upcoming topics.

2) **Port I/O**- This type functions perform I/O operations among various codes like printer port, mouse port etc. the detailed study of port I/O function is beyond the scope of the book.

3) **Disk I/O**- The function which performs secondary storage devices like floppy disk or hard disk is called disk I/O functions. We know that files are located in different secondary storage generally on disk. So, disk I/O is just file handling functions in C which we will read in detail in upcoming lessons.

4.3.1 Unformatted Console I/O Function

As mentioned earlier, unformatted console I/O function doesn't allow input and output to be formatted according to user requirements. There are several standard library functions under this category- those that can deal with single character and those that can deal with string of characters. So, unformatted console I/O functions are further classified as shown in Table 4.3.1.1.

- Character I/O functions

- String I/O functions

Table 4.3.1.1: Classification of Unformatted I/O functions

Unformatted Console I/O			
Character I/O functions		**String I/O functions**	
Input functions	**Output functions**	**Input functions**	**Output functions**
getchar() getch() getche()	putchar() putch()	gets()	puts()

Let us discuss all these I/O functions with the help of programs in detail. The functions that process one character at a time is called as character I/O functions. These functions deal with the individual character value.

Single character input: getchar()

The most basic way of reading input is by calling the function getchar. Getchar() reads one character from standard input which is user's keyboard. Getchar() returns the character it reads or if there is no more character available, the special value EOF(end of file). The syntax of getchar() function is

character variable = getchar (void);

where character variable refers to previously declared character variable. The word void indicates that no argument is needed for calling the function. The function does not require any argument except void. For e.g.

```
char a;
a=getchar();
```

In this code, the function getchar() reads a single character from keyboard and assigns to character variable 'a'.

If an end of file condition is encountered when reading a character with the getchar() function, the value of symbolic constant EOF is automatically returned. This value if EOF will be assigned within the

stdio.h file. Typically the EOF will be assigned the value -1(value assigned to EOF may vary compiler to compiler). The detection of EOF in this manner offers a convenient way to detect the end of file, whenever and wherever it may occur. getchar() function is also used to read multi-character string by reading one character at a time within a multipass loop.

Single character output: putchar()

Single character can be displayed using the C library function putchar(). This function is complementary to getchar(). The putchar() function like getchar() is part of standard C I/O library. It transmits single character to the standard output device, monitor. The character being transmitted will be represented as a character type variable. It must be expressed as an argument to the function, enclosed in parenthesis, following the word putchar. Syntax of putchar function is

putchar (character variable)

where character variable refers to some previously declared character variable. e.g.

Program - Write a program to print 26 alphabets in capitals using putchar().

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

    for (c='A'; c<='Z'; c++)
    putchar(c)
    return 0;
}
```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

In C putchar() can display string in various ways-

- Taking character as argument

putchar(‘a’)

In this statement single character ‘a’ is given as argument to putchar () function. This statement will display character ‘a’ on monitor.

- Taking variable as argument

putchar(a)

In this statement ‘a’ is variable of type character declared previously in the program. This statement will display character stored in character variable a. e.g.

char a;

a=’X’;

putchar(a);

This code will output X on monitor.

- Displaying particular character from array

putchar (a[0]);

This statement will display $a[0]^{th}$ element of array character array a[ ]. Like accessing individual array element, characters can be displayed one by one using putchar(). The following program shows the usage of getchar() and putchar() together.

Program - Write a program to display a single character entered by user.

```
#include <stdio.h>
int main()
{
    char c;
    printf (“ Enter Character: ”);
    c = getchar ( );
    printf (“ Charcater Entered: ”);
    putchar ( c );
    return 0;
```

```
}
```

Output:

Enter Character: m

Character Entered: m

In the above program, the statement c=getchar( ); accepts a character entered by user, in this case and stores in character variable c , initialized by statement char c;. The statement putchar( ); prints the character stored in variable c.

Program - Write a program to display string of characters entered by user.

```
#include < stdio.h >
int main ( )
{
        int i;
        char ch;
        for ( i=1; i<=10; ++i )
        ch = getchar( );
        putchar ( );
        return 0;
}
```

Output:

AaBbCcDdEe

This program reads ten characters, one for each iteration of the for loop from the keyboard as entered by user. getchar() gets a single character from user and stores in character variable ch. putchar() displays stored single character to monitor

Although getchar() and putchar() appear to handle characters, they are actually manipulating the ASCII

values for the characters. getchar() reads a char and converts it into ASCII (American Standard Code for Information Interchange), and putchar() converts an ASCII value into a char. Technically, when typing inputs in the keyboard , their ASCII values are stored in a place called buffer. Every character on the keyboard had some ASCII value for e.g. ASCII code for character 'a' is 97 and that of new line character '\n' is 10. The getchar() function takes the input from this buffer only not directly from the keyboard. When the getchar() function is called it goes and get a value from the buffer. If the buffer is empty it waits for some time until the buffer being loaded.

The following program uses ASCII code for displaying the lower case characters on monitor.

Program - Write a program to print all lower case characters using their ASCII codes.

```
#include < stdio.h >
void main( )
{
        int i;
        for (i=97;  i<=122;  i++)
                putchar(i);
 }
```

Abcdefghijklmnopqrstuvwxyz

ASCII code for 'a' is 97 and of 'A' is 65. As the difference of 'a' and 'A' is 32 (97-65=32), program can also be written to convert lower case character to upper case and vice-versa as shown in following program.

Program - Write a program to convert lower case to upper case

```
#include < stdio.h >
void main ( )
{
        char ch;
        printf(" Enter any character in lower case: ");
```

```
    ch=getch( );
    printf("\n Upper case character of typed character is: ")
    putchar (ch-32 );
}
```

Output:

Enter any character in lower case: a

Upper case character of typed character is: A

Functions can also be written directly passing ASCII values of characters and special symbols as arguments as shown in following program.

Program - Write a program to illustrate passing of ASCII values as arguments.

```
#include < stdio.h >
void main ( )
{
    char c1=120, c2='\n';          /*ASCII code for 'x' is 120*/
    putchar(c1);
    putchar(c2);
    putchar('y');
    putchar('\n');
    putchar(122);                  /*ASCII code for 'z' is 122*/
}
```

Output:

x

y

z

Single character input without echo: getch( )

getch( ) also serve the same purpose as getchar( ) i.e. getch( ) is used to accept a character from the keyboard. Although getch() also has the same syntax as getchar():

character variable = getchar (void);

The main difference between getchar( ) and getch( ) is that function getch() gets a character from console but does not echo to the screen whereas getchar( ) does. getch() reads a single character directly from the keyboard, without echoing to the screen. getch( ) function prompts the user to press a character and that character is not printed on screen, getch( ) header file is conio.h.

Program - Write a program to illustrate getch() in C.

```
#include <stdio.h>
#include <conio.h>
void  main()
{
  int c;
  clrscr();
  printf("Press any key\n");
  c = getch();
  if (c)
    printf(" A key is pressed from keyboard ");
  else
    printf("An error occurred ");
}
```

Output:

Press any key

A key is pressed from keyboard

For the first getch( ) function, when we press any key from keyboard we'll see the next line but won't see the key which is pressed.

The functionality of getch( ) for not showing the pressed character is used as a screen stopper or to hold the output window until any key is hit from the keyboard. We can also say that it is used to temporarily halt the execution of program until we hit any character from keyboard. Basically this is the most general use of getch( ) function. You will see getch( ) function used in the last statement of many programs in C. the following program will show how getch( ) is used to hold the output screen.

Program - Write a program to illustrate getch() in C as screen stopper or holding the output window.

```
#include<stdio.h>
#include<conio.h>
 void main()
{
   printf("Press any character from the keyboard to exit.\n");
   getch();
}
```

Output:

Press any character from the keyboard to exit.

The program will exit only when you press any valid character from keyboard, until then the output screen will show "Press any character from the keyboard to exit." only. As user hits the key, output screen disappears.

Single character output: putch( )

putch**( )** also serves the same purpose as putchar( ). putch( ) is also used to print a character on monitor. The syntax for putch( ) is:

putch(character variable)

The functioning of putch( ) is very much similar to putchar( ) except that header file for putch( ) is conio.h.

Single character input with echo: getche( )

We can also use getch**( )** for receiving a character from the keyboard without echoing on screen. getche(

) function prompts the user to press a character and that character is printed on screen. This is the basic difference between getch( ) and getche( ). getche ( ) echoes character to the screen without the entered key being pressed. Syntax for getche( ) is:

character variable=getche(void)

When this function is executed, the character entered by the user is displayed on the screen.

Program - Write a program to illustrate getche( ) function in C.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
char c;
printf("Enter any character:");
c=getche();
printf("The character u entered=%c", c );
getch();
}
```

Output:

Enter any character: z

The character u entered=z

getch() is a function which holds your screen until you press any key whereas getche() is a function which holds the screen and waits for the correct key so in other way you can say that it waits for character, read it and then return to program.

String I/O function

The disadvantage of character I/O function is that it accepts only single character; they cannot handle more than one character at a time. On the other hand string I/O functions can handle more than one character at a time. They can handle strings which are sequence of characters. Strings are implemented

as arrays of type char. So you declare an array like:

char string[30];

This will hold a string of up to 29 characters in length.

All strings of characters are terminated with the null character '\0'. The null character marks the end of the string, allowing the different C functions to know where the end of the string is. All characters following the '\0' are ignored.

Strings can be initialized with either a brace enclosed character list, or a string constant.

e.g. char str[4] = {'U', 'S', 'A', '\0'};

char str[20] = "Initial value";

Two functions that can be used for handling I/O strings are:

- gets( )
- puts( )

gets() accepts any line of string including spaces from the standard Input device (Keyboard). gets() stops reading character from keyboard only when the enter key is pressed. This function "gets" reads a whole line of input from standard input (stdin) into a string until a newline or EOF (End-of-File) is encountered. Syntax for gets ( ) is:

gets(character array)

where character array is C variable previously declare as an array of characters. Consider an e.g.

char str[20];

gets(str);

Array str[ ] of type char size 20 is declared. gets( ) is used to receive user input to the array. gets( ) stops receiving user input only when the Newline character (Enter Key) is interrupted. This character array str can store a string of 19 valid characters, 20th space is reserved for null character (‘\0’) with which a string is always terminated as shown in Figure.4.3.1.1.

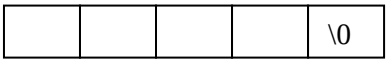


Figure 4.3.1.1: Representation of Array Containing String

The function gets( ) will accept a string of maximum 19 characters and will store in memory address as pointed by str. The length of string is limited by the declaration of string. As the enter key is pressed, a null character is automatically added in the end of the string and is indication of that the input of string is completed.

puts( ) function writes sequence of characters or string on monitor. The puts() function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with puts() is on its own line. puts() returns non-negative on success, or EOF on failure. Syntax for puts( ) is:

puts(character array);

where character array refers to some previously declared array of char type. Consider an e.g.

puts(str);

This statement will display a string on the monitor whatever is stored inside str and causes the cursor to move in next line. puts( ) also accepts string literals as shown below:

puts("This book is related to C programming language");

White spaces are also allowed in puts. Both the functions gets( ) and puts( ) returns EOF if an error occurs. The following program is example of how we can use both gets( ) and puts( ) in a single program.

Program - Write a program to illustrate gets( ) and puts( ).

```
#include<stdio.h>
#include<conio.h>
void  main()
{
        char data[100];
        printf("Enter a String:");
         gets(data);
        printf("Entered Data Is :");
        puts(data);
```

getch();

}

Output:

Enter a String: Programming

Entered Data Is: Programming

4.3.2 Formatted I/O Function

Formatted I/O consists of functions that allow input output operations to be done in some formatted manner. By formatting the data we mean that:

- Alignment of data: whether data is left aligned or right aligned?
- How various kinds of data i.e. integer, float, character, string is handled by I/O functions?
- How to display the fractional data?
- How much field width is required in between data to be displayed on monitor?

All these kind of formatting is done by two formatted I/O functions: printf( ) and scanf( ). printf( ) is used to display the formatted data items on standard output device i.e. monitor whereas scanf( ) is used to read the formatted data input from standard input device i.e. keyboard. Both these I/O functions are somehow different from I/O functions which we have previously discussed. Both of these functions are defined in header file stdio.h, so it is necessary to include stdio.h file in the program where we are goin to use either of these function. Both of these functions return EOF, in case error occurs.

printf( )

We have use printf( ) function in almost every program in C. it is one of the most useful function to display the any data on monitor. printf( ) function simply prints the data written inside the inverted commas (“ “). For e.g. printf(“this book is related to programming in C”) will simply print the text in between” “ on screen. But simply displaying the text data is not the use of printf( ) function. User may need to display numeric data, fraction data and other kinds of data on the screen. The printf function allows numeric and character data to be printed in almost any desired format. Some of the formatting done under printf( ) function is shown below:

printf("Hello, you."); prints Hello, you.

printf("6 times 9 is %d", 6*9);	prints	6 times 9 is 54
printf("%d times %d is %d", a, b, a*b);	prints	something like 12 times 102 is 1224
printf("The %dth of %s", 22, "March");	prints	The 22th of March
printf(s);	prints	the string s, unless it contains a %
printf("one\ttwo\tthree");	prints	one two three

The outputs of all these codes will be discussed in upcoming sections. If an input failure occurs before any conversion, EOF is returned. Otherwise, the number of successful conversions is returned: this may be zero if no conversions are performed. An input failure is caused by reading EOF or reaching the end of the input string (as appropriate). A conversion failure is caused by a failure to match the proper pattern for a particular conversion.

Let us now discuss the basic format of printf( ), which is as follows:

printf (“Format control string”, List of expressions);

where:

- Format control string is the layout of what's being printed and is enclosed in “ ”.
- List of expressions: It is a comma-separated list of variables or expressions yielding results to be inserted into the output. These expressions may be constants, variables or other complex expressions whose values are formatted and printed according to the specifications of format string.

For precise output formatting, every printf( ) call contains a format control string that describes the output format.

The format control string consists of:

- Escape sequence
- Literal characters
- Conversion specifiers
- Flags
- Field widths

- Precisions

Below is the format for format control string of printf( ) function:

%< flags><field width><precision><length>conversion

This format control string controls the format of displayed data. The meaning of flags, field width, precision, length, and conversion are discussed in upcoming sections.

Together with percent sign (%), these, form conversion specifications. Function printf() can perform the following formatting capabilities by using various components of format control string:

- Rounding floating-point values to an indicated number of decimal places.
- Aligning a column of numbers with decimal points appearing one above the other.
- Right-justification and left-justification of outputs.
- Inserting literal characters at precise locations in a line of output.
- Representing floating-point number in exponential format.
- Representing unsigned integers in octal and hexadecimal format.
- Displaying all types of data with fixed-size field widths and precisions.

Now, let us discuss the various components of format control string with the programs

Literal Text

The format string consists of multiple characters. Except for double quotes(“ ”), escape sequences(\), and conversion specifiers(%d,%f..), all characters within pair of double quotes will be treated as literal text and it will be displayed as it is on screen. For e.g.

printf(“\nHello\n World”)

After running this code following output is displayed on screen:

Hello

World

Here Hello World is considered to be literal text.

Escape sequence

Any character that is prefix with a backslash (\) is considered to be as escape sequence. Various control characters, such as newline and tab, must be represented by escape sequences.

An escape sequence is represented by a backslash (\) followed by a particular escape character.

Table 4.3.2.1 summarizes all the escape sequences and the actions they cause.

Table 4.3.2.1: Escape Sequences

Escape sequence	Description
\ ’	Output the single quote (’) character.
\ ”	Output the double quote (") character.
\ ?	Output the question mark (?) character.
\ \	Output the backslash (\) character.
\ a	Cause an audible (bell) or visual alert.
\ b	Move the cursor back one position on the current line.
\ f	Move the cursor to the start of the next logical page.
\ n	Move the cursor to the beginning of the next line.
\ r	Move the cursor to the beginning of the current line.
\ t	Move the cursor to the next horizontal tab position.
\ v	Move the cursor to the next vertical tab position.

The following program shows the usage of some of common escape sequences.

Program - Write a program to illustrate escape sequences.

```
#include< stdio.h >

void main( )
```

{

printf("\n This book is related to \"C programming\".\nThis is an example of printf.\nThat\'s would be all.\n\"Thank You\" ");

}

Output:

This book is related to "C programming".

This is an example of printf.

That's would be all.

"Thank You"

Conversion Specifiers

A conversion specifier always begin with percentage (%) sign and ends with conversion specifier. A conversion specifier is a symbol that is used as a placeholder in a formatting string. For integer output (for example), %d is the specifier that holds the place for integers The following table summarizes all the format specifiers that can be used in printf( ).

Table 4.3.2.2 Integer specifiers

Conversion Specifier	Description
%d	Display a signed decimal integer
%i	Display a signed decimal integer.
%o	Display an unsigned octal integer.
%u	Display an unsigned decimal integer.
%X or %x	Display an unsigned decimal integer. X causes the digit 0-9 and the letters A-F to be displayed, and x causes the digits 0-9 and a–f to be displayed.
%h or %l	Place before any integer conversion specifier to indicate that a short or long integer is displayed respectively.

Program - Write a program to use integer conversion specifiers.

```
#include <stdio.h>
int main()
{
    printf("Various format for integer printing\n");
    printf("-------------------------------------\n");
    printf("%d\n",  455);
    printf("%i\n",  455);  //i same as d in printf()
    printf("%d\n",  +455);
    printf("%d\n",  -455);
    printf("%hd\n",  32000);
    printf("%ld\n",  2000000000L);
    printf("%o\n",  455);
    printf("%u\n",  455);
   printf("%x\n",  455);
  printf("%X\n",  455);
 return 0;
}
```

Output:

```
Various format for integer printing
-------------------------------------
455
455
455
-455
```

32000

2000000000

707

455

455

65081

1c7

1C7

Table 4.3.2.3: Floating point specifier.

Conversion specifier	Description
%e or %E	Display a floating-point value in exponential notation.
%f	Display floating-point values.
%g or %G	Display a floating-point value in either the floating-point form f or the exponential form e (or E)
%l	Place before any floating-point conversion specifier to indicate that a long double floating-point value is displayed.

Program - Write a program to print floating-point numbers with floating-point conversion specifiers

```
#include <stdio.h>
 void main()
{
     printf("Printing floating-point numbers with floating-point conversion specifiers.");
     printf("1.  %e\n", 1234567.89);
    printf("2.  %e\n", +1234567.89);
    printf("3.  %e\n", -1234567.89);
```

```
   printf("4.  %E\n", 1234567.89);
   printf("5.  %f\n", 1234567.89);
  printf("6.  %g\n", 1234567.89);
  printf("7.  %G\n", 1234567.89);
}
```

Output:

Printing floating-point numbers with floating-point conversion specifiers.

1. *1.234568e+06*
2. *1.234568e+06*
3. *-1.234568e+06*
4. *1.234568E+06*
5. *1.234567.890000*
6. *1.23457e+06*
7. *1.23457E+06*

Table 4.3.2.4: String and character specifier.

Conversion specifier	Description
%c	Print data as a single character
%s	Print data as a string

Program - Write a program to print strings and characters

```
#include <stdio.h>
int main()
{
   char character = 'A';
   char string[ ]  =  "This is a string";
```

```
    char *stringPtr = "This is also a string";
    printf("---------------------------------\n");
    printf("---Character and String format---\n");
    printf("---------------------------------\n\n");
    printf("%c <--This one is character\n", character);
    printf("\nLateral string\n");
    printf("%s\n", "This is a string");
    printf("\nUsing array name, the pointer to the first array's element\n");
    printf("%s\n", string);
    printf("\nUsing pointer, pointing to the first character of string\n");
    printf("%s\n", stringPtr);
    return 0;
}
```

Output:

```
---------------------------------
---Character and String format---
---------------------------------
A<--This one is character

Lateral string
This is a lateral string

Using array name, the pointer to the first array's element
This is a string

Using pointer, pointing to the first character of string
This is also a string
```

Table 4.3.2.5: Other conversion Specifiers

Conversion specifier	Description
%p	Display a pointer value (memory address) in an implementation defined manner.
%n	Store the number of characters already output in the current printf() statement. Nothing is displayed
%%	Display the percent character.

Program - Write a program to illustrate %p, %n, and %% conversion specifiers.

```
#include <stdio.h>
 void main()
{
    int  *ptr;          // pointer variable
    int  x  =  12345, y;
    ptr = &x;           // assigning address of variable x to variable ptr
    printf("The value of pointer ptr is %p\n", ptr);
    printf("The address of variable x is %p\n\n", &x);
    printf("Total characters printed on this line is:%n", &y);
    printf(" %d\n\n", y);
    y = printf("This line has 28 characters\n");
    printf("%d characters were printed\n\n", y);
    printf("Printing a %% in a format control string\n");
}
```

Output:

The value of pointer ptr is 162B:0FFA

The address of variable x is 162B:0FFA

Total characters printed on this line is: 41

This line has 28 characters

28 characters were printed

Printing a % in a format control string

Field width specifier

A field width determines the exact size of a field in which data is printed. If the field width is larger than the data being printed, the data will normally be right-justified within that field.An integer representing the field width is inserted between the percent sign (%) and the conversion specifier in the conversion specification. Function printf() also provides the ability to specify the precision with which data is printed. Precision has different meanings for different data types. The format for field width specifer is:

%w conversion character

Where w is the minimum field width for the output.

The following is a program example.

Program - Write a program to print integers right-justified

```
#include <stdio.h>
void main()
{
   printf("%4d\n",  1);
   printf("%4d\n",  12);
   printf("%4d\n",  123);
   printf("%4d\n",  1234);
   printf("%4d\n\n",  12345);
```

```
    printf("%4d\n",   -1);
    printf("%4d\n",   -12);
    printf("%4d\n",   -123);
    printf("%4d\n",   -1234);
    printf("%4d\n",   -12345);
    return 0;
}
```

Output:

```
   1
  12
 123
1234
12345

   -1
  -12
 -123
-1234
-12345
```

Program - Write a program using precision while printing integers, floating-point numbers and strings

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

```
    int     i = 873;
    float   f =  123.94536;
    chars[ ] = "Happy Birthday";
    printf("Using precision for integers\n");
    printf("\t%.4d\n\t%.9d\n\n", i, i);
    printf("Using precision for floating-point numbers\n");
    printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
    printf("Using precision for strings\n");
    printf("\t%.11s\n", s);
}
```

Output:

```
Using precision for integers
        0873
        000000873
Using precision for floating-point numbers
        123.945
        1.239e+02
        124
Using precision for strings
       Happy Birth
```

Flags

Flags used to supplement its output formatting capabilities. Five flags are available to be used in format control string. A flag must be placed immediately after the percent(%) sign. The format for using flag in format control string is:

%flg conversion character

Where flg be any of the 5 flags. There may be or may not be minimum field width specifier appear before the conversion character. Table 4.3.2.6 shows the five flags available with their description.

Table 4.3.2.6: Flags

Flag	Description
- (minus)	Left-justify the output within the specified field
+ (plus)	Display a plus sign preceding positive values and a minus sign preceding negative values.
Space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion specifier o. Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X. Force a decimal points for a floating-point number printed with e, E, f, g, or G that does not contain a fractional part. (Normally the decimal point is only printed if a digit follows it). For g and G specifiers, trailing zeros are not eliminated.
0(zero)	Pad a field with leading zeros

Consider the following codes for using flags in various ways and their outputs:

```
//right justifying and left justifying values
printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
```

Output:

```
     hello         7         a    1.230000

hello         7         a         1.230000
```

```
// printing numbers with and without the + flag
printf("%d\n%d\n", 786, -786);
printf("%+d\n%+d\n", 786, -786);
```

Output:

```
786
-786
+786
-786
```

```
// printing a space before signed values not preceded by + or –
printf("% d\n% d\n", 877,  -877);
```

output:

```
 877
-877
```

```
// printing with the 0 (zero) flag fills in leading zeros
printf("%+09d\n", 762);
printf("%09d", 762);
```

output:

```
+00000762
000000762
```

scanf( )

scanf( ), referred as formatted input function, the complement of printf( ), can be used to read the different type of data from the keyboard in specified format. Like printf( ), scanf( ) also uses a format string to describe the format of the input, but with some little variations:

- It does not allow escape sequences in the format string.

- It requires a special operator '&' called as "address of", which is to be prefix with the variable identifier.

General format for scanf( ) is:

scanf(format_string, list_of_variable_addresses);

- The format string is like that of printf. The format string is used to control interpretation of a stream of input data, which generally contains values to be assigned to the objects pointed to by the remaining arguments to scanf.
- But instead of expressions, we need space to store incoming data, hence the list of variable addresses
- If x is a variable, then the expression &x means "address of x"

The contents of the format string may contain:

- White space: The input stream to be read up to the next non-white-space character.
- Ordinary character: Anything except white-space or % characters. The next character in the input stream must match this character. This is text that must be matched character by character with the input.
- Conversion specification: This is a % character, followed by an optional * character (which suppresses the conversion), followed by an optional nonzero decimal integer specifying the maximum field width, an optional h, l or L to control the length of the conversion and finally a non-optional conversion specifier. Note that use of h, l, or L will affect the type of pointer which must be used.

scanf( ) reads data from user and stores it somewhere in the memory, then writes it on the standard output. Opposite to the printf() write formatted data to the standard output, scanf() read formatted data from the standard input, the keyboard. The scanf() function reads data from the standard input stream stdin and writes the data into the location given by argument. Each argument must be a pointer to a variable of a type that corresponds to a type specifier in format.

If copying takes place between strings that overlap, the behavior is undefined. Consider an e.g. for scanf( ):

Program - Write a program to receive user input from keyboard using scanf()

```
#include <stdio.h>
int main()
{
 int i;
 float f;
 char c;
 printf("Enter an integer and a float, then Y or N\n> ");
 scanf("%d%f%c", &i, &f, &c);
 printf("You entered:\n");
 printf("i = %d, f = %f, c = %c\n", i, f, c);
}
```

On running this program following output is shown:

```
Enter an integer and a float, then Y or N
> 34 45.6Y
You entered:
i = 34, f = 45.600, c = Y
```

In this program, compiler receives an integer, float and character from user and stores in the variables i, f and c respectively as indicated by code scanf("%d%f%c", &i, &f, &c);. Once the values has been stored in variables, it can be used anywhere for any purpose.

White spaces

A white-space character causes scanf() to read, but not store, all consecutive white-space characters in the input up to the next non–white-space character. We can use Blank (' '); tab ('\t'); or newline ('\n') as white spaces. Consider the statement

scanf("%d %d", &a, &b);

During runtime you can enter integer values of a and b in any of these ways:

100 200 // single space between values

100 200 // tab between values

100 // new line between values

200

In the next segment we are going to discuss the use of scanf( ) with character and string values. Consider the statements:

char ch;

printf("Enter any character:");

scanf("%c",&ch);

printf("\nThe typed character is %c", ch);

Output:

Enter any character: a

The typed character is a

To understand receiving of string by scanf( ), consider the code:

char name[10];

printf("\nEnter your name:");

scanf("%s", name); //& is not required before name/*

printf("\nyour name is %s",name);

Output:

Enter your name: Ashok

your name is Ashok

Note the difference between two codes, while receiving string using scanf( ), ampersand(&) is not required before name array. This is because name array is of type char and the first index is the address

of the first element of the array. The format specifier %s cause the scanf( )to read character until a white space is encountered . Then a null character is automatically appended to the array, which is the indication of end of string.

To perform the additional formatting of data, the format string of scanf( ) is provided with some additional features such as maximum field width, asterisk(*) sign.

Maximum field width

We previously studied that all continous non white spaces is considered as single value for scanf( ). However it is possible to assign a limited number of characters to a particular variable from the set of non continuous non white space characters. Consider the statement:

scanf("%d %d",&a, &b);

what if we provide 1234567 as input to scanf( ). Clearly this value will be stored in variable a until scanf( ) finds some white space. But if we want to assign values of specific width for a and b, from this single input, without considering white space, in that case, maximum field width specifiers are used. The format for width specifier is:

%w conversion character

Where w is integer number which specifies the characters to read and conversion character may be any character. For e.g.

scanf("%3d %4d",&a, &b);

Now if we provide 1234567 as input, then 123 is assigned to a and 4567 is assigned to b.

But what if the input values contains lesser character than specified field width or if the data contains more characters then the specified field width in scanf( ). Consider the following codes and you will able to clearly understand:

- Input: scanf("%3d %4d",&a, &b);

  User values: 12345 6789

  Output: a=123 b=45
- Input: scanf("%3d %3d %3d",&a, &b, &c);

  User values: 123456789

Output: a=123 b=456 c=789

- Input: scanf("%3d %3d %3d",&a, &b, &c);

  User values: 1234 5678 9

  Output: a=123 b=4 c=567

- Input: scanf("%2d %3f %c", &a, &b, &c);

  User values: 12 1.23 a

  Output: a=123 b=1.2 c=3

- Input: scanf("%4s %c", a, &b);

  User values: HELLOX

  Output: a=HELL b=O

Asterisk sign (*)

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored. Sometimes there is a situation when we don't want to assign the particular data value to the concerned variable in scanf( ), so in order to suppress the particular assignment into variable, asterisk sign(*) called as suppression character can be prefixed to the conversion character. The * indicates that the field is to be read but ignored, so that i.e. scanf("%*d %d", &i) for the input "12 34" will ignore 12 and read 34 into the integer i. This statement will read two inputs for i, but will store only second input into i.

printf( ) vs puts( )

Both printf( ) and puts( ) have their own advantages and disadvantages. Following are the few points of difference between two input functions.

Advantages of using printf( ):

- printf( ) can print any kind of data whereas puts( ) do not have this ability.
- printf( ) has the ability to format the output according to the user requirements whereas puts( ) has ability to output only in particular format.

- printf( ) can be used to print multiple values in a single statement of printf( ) whereas puts( ) has no ability to print multiple values.
- printf( ) is more widely used than puts( ).

Advantages of using puts ( ):

- syntax of puts( ) is very simple whereas printf( ) have somewhat complex format.
- puts ( ) is more faster in processing than printf( ).
- puts( ) ends with a new line character whereas printf( ) do not have this ability.

Consider a program to differentiate between printf( ) and puts( ):

Program - Write a program to differentiate between printf( ) and puts( )

```
#include <stdio.h>
void main( )
{
        printf(" printf( ) will print: ");
        printf("how");
        printf("are");
        printf("you");
        printf("\n");
        puts(" puts( ) will print:");
        puts("how");
        puts("are");
        puts("you");
}
```

Output:

```
printf( ) will print: how are you
puts( ) will print:
```

how

are

you

scanf( ) vs gets( )

Advantages of scanf( ) over gets( ):

- scanf( ) can read any type of data value from keyboard.
- scanf( ) has capability to read the input in user defined way.
- scanf( ) can accept multiple values within a single scanf( ) statement.

Advantages of gets( ) over scanf( ):

- gets( ) has more simple syntax as compares to scanf( ).
- gets( ) has more faster processing than scanf( ).
- gets( ) continuously reads the characters from the keyboard until enter key is read, spaces and tabs are accepted in the string, which is not in case for scanf( ).

Following program will help in differentiating between two functions.

Program - Write a program to differentiate between scanf() and gets().

```
#include <stdio.h>
void main( )
{
    char name1[20], name2[20];
    printf("\n Enter your first name and second name: ");
    scanf("%s", name1);
    printf("\n Re-enter your first name and second name: ");
    gets(name2);
    printf("\n Output for scanf( ): %s", name1);
    printf("\n Output for gets( ): %s", name2);
```

}

Output:

Enter your first name and second name: Dennis Ritchie

Re-enter your first name and second name: Dennis Ritchie

Output for scanf( ): Dennis

Output for gets( ): Dennis Ritchie

As we seen in the above program that scanf( ) do not accepts the string including space, tab or any other user defined character whereas gets( ) does. So in order to overcome this problem, a special conversion specifier i.e. […] can be used in place of %s to read the string form keyboard. The format for using […] is:

scanf("%[character]", variable name);

where character may be single character or group of characters that is acceptable by scanf( ). If any other character other than specified in […] is used then it will cause the string to be terminated and appends with null (\0) character or end of string.

Program - Write a program to illustrate […].

```
#include <stdio.h>
void main( )
{
    char name[100];
    printf("\nEnter your first name and second name in capital letters: ");
    scanf("%[ABCDEFGHIJKLMNOPQRSTUVWXYZ]", name);
    printf("Entered name is: %s", name);
}
```

Output:

Enter your first name and second name in capital letters: DENNIS RITCHIE

Entered name is: DENNIS RITCHIE

Enter your first name and second name in capital letters: DENNIS Ritchie

Entered name is: DENNIS R

When user inputs name as DENNIS Ritchie, then scanf( ) will read upto R. there is another conversion specifier [^] which performs the opposite function of […]. It means whatever character follows a cirumplex (^), will not be acceptable in the input string. If any of these characters which follows circumplex (^) is entered in input string will cause the string to be terminated and append with null (\0) character. For e.g. consider a statement:

scanf("%[^ABCDEFGHIJKLMNOPQRSTUVWXYZ]", name);

Now whenever any capital character is input then the string will be terminated.

Input: denniS

Output: denni

In order to accept full string of characters in a single line, [^\n] is used. Consider the following Program -

```
#include <stdio.h>
void main( )
{
        char name[100];
        printf("\nEnter your first name and second name in capital letters: ");
        scanf("%[^\n]", name);
        printf("\nEntered name is: %s", name);
}
```

Output:

Enter your first name and second name in capital letters: Dennis Ritchie

Entered name is: Dennis Ritchie

In this output, even small letters are also accepted as only new line (\n) character will terminate the string.

4.4 DIFFERENCE BETWEEN getch (), getche(), getchar()

- The two functions getch() and getche() are very similar, as they respond without pressing the Enter key. The difference is that with the function getche() the echo of the pressed key is displayed on the screen (the letter "e" stands for "echo"), but with the function getch(), there is no echoing.
- Function Header File

  getch() - conio.h

  getche() - conio.h

  getchar() - stdio.h

  The work of all the function written above is to take input of a single character.
- The work of all the function written above is to take input of a single character.
- First function getch(), this function takes a input of a single character but the character is not displayed n the screen when we press any key on Console Window. That is why we use this function in the last of our program so that it holds the output till any key is pressed from the keyboard.

  The second function getche() also takes input of a single character but the character is displayed on the screen when we press key on console Window.

  The third function getchar() also takes input of a single character but you have to pres carriage return (Enter) key after typing the character. In the case you have type more than one character only the first character will be taken as a input the rest of the character will be ignored.

Now consider this program the difference will be cleared.

Program - Write a program to differentiate between getch(), getche() and getchar().

```
#include<conio.h>
#include<stdio.h.> /*forgetchar() function*/
int main()
{
```

```
char ch;
printf("Enter a character :")
scanf("%c",ch);
printf("You have entered : %c",ch);
printf("\nEnter a character :");
ch=getch();
printf("\nYou have entered :%c",ch);
printf("\nEnter a character :");
ch=getche();
printf("\nYou have entered :%c",ch);
printf("\nEnter a character :");
ch=getchar();
printf("\nYou have entered :%c",ch);
}
```

Output

Enter a character: a

You have entered: a

Enter a character:

You have entered: b

Enter a character: c

You have entered: c

Enter a character: d

You have entered: d

Firstly we have taken a single character as input with the help of scanf(). The we have print it, simply

with the help of printf(). Now, with the help of getch() you'll see that the input character is now shown on console window. Now proceeding further to getche() you'll notice that the input character is also displayed on the console window. Now at the last while using getchar() we required to press Enter key to give input from console window.

4.5 SUMMARY

In this lesson we have studied various type of unformatted and formatted I/O functions and their further classification. We can conclude that C has vast range of I/O functions to deal with reading and writing user data. Unformatted I/O functions works only with character datatype (char). The unformatted input functions used in C are getch(0, getche(),getchar(), gets(). The formatted Input/Output functions can read and write values of all data types, provided the appropriate conversion symbol is used to identify datatype. scanf is used for receiving formatted Input and printf is used for displaying formatted output.

4.6 KEYWORDS

Standard library – A group of inbuilt functions stored in a single file as unit.

Console I/O- Console refers to the standard input and output devices. Console I/O functions refer to the operation that occur at the keyboard and screen of your computer.

getchar()- This functions get a single character from keyword and echoes it on the screen.

putchar()- This function displays a single character on screen. It is complementary to getchar(). The character being transmitted will be represented as a character type variable.

getch()- this function gets a character from console but does not echo to the screen

putch()- putch() is also used to print a character on monitor. Header file is conio.h.

gets()- gets() continuously reads the character from the keyboard until enter key is read, spaces and tabs are accepted in the string.

puts()- puts() function writes sequence of character or string on monitor. The puts() function automatically inserts a newline character at end of each string it displays.

printf()- It is one of the most useful function to display the any data on monitor.

scanf()- The complement of printf(), can be used to read the different type of data from the keyboard in specified format.

4.7 REVIEW QUESTIONS

1. What do you mean by input output functions? How do we classify I/O functions in C. Explain?

2. What do you mean by header files? What is header file for printf(), scanf(), getch(), getchar(), getche(), gets()?

3. Write the outputs fro following codes:

3.1 printf("Hello, world!\n");

printf("greeting, Earthing\n\n');

3.2 int numStudents = 35123;

printf("GJU has %d students", numStudents);

3.3 printf("GJU has %10d students", numStudents);

3.4 printf("GJU has %-10d students", numStudents);

3.5 double cost = 123.45;

printf("You total id $%f today\n", cost);

3.6 printf("You total is $%e today\n", cost);

3.7 printf("You total is $%9.2f today\n", cost);

3.8 printf("%s%10s%-10sEND\n", Hello", "Alice", "Bob");

4 Write a program to convert days to months and months to days.

5 Write a program to requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

6 Write a program that reads a character from keywords and then prints it in reverse case is given in That is, if input is upper case, the output will be lower case and vice versa.

7 Write a program to print an decimal integer into its equivalent octal and hexadecimal.

4.8 FURTHER READINGS

[1] E.Balaguruswamy, Introduction to C, Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, The C Programming Language, Prentice Hall, 1988.

[3] R.Hutchison, Programming in C, Tata McGraw Hill, 1990.

[4] R..Johnsonbaugh and M.Kalin, Applications Programming in C, MacMillian, 1990

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 5	
DECESION MAKING CONTROL	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

5.1 OBJECTIVE

The objective of this lesson is to learn decision and control making in C. C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution. C provides various key condition statements to check condition and execute statements according conditional criteria. Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program. After going through this lesson students will

learn about various decision making statements used in C for e.g. if, if-else, nested if else etc.

5.2 INTRODUCTION

When a program is run, the CPU begins execution at the top of main (), executes some number of statements, and then terminates at the end of main (). The sequence of statements that the CPU executes is called the program's path. Most of the programs you have seen so far have been straight-line programs. Straight-line programs have sequential flow — that is, they take the same path (execute the same statements) every time they are run (even if the user input changes). However, often this is not what we desire. For example, if we ask the user to make a selection, and the user enters an invalid choice, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program. To write a realistic program doesn't mean that collection of statements arranged in a particular sequence. There are some situations in the program where user have to take decision whether to run particular statement or not or the execution of some statements may depend upon some condition provided by user. There may also be condition where user wants to run some statements again and again with defined number of times as decided by user. We can handle these situations by writing straight line programs but

that will be too lengthy and clumsy to write such programs for e.g. to enter marks of a student in different subjects we have to write number of statements for entering the marks. But C has some special provisions to handle these kinds of situations.

In this unit we will discuss various kinds of control statements like sequential, selection and iteration in detail and how to implement these flow statements in C program. Some jump statements are also discussed.

5.3 STATEMENTS

In some situations we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. For these situations C provides control statements. In other words, control flow statements are statements by which we can control the flow of program execution according to user needs i.e. if the programmer wants to specify its own way of the order by which the statements are executed than it has to make use of control flow statements. There are four types of control flow statements in C:

- Decision and selection making statements
- Iteration statements
- Jump statements

Each one of these types of statements is discussed below.

1.) Decision statements in C-: Decision statements always test a condition and then accordingly decide the order of execution of statements. Sometimes a programmer wants to execute a set of instructions in one situation and might be another set of instructions in another situation. In this type of scenarios, we should use decision statements that will help us to execute properly in this scenarios. The different types of decision control statements are listed in Figure. 5.3.1.

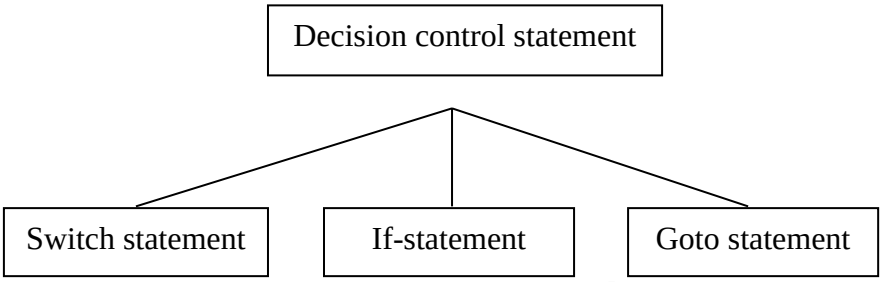


Figure 5.3.1: Decision Control Statements

In decision statements two or more sets of statements are written but only one of these sets is executed according to condition as shown in the Figure.5.3.2:

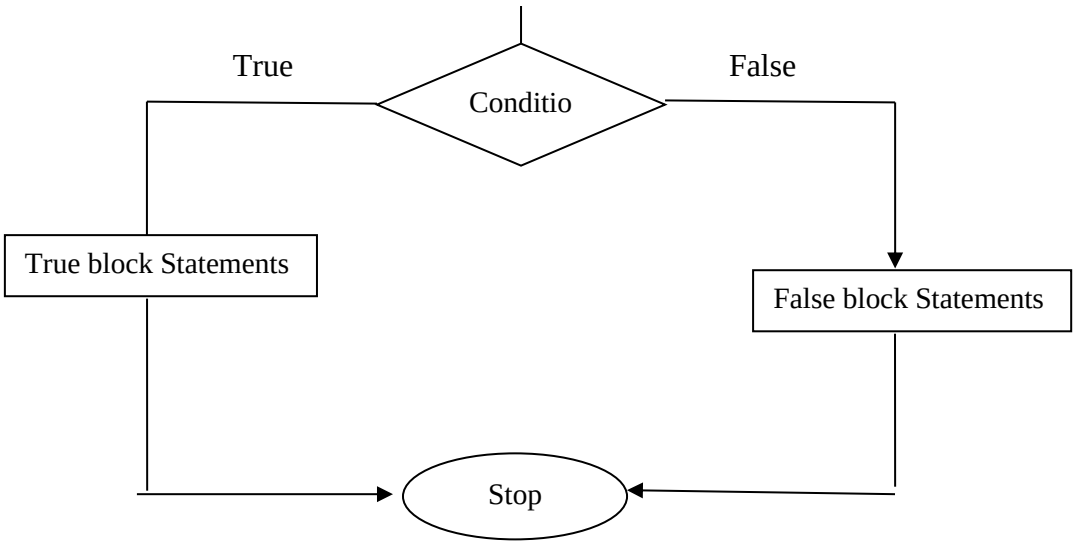


Figure 5.3.2: Decision Control Statement with condition

The if statement make use of relational and logical operators for decision making as shown in code below where if marks are greater than or equal to 50 then student is considered as pass and if marks obtained is less than 50 then fail:

```
if (marks >= 50)
        printf ("PASS");
if (marks < 50)
        printf ("FAIL");
```

2.) Looping (Iteration) Statements in C Language: - Sometime we might require executing a certain set of statements repeatedly again and again. In this type of scenario, we need looping statements to operate in these circumstances. For example, we need to print the all the even numbers which are less than 100. In this problem, there is a condition until the number is not greater than 100 we want to repeatedly execute the print statement that will display the even number until it has exceeded 100. Looping statements always are written with a condition. The control of the program keeps on looping (repeatedly execute) until the condition is true. As we saw in the previous example, until the number is 100 we have to keep on displaying all the even numbers. The different types of decision control statements used in C are shown in Figure. 5.3.3:

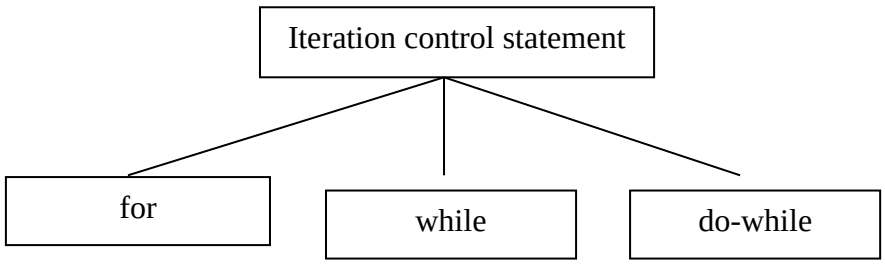


Figure 5.3.3: Looping Decision Making Statements

In looping, set of statements is run whenever condition is true and until condition becomes false as shown in Figure 5.33.4:

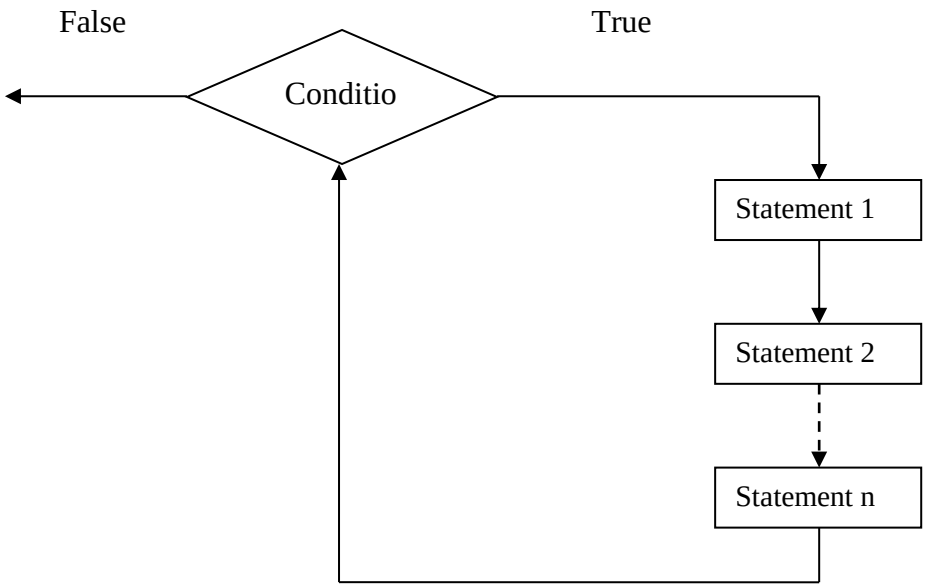


Figure 5.3.4: Structure of Looping Statements

Program iterations are used to process certain program's block, multiple number of times. Number of repeats (iterations) can be, or don't have to be previously known. If test condition before starting iteration then while or for is used and if test condition is meet at the end then do-while is used. For e.g consider the code of displaying numbers from 1 to 5:

.

.

while (i<=5)

printf("%d",&i);

…

3.) **Jump statements:** Branching is performed using jump statements, which cause an immediate transfer of the program control. Jump statements allow you to exit a loop, start the next iteration of a loop, or explicitly transfer program control to a specified location in your program. The different types of decision control statements used in C are shown in Figure 5.3.5:

4.)

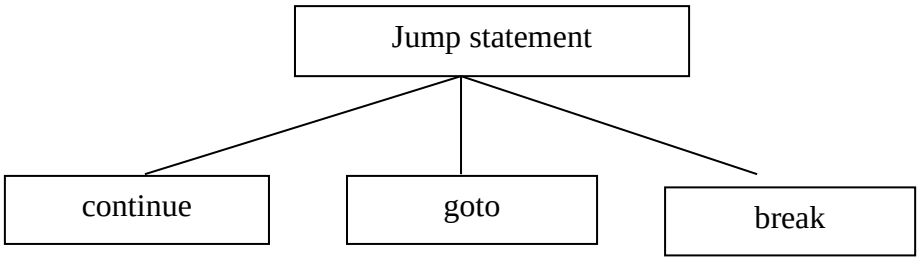


Figure 5.3.5: Jump Decision Control Statement

The jump statement is used to alter the normal sequence of program execution by transferring control to some other part of the program unconditionally as shown in Figure 5.3.6 where control is directly transferred to statement 3 after statement 1:

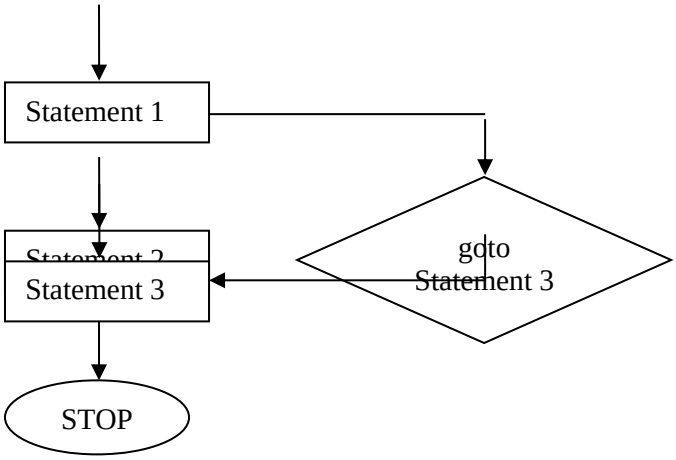


Figure 5.3.6 Structure of Jump Statements

In this chapter we are going to discuss only decision control and selection statement in detail. Looping and jump statements are discussed in detail in upcoming chapters.

Decision statement

The decision statement enables the user to execute the particular set of code. This execution of code depends upon the test condition. If test condition satisfies then code is executed otherwise another set of code is executed. This type of programming gives user the flexibility in execution of program. In C following decision statements are used:

- Simple if statement
- if-else statement
- Nested if else statement

- else if statement

Without a conditional statement, programs would run almost the exact same way every time, always following the same sequence. Selection of these statements depends upon user and complexity of program. Before discussing the decision statements, let us examine the meaning of TRUE and FALSE in computer terminology. A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false. For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1. If this confuses you, try to use a printf statement to output the result of those various comparisons (for example printf ("%d", 2 == 1) ;

When programming, the aim of the program will often require the checking of one value stored by a variable against another value to determine whether one is larger, smaller, or equal to the other. There are a number of operators that allow these checks. Here are the relational operators, as they are known, along with examples:

>	greater than	5 > 4 is TRUE
<	less than	4 < 5 is TRUE
>=	greater than or equal	4 >= 4 is TRUE
<=	less than or equal	3 <= 4 is TRUE
==	equal to	5 == 5 is TRUE
!=	not equal to	5! = 4 is TRUE

Let us discuss each of these decision or conditional statements in detail.

5.3.1 Simple IF statement

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. Syntax for simple if statement is:

```
if (conditional)
{
        block of statements executed if conditional is true
```

}

Test condition must be enclosed in braces ( ). The TRUE block or FALSE block statement may be a single statement, a compound statement or it may also consist of no statement.

Here is a simple example that shows the syntax:

```
if ( marks < 50 )
printf( "FAIL" );
```

Here, we're just evaluating the statement, "if marks is less than 50", to see if it is true or not. If test condition becomes true then FAIL is displayed on user screen and if condition is FALSE then control simply ignore this statement and will pass to next statement. Figure 5.3.1.1 shows flowchart of if statement.

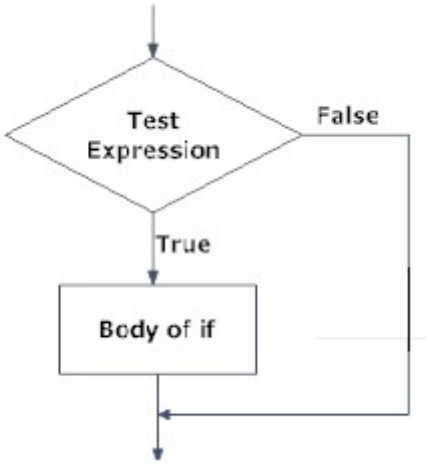


Figure 5.3.1.1 Flowchart of if statement

If the test expression is true then, statements for the body if, i.e., statements inside parenthesis are executed. But, if the test expression is false, the execution of the statements for the body of **if** statements are skipped. Let us consider few examples of if statements.

Program - Write a program to find whether a number is positive or negative using if statement

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

```
{
        int number;
        printf("Enter a number:\n");
        scanf("%d",&number);
        if(number>0)
                printf("given number is positive");
        getch();
}
```

Output:

Enter a number: 9

given number is positive

In this program if number entered by user is greater than 0 then printf statement in if block is executed otherwise if number less than 0 is entered then control will skip the printf statement and nothing will be displayed on user screen.

Consider another example of finding greatest of three numbers using if statement only.

Program - Write a program to find largest number using if statement only

```
#include <stdio.h>
void main()
{
    float a, b, c;
    printf("Enter three numbers: ");
    scanf("%f %f %f", &a, &b, &c);
    if (a>=b && a>=c)
      printf("Largest number = %.2f", a);
```

```
    if (b>=a && b>=c)
      printf("Largest number = %.2f", b);
    If(c>=a && c>=b)
      printf("Largest number = %.2f", c);
}
```

Output:

Enter three numbers: 1.234

3.456

8.912

Largest number = 8.91

Enter three numbers: 78.87

45.56

32.23

Largest number = 78.87

5.3.2 IF ELSE STATEMNET

The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false). The if....else statement is an extension of the simple if statement. Format for if…else is:

```
if (test expression)
        {
  True-block statement(s)
        }
   else{
False-block statement(s)
}
```

Figure 5.3.2.1 shows flowchart of if else statement.

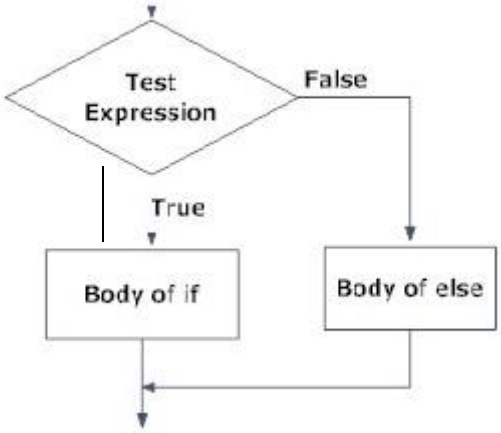


Figure 5.3.2.1: Flowchart of if else statement

If the condition specified in the if statement evaluates to true, the statements inside the if-block are executed and then the control gets transferred to the statement immediately after the if-block. Even if the condition is false and no else-block is present, control gets transferred to the statement immediately after the if-block. In either case, either true-block or false-block will be executed, not both. The else part is required only if a certain sequence of instructions needs to be executed if the condition evaluates to false. It is important to note that the condition is always specified in parentheses and that it is a good practice to enclose the statements in the if blocks or in the else-block in braces, whether it is a single statement or a compound statement. Consider the example for using if…else statement in C.

Program - Write a program to check whether the entered number is positive or negative.

```
#include<stdio.h>
void main( )
{
    int a;
    printf("/n Enter a number:");
    scanf("%d", &a);
    if(a>0)
```

```
    {
        printf( "/n The number %d is positive.",a);
    }
     else
    {
        printf("n The number %d is negative.",a);


    }
}
```

Output:

Enter a number: 9

The number 9 is positive.

Enter a number: -78

The number -78 is negative.

In this program when a positive number is entered then control will check the condition in if block and compare the entered number with 0, if the condition becomes true and number is greater than 0 then printf statement in if block is displayed. But when we entered a negative number then condition in if block fails and control get transferred to else block and printf statement in else block is displayed.

Consider another program of if else statement.

Program - Write a program to find whether an number is odd or even

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

```
    printf("Enter a number you want to check.\n");
    scanf("%d",&num);
    if((num%2)==0)        //checking whether remainder is 0 or not.
        printf("%d is even.",num);
    else
        printf("%d is odd.",num);
}
```

Output:

Enter a number you want to check.

25

25 is odd.

Consider another example of if else statement of finding greatest of three numbers which we have previously discussed above using if statement only.

Program - Write a program to find largest number using if...else statement

```
#include <stdio.h>
void main()
{
    float a, b, c;
    printf("Enter three numbers: ");
    scanf("%f %f %f", &a, &b, &c);
    if (a>=b)
    {
       if(a>=c)
         printf("Largest number = %.2f",a);
```

```
        else
          printf("Largest number = %.2f",c);
    }
    else
    {
        if(b>=c)
          printf("Largest number = %.2f",b);
        else
          printf("Largest number = %.2f",c);
    }
}
```

Output:

Enter three numbers: 1.234

3.456

8.912

Largest number = 8.91

Enter three numbers: 78.87

45.56

32.23

Largest number = 78.87

5.3.3 Nested if...else statement (if...elseif....else Statement)

The if...else statement can be used in nested form when a serious decision are involved. If the test expression is true, it will execute the code before else part but, if it is false, the control of the program jumps to the else part and check test expression 1 and the process continues. If all the test expression are

false then, the last statement is executed. It is a conditional statement which is used when we want to check more than 1 condition at a time in a same program. The conditions are executed from top to bottom checking each condition whether it meets the conditional criteria or not. If it found the condition is true then it executes the block of associated statements of true part else it goes to next condition to execute.

Syntax of nested if...else statement.

```
if(condition_1)
{
 if(condition_2)
 {
   block statement_1;
 }
 else
 {
   block statement_2;
 }
}
else
{
  block statement_3;
}
block statement_4;
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part. Figure 5.3.3.1 shows flowchart of nested if else.

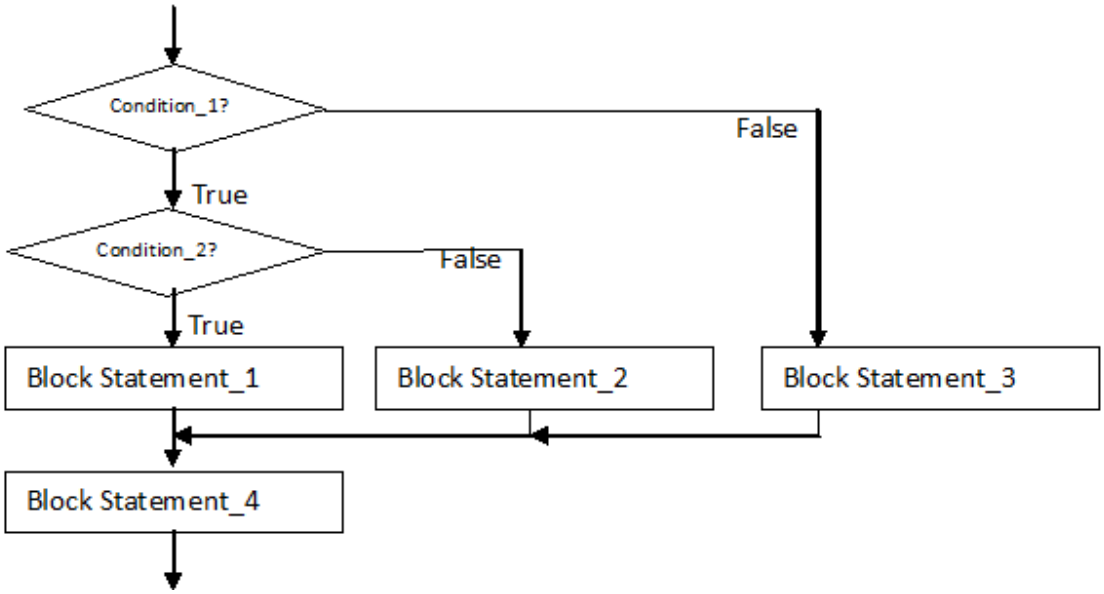


Figure 5.3.3.1: Flowchart of nested if else

Program- Write a program to find largest of two numbers using nested if else

```
#include <stdio.h>
void main(){
    int numb1, numb2;
    printf("Enter two integers to check".\n);
    scanf("%d %d",&numb1,&numb2);
    if(numb1==numb2) //checking whether two integers are equal.
        printf("Result: %d=%d",numb1,numb2);
    else
      if(numb1>numb2) //checking whether numb1 is greater than numb2.
        printf("Result: %d>%d",numb1,numb2);
      else
        printf("Result: %d>%d",numb2,numb1);
}
```

Output:

Enter two integers to check.

5

3

Result: 5>3

Enter two integers to check.

-4

-4

Result: -4=-4

You can set up an if-else statement to test for multiple conditions. The following example uses two conditions so that if the first test fails, we want to perform a second test before deciding what to do:

```
if (x%2==0)
{
    printf("x is an even number");
}
else
{
     if (x>10)
     {
          printf("x is an odd number and greater than 10");
     }
     else
     {
          printf("x is an odd number and less than 10");
     }
 }
```

This brings up the other if-else construct, the if, else if, else. This construct is useful where two or more alternatives are available for selection.

5.3.4 Else if Statement

Another use of else is when there are multiple conditional statements that may all evaluate to true, yet you want only one if statement's body to execute. It is also possible to embed or to nest if-else statements one within the other. Nesting is useful in situations where one of several different courses of action need to be selected. You can use an "else if" statement following an if statement and its body; that way, if the first statement is true, the "else if" will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements to ensure that only one block of code is executed.

The else-if statement is nothing but the rearrangement of else with the if that follows.

Consider the below program.

```
if(condition)
        perform this
else
{
        if(condition)
        perform this;
}
```

The above program can be re-written as,

```
if(condition)
        perform this
else if(condition)
        perform this;
```

The syntax for else if statement is

```
if(condition_1)
        block statement_1;
else if(condition_2)
        block statement_2;
else if(condition_n)
        block statement_n;
.
.
.
else
        default statement;
```

The above is also called the if-else ladder. The various conditions are evaluated one by one starting from top to bottom, on reaching a condition evaluating to true the statement group associated with it are executed and skip other statement. Figure 5.3.4.1 shows flowchart for else if statement is:

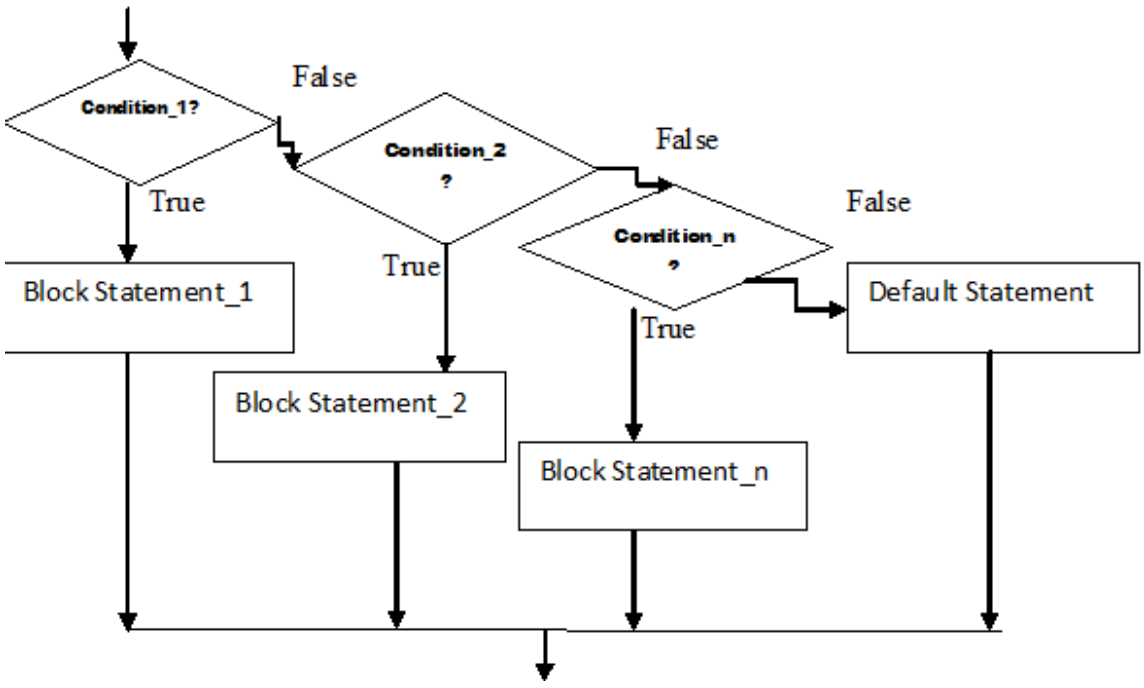


Figure 5.3.4.1: Flowchart of else if statement

During the execution of a nested if-else statement, as soon as a condition is encountered which evaluates to true, the statements associated with that particular if-block will be executed and the remainder of the nested if-else statements will be bypassed. If neither of the conditions are true, either the last else-block is executed or if the else-block is absent, the control gets transferred to the next instruction present immediately after the else-if ladder.

The following program makes use of the nested if-else statement to find the greatest of three numbers.

Program - Write a program to find greater of three numbers using else if

```
#include<stdio.h>
void main( )
{
    int a=6, b=5, c=10;
    if(a>b)
      {
        if(b>c)
         {
           printf("nGreatest is: " , a);
          }
        else if(c>a)
             {
              printf("nGreatest is: ", c);
             }
      }
     else if(b>c)     //outermost if-else block
         {
          printf("nGreatest is: " , b);
          }
```

```
        else
        {
         printf( "nGreatest is: " , c);
        }

}
```

The above program compares three integer quantities, and prints the greatest. The first if statement compares the values of a and b. If a>b is true, program control gets transferred to the if-else statement nested inside the if block, where b is compared to c. If b>c is also true, the value of a is printed; else the value of c and a are compared and if c>a is true, the value of c is printed. After this the rest of the if-else ladder is bypassed.

However, if the first condition a>b is false, the control directly gets transferred to the outermost else-if block, where the value of b is compared with c (as a is not the greatest). If b>c is true the value of b is printed else the value of c is printed. Note the nesting, the use of braces, and the indentation. All this is required to maintain clarity.

Consider another program to find out whether the number entered is positive negative or zero.

Program - Write a program to find if number is positive, negative or zero

```
#include<stdio.h>
void main( )
{
        int  number;
        printf("Type any Number : ");
        scanf("%d", &number);
        if(number > 0)
        {
                printf("%d is the positive number", number);
```

```
    }
    else if(number < 0)
        printf("%d is the Negative number", number);
    else printf("%d is zero", number);
}
```

Output:

Type any Number: 8

8 is the positive number

Type any Number: -8

-8 is the Negative number

Type any Number: 0

0 is zero

Decision making are needed when, the program encounters the situation to choose a particular statement among many statements. If a programmer has to choose one among many alternatives if...else can be used but, this makes programming logic complex. This type of problem can be handled in C programming using switch...case statement. Let us discuss switch case statement in detail.

5.3.5 The Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

The syntax for a switch statement in C programming language is as follows:

```
switch(expression)
{
case value1 :
   body1
   break;
```

```
case value2 :
   body2
   break;

case value3 :
   body3
   break;

default :
   default-body
   break;
}
next-statement;
```

In switch...case, expression is either an integer or a character. Control passes to the statement whose case constant-expression matches the value of switch (expression). If the expression doesn't matches any of the constant in case, then the default statement is executed. The switch statement can include any number of case instances, but no two case constants within the same switch statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a break statement transfers control out of the body.

You can use the break statement to end processing of a particular case within the switch statement and to branch to the end of the switch statement. Without break, the program continues to the next case, executing the statements until a break or the end of the statement is reached. In some situations, this continuation may be desirable. The default statement is executed if no case constant-expression is equal to the value of switch (expression). If the default statement is omitted, and no case match is found, none of the statements in the switch body are executed. There can be at most one default statement. The default statement need not come at the end; it can appear anywhere in the body of the switch statement. A case or default label can only appear inside a switch statement. The type of switch expression and

case constant-expression must be integral. The value of each case constant-expression must be unique within the statement body.

The case and default labels of the switch statement body are significant only in the initial test that determines where execution starts in the statement body. Switch statements can be nested. Any static variables are initialized before executing into any switch statements.

Figure 5.3.5.1 shows flowchart for switch case statement

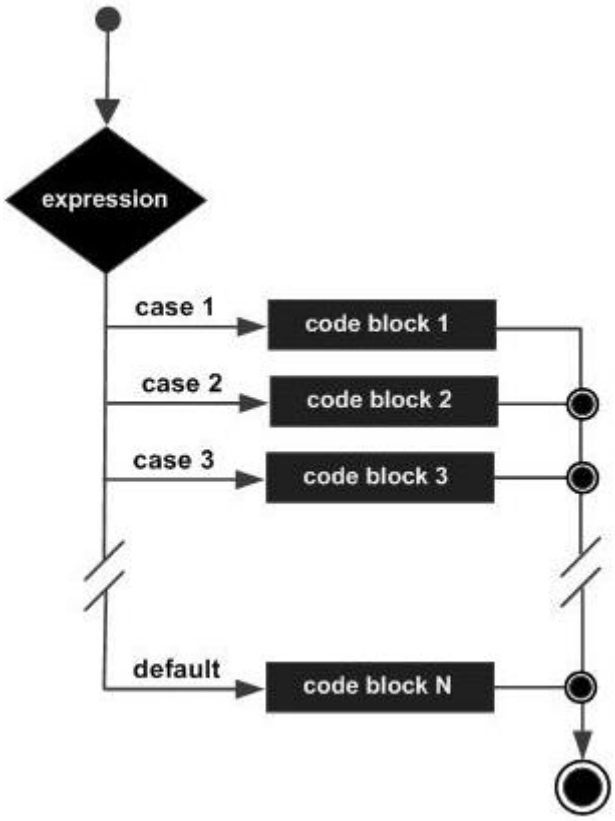


Figure 5.3.5.1: Flowchart of switch statement

Program - Write a program to illustrate switch case statement.

```
#include<stdio.h>

void main()

{

int roll = 3 ;
```

```
switch ( roll )
    {
    case 1:
        printf ( " I am Pankaj ");
        break;
    case 2:
        printf ( " I am Nikhil ");
        break;
    case 3:
        printf ( " I am John ");
        break;
    default :
        printf ( "No student found");
        break;
     }
}
```

3 is assigned to integer variable 'roll', Switch(roll) decides – "We have to execute block of code specified in 3rd case". Switch Case executes code from top to bottom. It will now enter into first

Case [i.e. case 1:] it will validate Case number with variable Roll. If no match found then it will jump to Next Case. When it finds matching case it will execute block of code specified in that case.

Program - Write a Program to create a simple calculator for addition, subtraction, and division.

```
# include <stdio.h>
int main(){
   char operator;
   float num1,num2;
```

```
    printf("Enter operator +, - , * or / :\n");
    operator=getche();
    printf("\nEnter two operands:\n");
    scanf("%f%f",&num1,&num2);
    switch(operator)
    {
    case '+':
            printf("num1+num2=%.2f",num1+num2);
            break;
    case '-':
            printf("num1-num2=%.2f",num1-num2);
            break;
    case '*':
            printf("num1*num2=%.2f",num1*num2);
            break;
    case '/':
            printf("num2/num1=%.2f",num1/num2);
            break;
    default:
            printf(Error! operator is not correct");    /* if operator is other than +, -, * or /, error message
is shown */
    break;
}
 return 0;
}
```

Output:

*Enter operator +, -, * or / :*

/

Enter two operands:

34

3

num2/num1=11.33

Notice break statement at the end of each case, which cause switch...case statement to exit. If break statement are not used, all statements below that case statement are also executed.

Besides the above mentioned usages of switch case statements, various other possibilities also exists. The following segment of code not only provides a few useful tips of switch statement but also some important points that should be taken care while using it.

Here we discuss about all the scenarios of switch that result into compile error. We call them as illegal use of switch case statement; following are the illegal ways of switch case statement in C Programming language

- Floating Case Not Allowed!!!

Switch case label allows only integer value inside case label. Floating points are not at all accepted inside switch case.

```
i = 1.5
switch ( i )
{
  case 1.3:
        printf ("Case 1.3");
        break;
  case 1.4:
```

```
        printf ("Case 1.4");
        break;
  case 1.5:
        printf ("Case 1.5");
        break;
   default :
        printf ("Default Case ");
}
```

- Case Labels Should be Unique

```
i = 1;
switch ( i )
{
case 1:
   printf ("Case 1");
   break;
case 1:
   printf ("Case 2");
   break;
case 3:
   printf ("Case 3");
   break;
default :
   printf ("Default Case ");
}
```

- Case Labels Should not Contain any Variable

Case label must be constant value.

```
i = 1;
switch ( i )
{
case 1:
   printf ("Case 1");
   break;
case i + i :
   printf ("Case 2");
   break;
case 3:
   printf ("Case 3");
   break;
default :
   printf ("Default Case ");
}
```

- All Cases must wrapped in any of the case

(Generally Compiler won't show Error Message but It will neglect statement which does not belongs to any case)

```
i = 1;
switch (i)
{
```

```
printf ( "Hi")              // It does not belongs to any case
case 1:
    printf("Case 1");
    break;
case 2:
    printf("Case 2");
    break;
case 3:
    printf("Case 3");
    break;
default :
    printf("Default Case ");
}
```

Output :

Case 1

- Comparison Not allowed

```
i = 1;
switch(i)
{
case i>1:
   printf ("Case 3");
   break;
default :
   printf ("Default Case ");
}
```

The Conditional Operator [?:] Ternary Operator Statement in C

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

expression 1 ? expression 2 : expression 3

where

- expression1 is Condition
- expression2 is Statement Followed if Condition is True
- expression2 is Statement Followed if Condition is False

Expression1 is nothing but Boolean Condition i.e. it results into either TRUE or FALSE. If result of expression1 is TRUE then expression2 is executed. Expression1 is said to be TRUE if its result is NON-ZERO. If result of expression1 is FALSE then expression3 is executed. Expression1 is said to be FALSE if its result is ZERO. Figure 5.3.5.2 shows conditional operator.

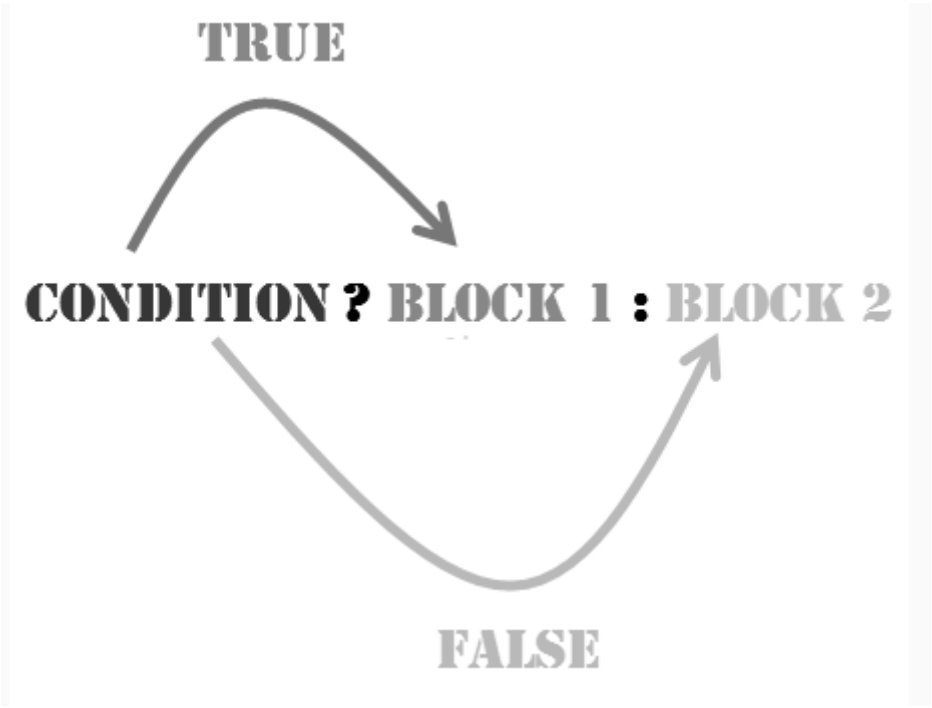


Figure 5.3.5.2: Conditional Operator

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of

```
if (x < y)
{
min = x;
}
else
{
min = y;
}
```

You just say:

```
min = (x < y) ? x : y;
```

Suppose that x and y are integer variables whose values are 100 and 4, respectively. After executing above statement, the value of min is 100.

Here is the simple program to check whether a number is odd or even by using ternary operator.

Program - Write a program to check whether an number is odd or even

```
#include<stdio.h>
void main()
{
int num;
printf("Enter the Number : ");
scanf("%d",&num);
(num%2==0)?printf("No. is Even"):printf("No. is Odd");
}
```

Output:

Enter the Number: 8

No. is Even

Enter the Number: 1

No. is Odd

5.4 SUMMARY

In this lesson we have studied:

- Control flow statements are statements which can control the flow of execution in a program.
- Decision statements always test a condition and then accordingly decide the order of execution of statement e.g. switch, if and goto statement.
- Looping statements are required to execute a certain set of statements repeatedly again and again e.g. for, while and do while statement.
- Branching is performed using jump which cause an immediate transfer of the program control e.g. continue, goto and break statement.
- A simple if and if else can be nested, means it can be placed within another if or if else statement. This is called nesting.
- Switch statement helps to transfer the control of execution based on the situation where expression may have multiple values.
- Conditional operator is ternary operator which evaluates an expression returning a value if that expression is true and different one if the expression is evaluated as false.

5.5 KEYWORDS

Switch Statements – A switch statement allows a variable to be tested for equality a list of value. Each value is called a case, and the variable being switched on is checked for each switch case.

Conditional Operator- A conditional operator in C, is an operator that takes three operands (conditions to be checked), the value when the condition is true and value when the condition is false.

Conditional statements – A statements that evaluates true or false.

If statements - The if statements is a powerful decision making statement and a two –way decision statement which is used in conjunction with an expression.

If-else- The if-else statement is an extension of the simple if statement used to carry out a logical test and then take one of two possible actions depending on the outcome of the test.

Nested if else – Nesting means to embed if-else statements on within the other.

5.6 REVIEW QUESTIONS

1. What do you mean by decision control statements? Explain them with the help of program.

2. Write a short note on following with suitable example:

2.1 if statement

2.2 if else statement

2.3 nested if –else statement

3. What do you mean by switch statement? How it works? Explain with an example.

4. Write a program of an calculator using switch case statement for division, multiplication, addition and subtraction.

5. Write the output of the following codes:

5.1
```
x-12;
If(x<10)
Printf("x is less than 10");
```

5.2
```
marks =56;
If(marks.=40)
        If(marks.=80)
        printf('pass");
else
        printf('fail");
```

6. An electric power distribution company charges its domestic consumer as follows;

Consumption Units	*Rate of charge*
0-200	Rs. 0.50 per unit
201-400	Rs. 100 plus Rs. 0.65 per unit excess of 200
401-600	Rs. 230 plus Rs. 0.80 per unit excess of 400
601 and above	Rs. 390 plus Rs. 1.00 per unit excess of 600

7. Write program that reads the customer number and power consumed and prints the amount to be paid by the customer.

8. Write a program to classify the given inputted number is single, two three and

   Four digit using nested if …. else statement. (Hint: use condition if(n/10000==0), to check four digit no.)

9. It is decided to base the fine for speeding in a built up area as follows – 50 pounds if speed is between 31 and 40 mph, 75 pounds if the speed is between 41 and 50 mph and 100 pounds if he speed is above 50 mph. A programmer writing a program to automate the assessment of fines produces the following statement;

```
If (speed> 30)
        fine = 50;
else if (speed>40)
        fine = 75;
else if (speed>50)
        fine = 100;
```

   Is this correct? What fine would it assign to a speed of 60 mph? IF incorrect how should it be written?

10. Write a program using switch statement to print days of week.

5.7 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to* C, Tata McGraew Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in* C, Tata McGraw Hill, 1990.

[4] R..Johnsonbaugh and M.Kalin, *Applications Programming in* C, MacMillian, 1990

[5] Peter Prinz, Tony Crawford, O'Rielly *C in Nutshell*

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 6	
LOOPING	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

6.1 OBJECTIVES

The main objective of this lesson is to make the students have introduction about the various loop control structure available in C. What is basically looping in C language, there syntax, their use.

How it helps to repeat some portion of the program either a specified number of times or until a particular condition is being satisfied.

6.2 INTRODUCTION

There may be a situation when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Some problems require that a set of statements should be executed a number of times, each time changing the values of one or more variables, so that every new execution is different from the previous one. This kind of repetitive execution of a set of statements in a program is known as a LOOP.

Programming languages provide various control structures that allow for more complicated execution paths. Conditional statements allow you to do something if a given condition is true. However often you want to repeat something while a particular condition is true, or a given number of times.

6.3 LOOP STATEMENT

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages.

We can categorize loop structures into two categories

1) Non deterministic loops –

When the number of times the loop is to be executed is not known then the loop is called Non deterministic loop.

2) Deterministic Loop –

When the number of times the loop is to be executed is known then the loop is called Deterministic loop i.e. the number of executions of set of statements are already known.

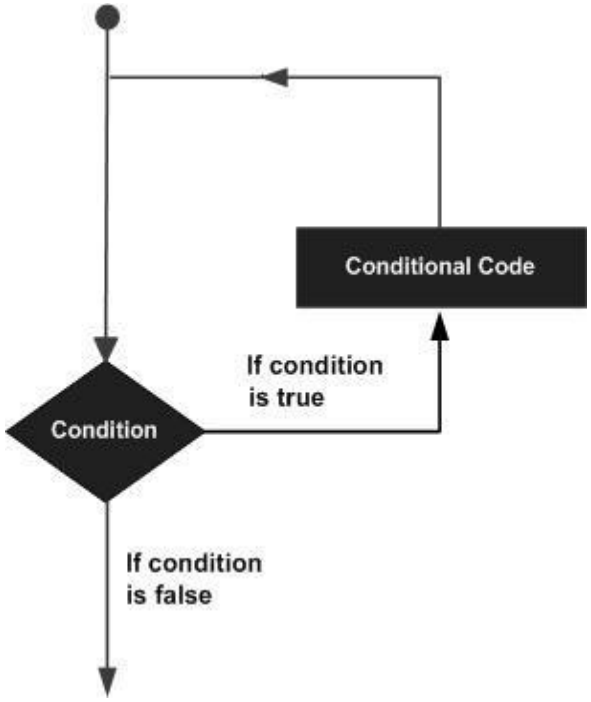


Figure.6.3.1: Looping Flowchart

C has three ways of doing this, depending on precisely what you are trying to do. C supports **while, do while and for** loop constructs to help repetitive execution of a compound statement in a program. The 'while' and 'do while' loops are non deterministic loops and the 'for' loop is a deterministic loop.

6.3.1. The for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. It is a count controlled loop in the sense that the program knows in advance how many times the loop is to be executed.

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

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

Here is the flow of control in a for loop:

1. **for :** for is a reserved word
2. **initialization:** The initialization 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.
3. **condition :**Next, the condition is evaluated. It is a conditional expression required to determine whether the loop should continue or to be terminated. If it is true, the body of the loop is

executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

4. **increment :**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. It modifies the value of the loop control variable by a certain amount.
5. **statement or statements(s):** This can be a simple or a compound statement.
6. 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.

The following example executes 10 times by counting 0…..9.

```
for (i = 0; i < 10; i++)
    …..
```

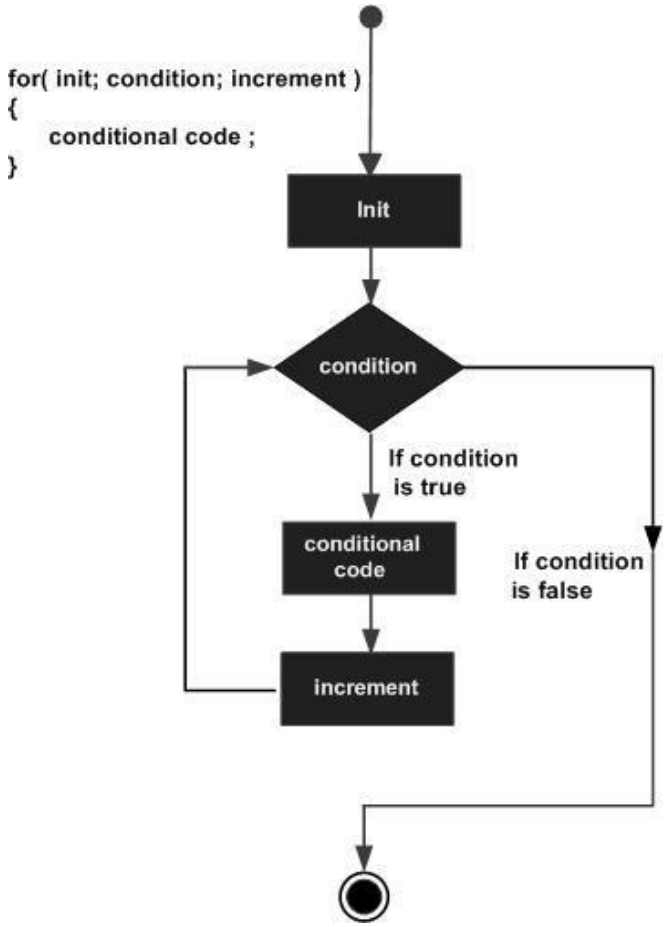


Figure 6.3.1.1: Flowchart of for() statement

Program - Write a program to add n numbers using for loop.

```
#include <stdio.h>
int main()
{
int n, sum = 0, c, value;
printf("Enter the number of integers you want to add\n");
scanf("%d", &n);
printf("Enter %d integers\n",n);
for (c = 1; c <= n; c++)
{
scanf("%d", &value);
sum = sum + value;
}
printf("Sum of entered integers = %d\n",sum);
return 0;
}
```

Output –

```
Enter the number of integers you want to add
4
Enter 4 integers
1
2
3
4
Sum of entered integers = 10
```

Program - Write a program to print numbers from 10 to 19.

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* for loop execution */
for( int a = 10; a < 20; a = a + 1 )
{
printf("value of a: %d\n", a);
}
return 0;
}
```

Output –

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

Note: If the need be, we can leave any of the three expressions out of a for statement, this is an additional feature which is being provided by for loop.

Consider the following for loop

```
for(1=1;i<10;i++)
{
Statements;
}
```

This can also be done as

a) Initialization outside the for statement

```
i=1;
for(;i<10;i++)
{
Statements;
}
```

b) Initialization & Increment outside the for statement

```
i=1;
for(;i<10;)
{
Statements;
i++;
}
```

c) Initialization , Increment & loop condition outside the for statement

```
i=1;
for( ; ; )
{
if(i<10)
break;
```

……..

Statements;

……..

i++;

}

In such a case, the delimiting semicolons must remain as shown above.

Comma operator –

To increase the power of for-lop we can use the comma operator. This operator allows the inclusion of more than one expression in place of a single expression in the for statement as shown below:

for(exp1a,exp1b;exp2;exp3a,exp3b)statements;

Example –

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

The variable *i* and *j* have been initialized to values 1 and 10 respectively. Please note that these initialization expressions are separated by a 'comma'. However, the required semicolon remains as such. During the execution of the loop, *i* increase from 1 to 10 whereas simultaneously *j* decreases from 10 to 1.

Though the power of for loop can be increased by including more than one initialization and increment expressions separated with the comma operator but there can be only one test expression which could be simple or complex.

Program - Write a program that computes the sum of first 10 natural numbers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,s;
/*multiple initialization*/
```

```
for(i=1,s=0;i<=10;i++)
{
s=s+i;
}
printf("\n The sum =%d",s);
}
```

Output –

The sum =55

The modified program after Comma operator in increment expression also is given below

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,s;
/*multiple initialization and increment*/
for(i=1,s=0;i<=10;s=s+i,i++)
{
;
}
printf("\n The sum =%d",s);
}
```

Infact for loop can also be written as

```
for(i=1,s=0;i<=10;s=s+i,i++);
```

Output –

The sum =55

Infinite Loop -

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

Program – Write a program to show the working of Infinite Loop.

```
#include <stdio.h>
int main ()
{
for( ; ; )
{
printf("This loop will run forever.\n");
}
return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing **Ctrl + C** keys.

6.3.2. The while loop –

A while loop statement in C programming language repeatedly executes a target statement as long as a given condition is true. It is the fundamental conditional repetitive control structure in C.

Syntax

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

```
while(condition)
{
```

statement(s);

}

Where

1. **while:** It is a reserved word.
2. **statement(s)**: It may be a single statement or a block of statements.
3. **Condition:** It may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The sequence of operation in a while loop is as follows:

1. Test the condition.
2. If the condition is true, then execute the statement and repeat step1.
3. If the condition is false, leave the loop and go on with the rest of the program.

An equivalent flow diagram for a while loop is given in Figure 6.2.2.1. Here key point of the while loop is that the loop might not ever run. 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.

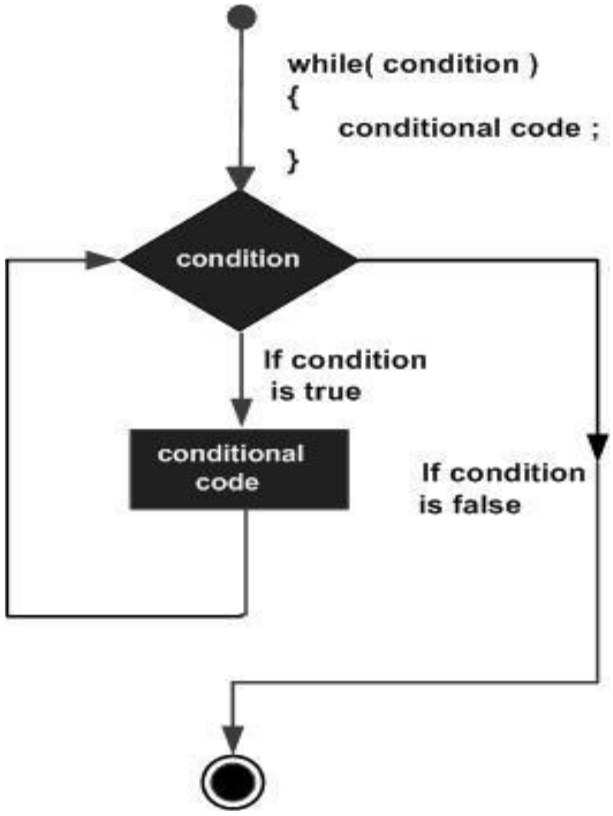


Figure 6.3.2.1: Flowchart of while loop

Example –

```
int num=1,s=0;                    //Find the sum of first 20 natural numbers
while(num<=20)
{
s +=num;
num +=1;
}
```

In this example, the loop num is initialized to 1 before entering the loop. The condition (num<=20), cause the loop to be executed as long as num is less than or equal to 20.The loop variable num is incremented in the body of the loop. The loop is executed 20 times.

Program – Write a program to print number from 10 till 20 using while loop.

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* while loop execution */
while( a < 20 )
{
printf("value of a: %d\n", a);
a++;
}
return 0;
}
```

Output –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Program – Write a program that generates and prints all even numbers from 2 to 20. It also prints their sum.

```
#include <stdio.h>
#include<conio.h>
void  main ()
{
int even=2;
int even_sum=0;
printf("\n The even numbers :");
while(even<=20)
{
printf("%d",even);
even_sum +=even;
even +=2;
}
printf("\n The sum of even numbers =%d",even_sum);
}
```

Output –

The even numbers:

2

4

6

8

10

12

14

16

18

20

The sum of even numbers =110

6.3.3. The do-while loop –

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming language checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. The syntax of a do...while loop in C programming language is:

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

Where

do: It is a reserved word.

statements(s): It can be simple or a compound statement.

while: It is a reserved word.

condition: Is a Boolean expression. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

The sequence of operations in a do-while loop is as follows –

1. Execute the statement.
2. Test the condition.

3. If the condition is true then repeat step 1 to 2.
4. If the condition d false, leave the loop and go on with the rest of the program.

Thus it performs a **post-test** in the sense that the condition is not tested until the body of the loop is executed at least once.

An equivalent flow diagram for a **do-while** loop is given in Figure 6.3.3.1.

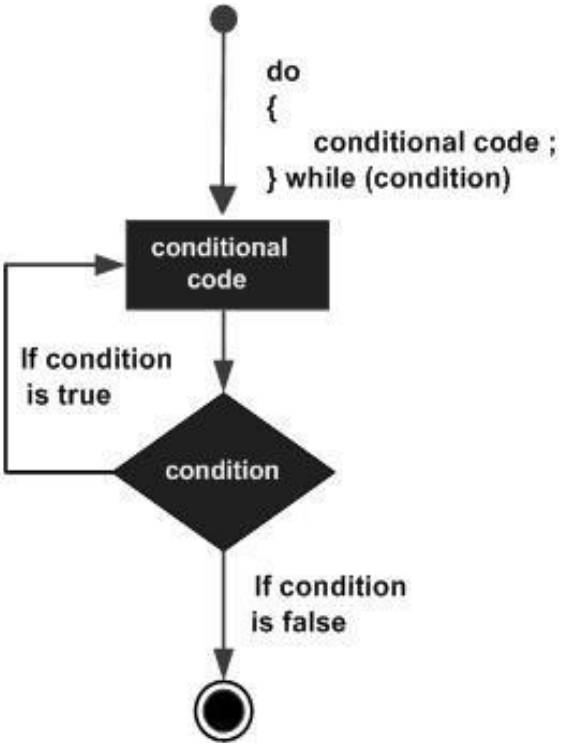


Fig.6.3.3.1: Flowchart of do-while loop

Program – Write a program to print numbers from 10 till 20 using do-while loop.

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* local variable definition */
int a = 10;
```

```
/* do loop execution */
do
{
printf("value of a: %d\n", a);
a = a + 1;
}
while( a < 20 );
return 0;
}
```

Output -

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

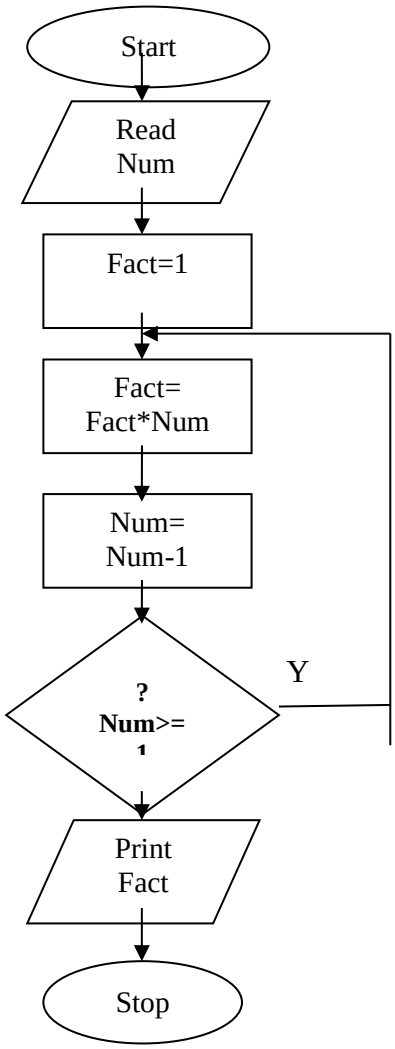
value of a: 19

Program – Write a program to compute factorial of a given positive number.

The factorial N can be computed by the formula using recursion (Detail is explained in Chapter Functions):

N! =N*(N-1)*…….*3*2*1

Flowchart of this problem is given below:



Program - Write a program using while loop to calculate factorial of a number

```
#include <stdio.h>
#include<conio.h>
void  main ()
{
int num;
int fact;
printf("\n Enter a number:");
```

```
scanf("%d",&num);
fact=1;
do
{
fact=fact*num;
num--;
}
while(num>1);
printf("\n The factorial=%d", fact);
}
```

Output –

Enter a number:

5

The factorial=120

Difference between while and do-while loop

WHILE	DO-WHILE
It performs a loop pre-test i.e. the condition is tested before executing the body of the loop.	It performs a loop post-test i.e. the condition is tested after executing the body of the loop.
The statement of loop may not be executed at all i.e. if initially the condition is not satisfied the body of the loop will not get executed even once.	The loop is executed at least once.

6.3.4 Nesting of loops

We can possibly nest one loop construct inside the body of another. The inner and outer loops need not be of the same construct.

1.

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

statement(s);

}
```

2.

```
while ( condition )
{

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

}
```

3.

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

statement(s);

}while(condition);
```

4.

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

while(condition);
}
```

5.

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

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

statement(s);
}
```

Rules for nested loops –

1. An outer for loop and an inner for loop cannot have the same control variable.
2. The inner loop must be completely nested inside the body of the outer loop.
3. Loops should not overlap
4. Jumping out of the loop is allowed butt jumping in is not allowed.

Program – Write a program to print out the following pattern on the screen.

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

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

```
{
int i,j;
printf("\n");
for (i=1;i<=5;i++)
        {
        for(j=1;j<=i;j++)
        printf("%d",j);
        printf("\n");
        }
}
```

Output –

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

Program – Write a program to print a multiplication table in following format:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30

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

```
int N;  /* Number of rows */
printf("\n Enter the number of rows");
scanf("%d",&N);
printf("\n The Multiplication Table….");
for(row=1;row<=N;row++)
{
printf("\n");
for(col=1;col<=10;col++)
printf("%d",row*col);
}
}
```

Output –

Enter the number of rows 3

Multiplication Table….

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30

Program – Write a program using nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* local variable definition */
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;
}
```

Output –

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

6.3.5 The break statement

The break statement in C programming language has following two usages:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

2. It can be used to terminate a case in the switch statement (covered in previous chapter).

If you are using nested loops (i.e. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block. The syntax for a break statement in C is as follows:

break;

where

break: It is a reserved word.

Figure 6.3.5.1 shows flowchart of break statement.

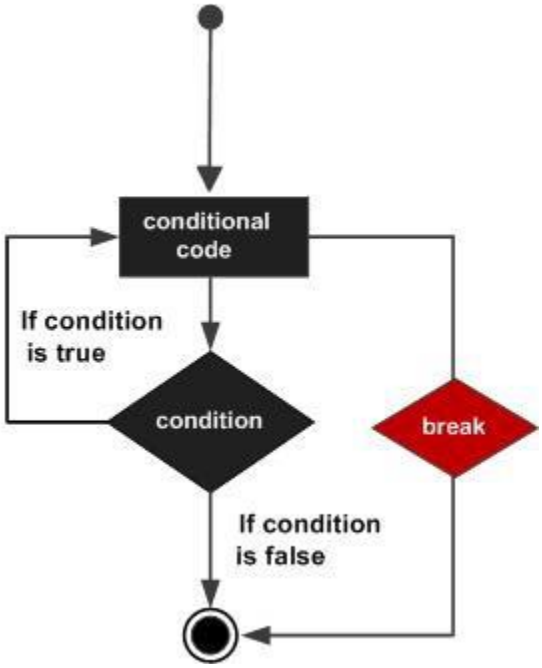


Figure.6.3.5.1: Flowchart of break statement

Program – Write a program to print number from 10 to 20 using while loop but using break statement terminate the execution if number is greater than 15.

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* while loop execution */
while ( a < 20 )
{
printf("value of a: %d\n", a);
a++;
if( a > 15)
```

```
{
/* terminate the loop using break statement */
break;
}
}
return 0;
}
```

Output –

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

6.3.6 The continue statement

The continue statement in C programming language works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the *for* loop, continue statement causes the conditional test and increment portions of the loop to execute. For the *while* and *do...while* loops, *continue* statement causes the program control passes to the conditional tests. The syntax for a continue statement in C is as follows:

continue;

where

continue: It is a reserved word.

Figure 6.3.6.1 shows flowchart of Continue Statement

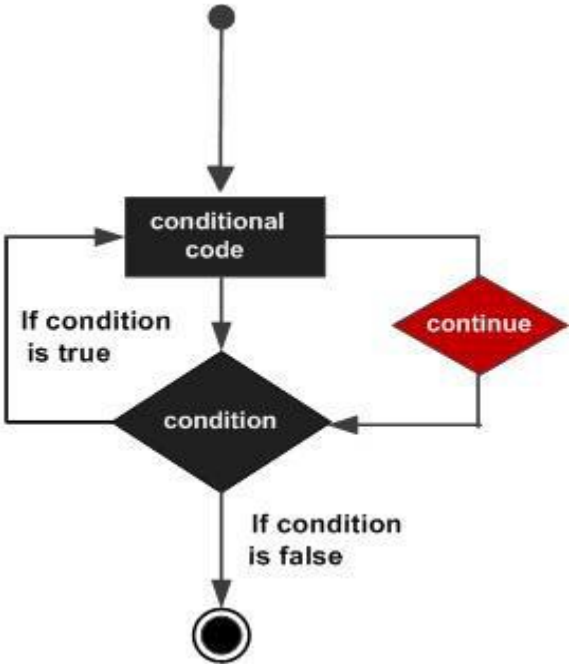


Figure.6.3.6.1: Flowchart of Continue Statement

Program – Write a program to print the number from 10 to 20 using do-while loop and skipping the number 15 using continue statement.

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
do
{
if( a == 15)
{
/* skip the iteration */
a = a + 1;
```

```
continue;
}
printf("value of a: %d\n", a);
a++;
}while( a < 20 );
return 0;
}
```

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

6.3.7 The goto statement (Unconditional Branching)

The unconditional transfer of control means that the sequence of execution will be broken without performing any test and the control will be transferred to some statement other than the immediate next one. The syntax for a **goto statement** in C is as follows:

```
goto label;
..
.
label : statement;
```

where

goto: It is a reserved word.

label: It is an identifier used to label the target statement. Here label can be any plain text except C keyword and it can be set anywhere in the C program above or below to goto statement.

Figure 6.3.7.1 shows flowchart of goto statement.

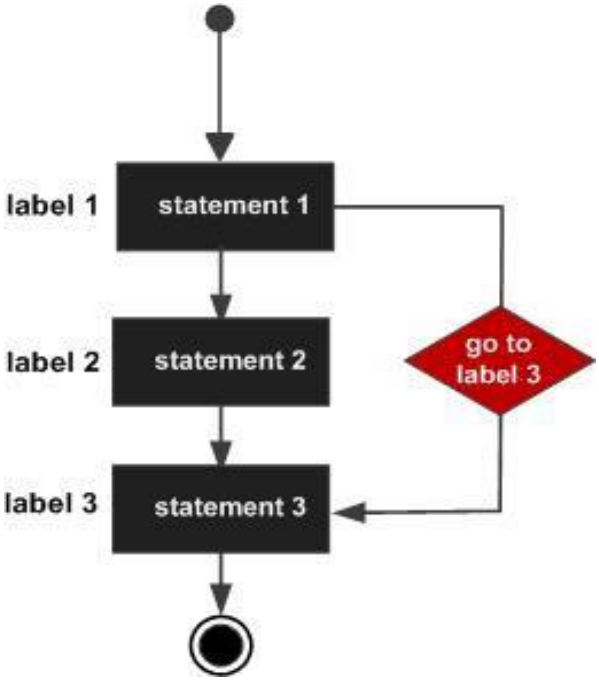


Figure.6.3.7.1: Flowchart of goto statement

Program – Write a program to print the number from 10 to 20 using do-while loop and skipping the number 15 using **goto statement**.

```
#include <stdio.h>
#include<conio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* do loop execution */
```

```
LOOP: do
{
if( a == 15)
{
/* skip the iteration */
a = a + 1;
goto LOOP;
}
printf("value of a: %d\n", a);
a++;
}while( a < 20 );
return 0;
}
```

Output –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Programming Example –

Program - Write a program to convert decimal to binary.

This is a C language code to convert an integer from decimal number system (base-10) to binary number system (base-2). Size of integer is assumed to be 32 bits. We use bitwise operators to perform the desired task. We right shift the original number by 31, 30, 29... 1, 0 bits using a loop and bitwise AND the number obtained with 1(one), if the result is 1 then that bit is 1 otherwise it is 0(zero).

```
#include <stdio.h>
int main()
{
  int n, c, k;
  printf("Enter an integer in decimal number system\n");
  scanf("%d", &n);
  printf("%d in binary number system is:\n", n);
  for (c = 31; c >= 0; c--)
  {
    k = n >> c;
    if (k & 1)
      printf("1");
    else
      printf("0");
  }
  printf("\n");
   return 0;
}
```

Output –

Enter an integer in decimal number system

100

100 in binary number system is:

00000000000000000000000001100100

Program – Write a program to reverse a number.

This program reverse the number entered by the user and then prints the reversed number on the screen. For example if user enter 123 as input then 321 is printed as output.

In our program we use modulus (%) operator to obtain the digits of a number. To invert number look at it and write it from opposite direction or the output of code is a number obtained by writing original number from right to left.

```
#include <stdio.h>
int main()
{
  int n, reverse = 0;
  printf("Enter a number to reverse\n");
  scanf("%d",&n);
  while (n != 0)
  {
    reverse = reverse * 10;
    reverse = reverse + n%10;
    n = n/10;
  }
  printf("Reverse of entered number is = %d\n", reverse);
 return 0;
}
```

Output –

Enter a number to reverse

1234

Reverse of entered number is = 4321

Program – Write a program prints various different patterns of numbers and stars.

These codes illustrate how to create various patterns using c programming. Most of these c programs involve usage of nested loops and space. A pattern of numbers, star or characters is a way of arranging these in some logical manner or they may form a sequence.

```
   *
  ***
 *****
*******
*********
```

```
#include <stdio.h>
int main()
{
  int row, c, n, temp;
  printf("Enter the number of rows in pyramid of stars you wish to see ");
  scanf("%d",&n);

  temp = n;
  for ( row = 1 ; row <= n ; row++ )
  {
    for ( c = 1 ; c < temp ; c++ )
    printf(" ");
    temp--;
```

```
    for ( c = 1 ; c <= 2*row - 1 ; c++ )
    printf("*");
    printf("\n");
  }
  return 0;
}
```

Output –

Enter the number of rows in pyramid of stars you wish to see 9

```
                                        *
                                       ***
                                      *****
                                     *******
                                    *********
                                   ***********
                                  *************
                                 ***************
                                *****************
```

Program - Write a program to print a Diamond pattern in C.

This program prints diamond pattern of stars. Diamond shape is as follows:

```
  *
 ***
*****
 ***
```

```
 *
#include <stdio.h>
int main()
{
 int n, c, k, space = 1;
 printf("Enter number of rows\n");
 scanf("%d", &n);
 space = n - 1;
 for (k = 1; k <= n; k++)
 {
  for (c = 1; c <= space; c++)
  printf(" ");
  space--;
  for (c = 1; c <= 2*k-1; c++)
   printf("*");
  printf("\n");
 }
 space = 1;

 for (k = 1; k <= n - 1; k++)
 {
  for (c = 1; c <= space; c++)
  printf(" ");
  space++;
   for (c = 1 ; c <= 2*(n-k)-1; c++)
```

```
    printf("*");
    printf("\n");
  }
  return 0;
}
```

Output –

Enter number of rows

10

```
                    *
                   ***
                  *****
                 *******
                *********
               ***********
              *************
             ***************
            *****************
           *******************
            *****************
             ***************
              *************
               ***********
                *********
                 *******
                  *****
                   ***
```

*

Program – Write a program to print to Fibonacci series in C programming.

Using the code below you can print as many numbers of terms of series as desired. Numbers of Fibonacci sequence are known as Fibonacci numbers. First few numbers of series are 0, 1, 1, 2, 3, 5, 8 etc, Except first two terms in sequence every other term is the sum of two previous terms, For example 8 = 3 + 5 (addition of 3, 5). This sequence has many applications in mathematics and Computer Science.

```
/* Fibonacci Series c language */
#include<stdio.h>
int main()
{
  int n, first = 0, second = 1, next, c;
  printf("Enter the number of terms\n");
  scanf("%d",&n);
  printf("First %d terms of Fibonacci series are :-\n", n);
  for ( c = 0 ; c < n ; c++ )
  {
    if ( c <= 1 )
      next = c;
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d\n",next);
```

```
  }
  return 0;
}
```

Output –

Enter the number of terms

5

First 5 terms of Fibonacci series are:-

0

1

1

2

3

Program – Write a program to check whether a number is armstrong or not.

This is as C programming code to check whether a number is armstrong or not. A number is armstrong if the sum of cubes of individual digits of a number is equal to the number itself. For example 371 is an armstrong number as $3^3 + 7^3 + 1^3 = 371$. Some other armstrong numbers are: 0, 1, 153, 370, 407.

```
#include <stdio.h>
int main()
{
  int number, sum = 0, temp, remainder;
  printf("Enter an integer\n");
  scanf("%d",&number);
  temp = number;
  while( temp != 0 )
```

```
{
 remainder = temp%10;
  sum = sum + remainder*remainder*remainder;
 temp = temp/10;
}
if ( number == sum )
printf("Entered number is an armstrong number.\n");
else
printf("Entered number is not an armstrong number.\n");
return 0;
}
```

Output –

Enter an integer

371

Entered number is an armstrong number.

6.4 SUMMARY

- Repetitive operation (looping) is achieved in C through a **while** or a **do-while** or a **for** statement.
- Repetitive structures consist of

  (1) Entry point & Initialization,

  (2)Body of the repletion (Loop),

  (3) Exit point.
- There are 2 types of repetitive structures: Condition controlled (while and do-while loop) & Counter Controlled (for loop).
- **While** loop is often used when the number of times the loop has to be executed is not known in advance.

- The **while** loop executes as long as the conditional expression is true. The body of the while loop is a compound statement which is repeatedly executed and again conditional expression is evaluated.
- We use **for** loop in place of while when both start and end conditions are known and after every iteration of the loop there is an increment step.
- We observe that **for** loop is preferable to the **while** loop whenever we know the start condition and stop condition of the loop and the incrementing condition in advance.
- The comma operator can be used to write two expressions in place a single expression in **for** loop. This allows one to set loop on two indices simultaneously.
- The **do-while** loop checks the condition at the bottom after execution of the body and therefore executes the body at least once. The difference with the while loop is that while loop tests the condition at the top before entering the body. The while will not execute the body if the condition is false at the outset.
- The **break** statement transfers the control outside the loop to the first statement following the body of the loop. It only exits from the current loop in case there are nested loops.
- The **continue** statement provides for partial skipping of loop without exiting the loop. The portion of the loop following the **continue**; statement is skipped and the next iteration of the loop begins.
- The continue statement does not terminate the execution of the loop like the break.

6.5 KEYWORDS

- **Loop:** It is a control structure that repeats a group of steps in a program.
- **Loop body:** the statements that are repeated in the loop
- **Counter-controlled loop (Counting loop)** : It is a loop whose required number of iterations can be determined before loop execution begins
- **Infinite loop**: A loop that executes forever.
- **For** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

- **Comma Operator** This operator allows the inclusion of more than one expression in place of a single expression in the for statement.
- **While** loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.
- **Do-while** loop: The loop which checks its condition at the bottom of the loop.
- **Break Statement: W**hen we want to jump out of a loop instantly, without waiting to get back to the conditional test then we use the 'break' keyword. When 'break' encountered inside any loop, control automatically passes to the first statement after the loop. A 'break' is usually associated with an 'if'.
- **Continue Statement** :

  At the time of making the program when we want to take the control to the beginning of the loop, bypassing the statement inside the loop which has not been executed then the keyword 'continue' allows us to do so. A continue statement is usually associated with an if.

6.6 REVIEW QUESTIONS

1. Explain the difference between the following.
   a. while and do while
   b. for loop and while loop
   c. break and continue
   d. simple statement and compound statement
   e. if statement and while statement
   f. If statement and switch statement
2. What is meant by looping? Describe different forms of looping with examples.
3. What is the purpose of while statement? Explain with examples.
4. What is the purpose of do-while statement? How it is differ from while statement? Explain with examples.
5. Explain Break and Continue statements with the help of suitable examples.

6. Write a program to print the following outputs.

i.
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

ii.
```
* * * * *
  * * * *
    * * *
      * *
        *
```

7. Write a program to check whether a given number is prime or not.
8. Write a program t find the H.C.F and L.C.M. of two given numbers.

6.7 FURTHER READING

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma,*Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press,1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 7	
ARRAY AND STRINGS	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

7.1 OBJECTIVE

The objective of this lesson is to learn the concept of arrays and string in C. After studying this unit, you will be able to:

- Declare, Initialize, and Use of Arrays
- Use of Loops for Array Traversal
- Pass Arrays as Parameters

- Return Arrays from Methods
- Understand Reference Semantics
- Use Multidimensional Arrays
- Read and write strings in a C program
- Use string handling functions in a C program.

7.2 INTRODUCTION

There are situations when it is needed to combine similar objects into a group and refer to them by a common name. The individual elements of the group are accessed by their position number in the group. C supports arrays for his purpose. An array is a data structure with the help of which a programmer can refer to and perform operations on a collection of similar data items such as simple lists or Tables of information. For example, a list of names of 30 students in a class can be grouped as shown in Figure 7.2.1.

Name_list= Name of the array

Figure 7.2.1: List of Names

The list shown in above Figure is an example of an array called Name_list which is a set of 0 to 29 memory locations sharing a common name i.e. Name_list. An individual element within the array can be designated by an integer known as a subscription or index number. For example the 14^{th} name in the list will be referred to as Name_list[14] and the 0^{th} name in the list as Name_list[0].

The individual elements of an array are called subscripted variables. Name_list[14] and Name_list[0] are examples of subscripted variables of array Name_list with subscripts 14 and 0 respectively. A subscript can be an integer or a variable of integer type.

An array can be compared to house in a block which are numbered as A0, A1, A2……A19. They share a common block name i.e. A. They also have unique house number 0, 1, 2, 3……19 assigned to them

based on their location. The house number along with the block name makes a unique address. Like the occupants of each house could be different at different times, the contents of each element of the array hold different values at different times of program execution.

So, **Array** can be defined as data structure which 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.

It may be noted here that all the elements of an array must be of same data type (except void). Therefore, an array is also known as derived data type because it is made up of data items which have their own type.

7.3 CLASSIFICATION OF ARRAYS

An array whose elements are specified by a single subscript is known as one-dimensional array. Example: Name is a One-Dimensional array containing n elements, the individual array elements will be Name [0], Name [1],……, Name[n-1]. Since only one subscript or index follow the variable name marks, therefore, this array is called one-dimensional array. One-Dimensional Array is also named as vectors or lists. The array whose elements are specified by more than one subscript is known as multi-dimensional array.

Multidimensional array consist of Two-Dimensional array and Three Dimensional Array. Two-Dimensional arrays are identified by two subscripts; Three-dimensional arrays have three subscripts and so on. These are discussed in further sections.

7.3.1 One Dimensional Array

One-Dimensional arrays are suitable for processing of lists of items of identical types. They are very useful for problems that require the same operation to be performed on a group of data.

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, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

Now balance is a variable array which is sufficient to hold up-to 10 double numbers.

The array declaration provides the following information to the compiler.

- Name of the array: In the above example Name of the array is *balance.*
- Types of the elements to be stored in the array: In the above example Types of the elements to be stored in the array are double.
- Number of subscript: In the above example Number of subscript it has are 1, so it make it One Dimensional.
- Array Size:In the above example Array Size is 10.

Examples of valid array declarations are:

1. int rollno[60];
2. char names[20];
3. double salary[200];
4. float percentage[25];

Generally arrays are manipulated in a fashion of component-by-component processing. In simple words we can say that similar operations are performed on some or all the components of the array.

Following **Table7.3.1.1** contain few array statements with their meaning.

Statement	Meaning
scanf("%d", &list[5]);	Read from keyboard, the fifth element of array called list.
printf("%d",list[i]);	Print the i^{th} element of array list on screen where i is an integer variable or constant.
list[i]= list[j];	Assign the contents of j^{th} element of array list to its i^{th} element.

list[k]++;	Increment k^{th} element of array list.
Net=gross-list[5];	Subtract the contents of 5^{th} element of array list from variable gross and store the result in variable Net.
for(i=0;i<10;i++) scanf("%d",&list[i]);	Read from keyboard, the elements of array list in the range list[0] to list[9].

From above table it can be observed that an individual element of an array behaves like a normal variable whereas the array as such represents a list of items of a certain data types. The items are internally stored in continuous memory locations.

Initializing Arrays -

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

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4^{th} index will be 5^{th} i.e. last element because all arrays have 0 as the index of their first element which is also called **base index**. Figure 7.3.1.1 is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
Balance	**1000.0**	**2.0**	**3.4**	**7.0**	**50.0**

Figure 7.3.1.1 Array Example

Accessing Array Elements -

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following program is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

Program – Write a program in which arrays are declared, value is assigned and accessing of arrays.

```
#include <stdio.h>
int main ()
{
int n[ 10 ]; /* n is an array of 10 integers */
int i,j;
/* initialize elements of array n to 0 */
for ( i = 0; i < 10; i++ )
{
n[ i ] = i + 100; /* set element at location i to i + 100 */
}
/* output each array element's value */
for (j = 0; j < 10; j++ )
{
printf("Element[%d] = %d\n", j, n[j] );
}
return 0;
```

```
}
```

Output –

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

Program – Write a program which reads marks of N students in an array and add a grace of 10 marks to every element of the array.

```
#include <stdio.h>
int main ()
{
int marks[30];
int i,NumStud;
printf("\n Enter the number of students :");
scanf("%d", &NumStud);
printf("\n Enter the marks of the students");
for(i=0;i<NumStud; i++)
{
printf("\n Enter Marks:");
```

```
scanf("%d", &marks[i]);      /* read marks of i^th student */
}
/* Add 10 every element of the array */
for(i=0;i<NumStud; i++)
marks[i]=marks[i]+10;
/*Display the modified list*/
printf("\n The modified list:");
for(i=0;i<NumStud; i++)
printf("%4d",marks[i]);
}
```

Output –

Enter the number of students: 4

Enter the marks of the students

Enter Marks: 79

Enter Marks: 70

Enter Marks: 60

Enter Marks: 65

The modified list: 89

80

70

75

Multi-dimensional Arrays

An array having more than one subscript is known as multidimensional array. C programming language allows multidimensional arrays of arbitrary dimensions. Here is the general form of a multidimensional array declaration:

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional 5, 10. 4 integer array:

int threedim[5][10][4];

7.3.2 Two Dimensional Array

The simplest form of the multidimensional array is the two-dimensional array. A two dimensional array (having two subscripts) is suitable for table processing or matrix manipulations. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x, y you would write something as follows:

type arrayName [x][y];

where

type: type can be any valid C data type

arrayName: Array Name will be a valid C identifier.

x and y: A two dimensional array can be think as a table which will have x number of rows and y number of columns.

A Two-Dimensional array a [3] [4] contains three rows and four columns is shown in Figure 7.3.2.1.

	Column 0	**Column1**	**Column 2**	**Column 3**
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Figure 7.3.2.1: Example of Multidimensional array of a[3][4]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i & j are the subscripts that uniquely identify each element in a. For Example, a two dimensional array of 5 rows and 4 columns (i.e. total 5*4= 20 locations) of integer type can be declared as

int mat[5][4];

Let us assume that name of the array is mat.

It may be noted here that a two-dimensional array is stored in row major order or column major order. In fact, most of the two-dimensional array processing is also done in row major order in the sense that the processing proceeds from 0^{th} row to the last row in the array and within a row the processing starts from 0^{th} column to the last column in the row. For example, the two-dimensional array mat[5][4] can be read in the following fashion:

0. read the array elements of row 0: mat[0][0],mat[0][1],……..mat[0][3]
1. read the array elements of row 1: mat [1][0],mat[1][1],……..mat[1][3]
2. ……..
3. ……..
4. read the array elements of row 4: mat [4][0],mat[4][1],……..mat[4][3]

From above given steps, we can observe that the row index varies from 0 to 4 and within a row, the column index varies from 0 to 3. Thus, it is a case of nested loops. Now we can code the steps 0 to 4 with the help of nested for loops as shown:

```
:
:
int mat[5][4];
:
:
for(i=0;i<5;i++)
        {
                for(j=0;j<4;j++)
                scanf("%d", &mat[i][j]);
        }
:
```

:

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3}, /* initializes for row indexed by 0 */
{4, 5, 6, 7}, /* initializes for row indexed by 1 */
{8, 9, 10, 11} /* initializes for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts i.e. row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array.

Program - Write a program to illustrate two dimensional array by using nested loop.

```
#include <stdio.h>
int main ()
{
/* an array with 5 rows and 2 columns*/
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
int i, j;
```

```
/* output each array element's value */
for ( i = 0; i < 5; i++ )
{
for ( j = 0; j < 2; j++ )
{
printf("a[%d][%d] = %d\n", i,j, a[i][j] );
}
}
return 0;
}
```

Output –

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

Program – Write a program which prints lower triangular and upper triangular matrices.

Consider square matrix given below:

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

The lower triangular matrix will be:

0			
6	7		
8	2	8	

And the upper triangular matrix will be:

	3	9	4
		2	5
			7

```
#include<stdio.h>
main()
{
int mat[10][10];
int i,j; /*indexes of nested loops*/
int N; /*order of square matrix*/
printf("\n Enter the order of the matrix<10:");
scanf("%d",&N);
printf("\n Enter the elements:");
for(i=0;i<N;i++)
for(j=0;j<N;j++)
scanf("%d", &mat[i][j]);
printf("\n The upper triangular matrix is ……\n");
for(i=0;i<N;i++)
{
```

```
for(j=0;j<N;j++)
{
if(i<j)
printf("%d",mat[i][j]);
}
printf("\n");
}
printf("\n The lower triangular matrix is .....\n");
for(i=0;i<N;i++)
{
for(j=0;j<N;j++)
{
if(i>j)
printf("%d",mat[i][j]);
}
printf("\n");
}
}
```

Output –

Enter the order of the matrix<10:4

Enter the elements: 2

3

9

4

0

1

2

5

6

7

2

7

8

2

8

1

The upper triangular matrix is ……

□	3	9	4
□	□	2	5
□	□	□	7
□	□	□	□

The lower triangular matrix is ….

□	□	□	□
0	□	□	□
6	7	□	□
8	2	8	□

7.3.3 Three Dimensional Array

We often use sketches to help us visualize arrays, such as:

- 1D array - single row or column
- 2D array – matrix
- 3D array – cube of cells

- 4D array – row or column of cubes?

Example -

int A,

B1[6],

B2[6],

C[3][4],

D[3][4][5],

E[3][4][5][3];

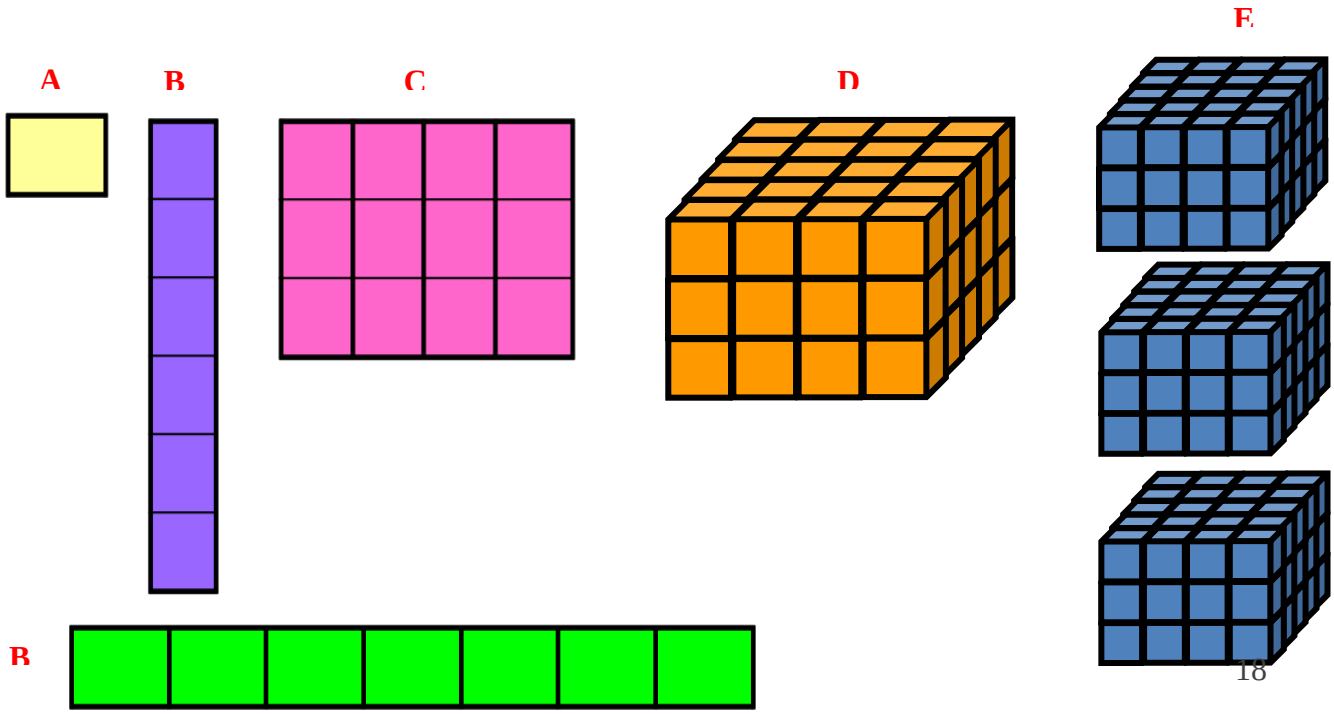


Figure7.2.3.1: Different dimensions of Array

Visualizing multi-dimensional arrays

Although we might find the visualizations to be useful, note that C stores the variables linearly, with the last index varied first (then the second index from the last, etc). This is important when loading an array with a list.

Example - int C[3][4], D[2][3][4];

Following diagram shows the visualization of multidimensional array. How the array is being stored in physical memory?

Array can be stored in two ways:

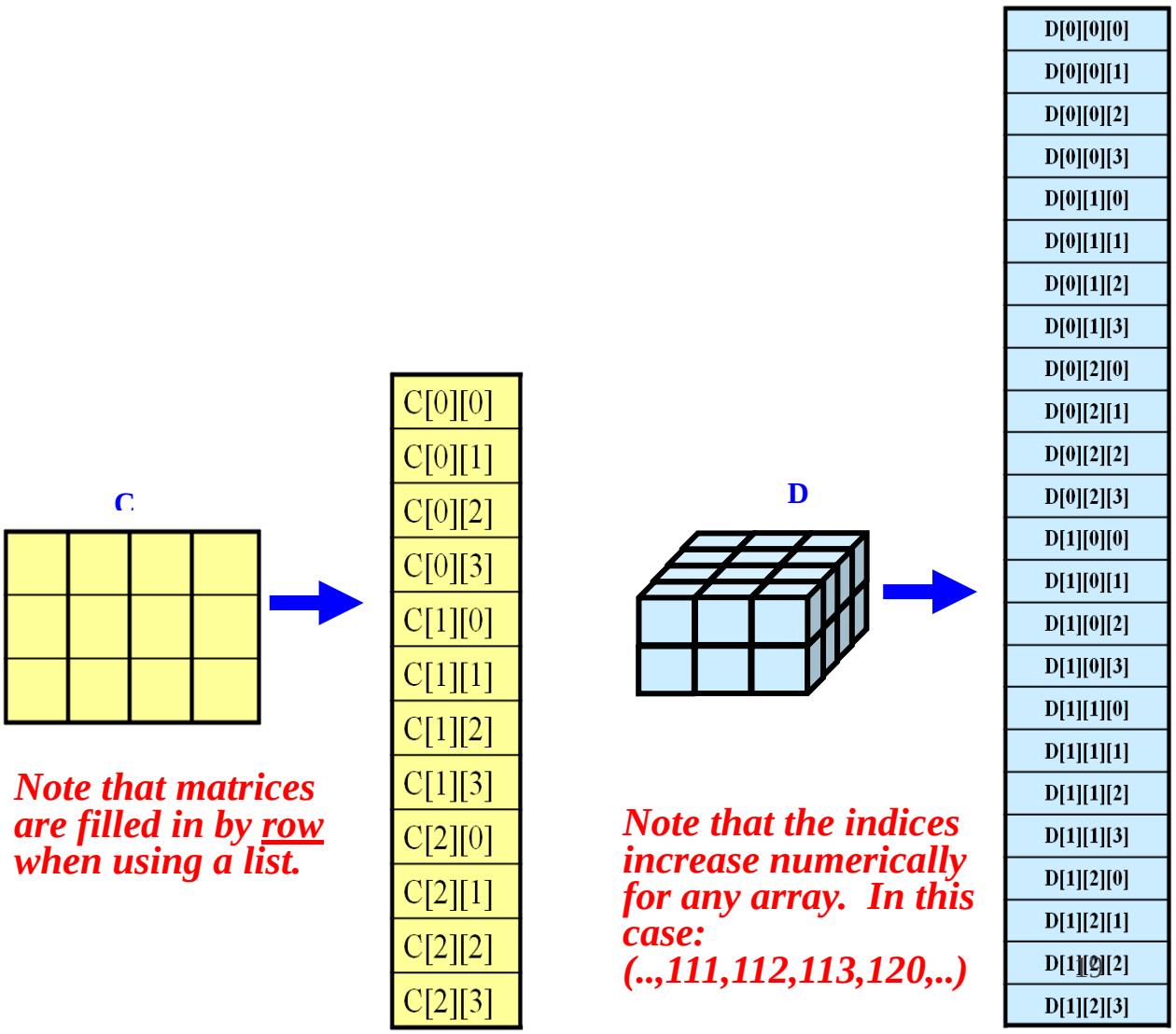
- Row major order
- Column major order

Computation of address in row major of i^{th} and j^{th} column in m*n array

= Base address+ size of data type ((i-1) *n + (j-1))

Computation of address in column major of i^{th} and j^{th} column in m*n array

= Base address+ size of data type ((i-1) + (j-1)*m)



Using 1D and 2D arrays is common and somewhat natural. Applications using 3 or more dimensions are harder to find.

Consider an example where rainfall is recorded over a 10-year period.

- Using a 1D array is awkward as there are 10*365 = 3650 days (not counting leap years). Using the following declaration:

```
double Rain[3650];  /* declare 1D array*/
Rain[2125] = 0.25;   /* it is hard to tell what date this is!*/
```

- Using a 3D array is clearer, where the 3 indices correspond to the year, month, and day.

```
const int Year = 10, Month = 13, Day = 32;
double Rain[Year][Month][Day];  /* declare 3D array*/
Rain[1][4][25] = 0.25;          /* 1/4 " of rain on April 25 in Year 1 of the study*/
```

Application of Arrays

The various applications of array have been illustrated through the following example program.

Program - Write a program for binary search of an element in an array.

Binary search is a faster way of searching an element in a given sorted list. This method uses the technique of divide and conquers. The list is divided into two halves. The element at the middle of the list is tested. If the desired element is found, the program terminates. Otherwise the desired element is present either in the first half or the second half of the list. This process is repeated successively until the element is found.

```
#include<stdio.h>
void main()
{
int series[20];
int i,n,pos,val;
```

```
int first,last,middle;
printf("\n Enter the size of  the list:");
scanf("%d",&n);
printf("\n Enter the list:");
for(i=0;i<n;i++)
{
printf("\n Enter number:");
scanf("%d",&series[i]);
}
printf("\n Enter the value to be searched:");
scanf("%d",&val);
/*Search the list*/
first=0;
last=n-1;
pos=-1;
while((first<=last)&&(pos==-1))
{
        middle=(first+last)/2;
        if(val= =series[middle])
        pos =middle;
                else
                if(val<series[middle])
                last=middle-1;
                        else
                        first=middle+1;
```

```
}
if(pos= =-1)
        printf("\n The elements is not present");
else
        printf("\n The element is present at %d Position", pos+1);
}
```

Output –

Enter the size of the list: 10

Enter the list: 1

2

3

4

5

6

7

8

9

10

Enter the value to be searched: 8

The element is present at 8 Position

Program – Write a program which inserts a value in an array at a particular location.

Insertion is an operation in which a new value is added at a particular place (sayi^{th}) in a list. In order to accommodate the new element all elements from the i^{th} location are shifted one step towards right. In this program a value Val is being inserted in an array called list at a given location called LOC .The program for this problem is given below:

```
#include<stdio.h>
#include<conio.h>
void main()
int list[20];
int back, i,n, loc, val;
printf("/n Enter the size of the list:");
scanf("%d",&n);
printf("\n Enter the list:");
printf("\n Enter Number:");
for(i=0;i<n;i++)
{
scanf("%d",&list[i]);
}
printf("\n Enter the value to be inserted and its location:");
scanf("%d%d",&val,&loc);
if(loc<n)
{
        back=n+1;
                while(back>loc)
                        {       /* Shift towards right*/
                        list[back]=list[back-1];
                        back--;
                        )
                list[loc]=val;
        n++;   /*increment the size of the list*/
```

```
        printf("\n The new list is....");
        for(i=0;i<n;i++)
        printf("%d",list[i]);
}
else
        printf("\n The insertion not allowed");
}
```

Output –

Enter the size of the list: 9

Enter the list:

Enter Number: 1

2

3

4

5

6

7

8

9

Enter the value to be inserted and its location: 10

10

The new list is....

12345678910

7.4 STRINGS

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null. Strings are used to manipulate text such as words and sentences.

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";

String constant is a sequence of characters enclosed by double quotes.

Figure 7.4.1 shows the memory presentation of above defined string in C:

H e l l o ‘\0’

Figure 7.4.1: Memory Representation of “Hello”

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

Declaration and Initialization of String–

String variable is an array of characters. The general form of declaration is

char string_name [size];

In this the C compiler automatically supplies the null character (\0) at the end of the string. So, the size of the string is always equal to the maximum number of characters in the string plus one (for null character or escape sequence).

Example –

- char name[30]; /*declares string variable name of length 30*/
- char fathers_name[30]; /* declares string variable fathers_name of length 30*/
- char name[10]= “ASHA RANI”

In this C compiler automatically inserts the null character*/

- char name[10]= {‘A’,’S’,’H’,’A’,’’,’R’,’A’,’N’,’I’,’\0’};

In this type of declaration it is programmer’s job to specify the null character at the last of the string.

Note –

char name[ ]= “ASHA RANI”

OR

char name[ ]= {‘A’,’S’,’H’,’A’,’’,’R’,’A’,’N’,’I’,’\0’};

These both statements declare string variable name of length 10 and name is initialized as ASHA RANII.

As we have studied in earlier lesson that **string uses %s format specification to read a string.**

char name[30];

scanf(“%s”, name);

This scanf function reads the input data like ASHA RANII.

printf(“%s”, name);

This statement is used to display the contents of the string name. We can also specify the precision with which the string is displayed. For Example,

printf (“%10.6 s”, name);

This specification indicates that the first six characters are to be displayed in a string of size 10.

Let us try to print above mentioned string.

Program - Write a program to print a string “Hello”.

```
#include <stdio.h>
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf ("Greeting message: %s\n", greeting );
```

return 0;

}

Output –

Greeting message: Hello

Note:

- If the number of elements being initialized is less that the specified size of an array then the remaining locations are loaded with 0's or banks.

  Consider the following declaration:

  int B[5] ={3,9};

  After initialization, the contents of B would be:

0	1	2	3	4
3	9	0	0	0

- While entering the string using scanf() two things are to kept in mind:
  1. The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays.
  2. scanf() is not capable of receiving multi-word strings. Therefore names such as 'ASHA RANI KUMARI' would be unacceptable. The way to get around this limitation is by using the function gets(). The usage of functions gets() and puts() is shown below:

For string input, the function gets () is used, and for string output function puts () is used.

gets() function -

This function is used to accept a string from standard input device. The entered string may consist of multiple words and the size of this entered string is limited by the declaration of the string variable. The input is supposed to be completed when an Enter key is being pressed.

puts() function –

This function is used to output a string constant to standard output device.

Program – Write a program that uses the gets() and puts() function to read a line of text from standard input device and put output on standard output device.

```
#include<stdio.h>
void main()
{
char name[30];
printf("\n Enter Full Name  and press Enter at last \n");
gets (name);
printf("You have typed \n");
puts(name);
}
```

Output –

Enter Full Name and press Enter at last

ASHA RANII KUMARI

You have typed

ASHA RANII KUMARI

Multidimensional Array of characters -

The multi dimensional arrays can be initialized in the similar fashion. Consider the matrix of order 4*3 given below:

$$\begin{bmatrix} 5 & 2 & 7 \\ 3 & 2 & 9 \\ 8 & 2 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

This matrix can be initialized in a two dimensional array as shown below:

```
int A [4] [3]=
{
5,  2,  7,
```

3, 2, 9,

8, 2, 5,

1, 2, 6

};

Similarly, a two dimensional array of characters or an array of strings can also be initialized as shown below:

```
char stud list[5] [10] = {
                        "Rohit",
                        "Harish",
                        "Rakesh",
                        "Asha"
                        "Seema"
                    };
```

Note:

The initialization of arrays at the time of declaration is possible only outside a function. Inside the function the initialized array has to be declared as static.

Program – Write a program that uses two parallel arrays country and capital to store the names of countries and their corresponding capitals of this world. It interacts with the user in such a way that for a given country, it gives the name of its capital. Use array initialization concept to store the names of states and capitals in the arrays. This program illustrates the usage of array initialization. It stores the names of countries and their capitals in two parallel arrays and manipulates them.

```
#include<stdio.h>
void main()
static char country[10][20]={
                        "India",
                        "Bangladesh",
```

```
                    "Bhutan",
                    "Nepal",
                    "Great Britain",
                    "Germany",
                    "Sri Lanka",
                    "Pakistan",
                    "France",
                    "Italy"
                    };
static char capital[10][20]= {
                    "New Delhi",
                    "Dhaka",
                    "Thimpu",
                    "Kathmandu",
                    "London",
                    "Berlin",
                    "Colombo",
                    "Islamabad",
                    "Paris",
                    "Rome"
                    };
int Num,i;
char choice;
do
{
```

```
printf("\n The world Database contains the following:");
printf("\n List of Countries....");
printf("\n County \t\t Number");
for(i=0;i<10;i++)
printf("\n%s\t\t%d", country[i],i);
printf("\n Enter the number of the country to see its capital:");
scanf("%d",&Num);
printf("\n The capital is .... %s", capital [Num]);
printf("\n Do you want to continue Y/N?");
fflush(stdin);
choice= getchar();     /*getchar is an character I/O function
}
while(choice = ='Y' || choice =='y');
}
```

Output –

The world Database contains the following:

List of Countries....

County	*Number*
India	*1*
Bangladesh	*2*
Bhutan	*3*
Nepal	*4*
Great Britain	*5*
Germany	*6*
Sri Lanka	*7*

Pakistan *8*

France *9*

Italy *10*

Enter the number of the country to see its capital: 1

The capital is ….New Delhi

Do you want to continue Y/N?

Character I/O functions:

For character input, the functions available are getchar(), getche() and getch(), whereas for character output, the function available is putchar().

getchar() function –

This function returns a character that has been recently typed. The typed character is echoed on screen and after typing there is a need to press Enter key.

General form of getchar() is –

variable_name=getchar();

getche() function –

This function also returns a character that has been recently typed without pressing Enter key.

General form of getche() is –

variable_name=getche();

getch() function –

This function also returns a character that has been recently typed and neither the programmer has t press enter key and nor the character is displayed on computer screen. This has advantage over other applications where user wants to hide the text, like password.

General form of getch() is –

variable_name=getch();

putchar()

This function output the character variable to standard output screen.

General form of putchar() is –

putchar(variable_name);

These functions are explained in detail in lesson 4.

Program – Write a program to show example of character I/O functions.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char c;
printf("\n Type a character and press Enter Key");
c=getchar();
printf("\n Typed character is");
putchar( c);
printf("\n Type character again");
c=getche();
printf("\n Typed character is");
putchar( c);
printf("\n Type a character again");
c=getch();
printf("\n Typed character is");
putchar(c);
}
```

Output –

Type a character and press Enter Key F

Typed character is F

Type character again o

Typed character is o

Type a character again

Typed character is h

7.5 STANDARD LIBRARY STRING FUNCTIONS

C supports a wide range of functions that manipulate null-terminated strings. Few of string functions are discussed below in following Table 7.5.1:

Table 7.5.1: String Functions

S.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.

Following example makes use of few of the above mentioned functions.

Program - Write a program to show example of string functions.

```
#include <stdio.h>
#include <string.h>
int main ()
{
char str1[10] = "Hello";
```

```
char str2[10] = "World";
char str3[10];
int len ;
/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) : %s\n", str3 );
/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );
/* total lenghth of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) : %d\n", len );
return 0;
}
```

Output –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

Programming Example –

Program – Write a program to calculate sum of the two matrices such that:

C (I, J) = A (I, J) + B (I, J)

```
/* this program adds two matrices of size m*n */
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int matA[10][10], matB[10][10],matC[10][10];
int i,j;  /*indices of nested loops*/
int m,n;
printf("\n Enter the size of the matrices");
scanf("%d %d",&m, &n);
printf("\n Enter the elements of matrix A");
for(i=0; i<m ;i++)
for(j=0; j<n; j++)
{
printf("\n Enter an element:");
scanf("%d",& matA[i][j]);
}
printf("\n Enter the elements of matrix B");
for(i=0;i<m;i++)
for(j=0;j<n; j++)
{
printf("\n Enter an element:");
scanf("%d",&matB[i][j]);
}
/*compute the sum */
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
matC[i][j]= matA[i][j]+ matB[i][j];
```

```
}
printf("\n The resultant matrix is ....\n");
for(i=0 ; i<m ; i++)
{
for(j=0; j<n; j++)
{
printf("%d",matC[i][j]);
}
printf("\n");
}
}
```

Output –

Enter the size of the matrices 2

2

Enter the elements of matrix A

Enter an element: 2

Enter an element: *4*

Enter an element: *2*

Enter an element: *4*

Enter the elements of matrix B

Enter an element: 3

Enter an element: *6*

Enter an element: *3*

Enter an element: *6*

The resultant matrix is ....

5

10

5

10

That is,

$$\begin{bmatrix}2 & 4\\4 & 4\end{bmatrix}+\begin{bmatrix}3 & 6\\3 & 6\end{bmatrix}=\begin{bmatrix}5 & 10\\5 & 10\end{bmatrix}$$

$$\begin{bmatrix}5 & 10\\5 & 10\end{bmatrix}$$

Program – Write a program to calculate the product of two matrices such that:

C[ ]m*n = A[ ]m*r * B[ ]r*n

The multiplication of two matrices requires that a row from matrix A is to be multiplied to a column of matrix B to generate an element of C shown below:

$$\begin{bmatrix}5 & 4\\3 & 2\end{bmatrix} * \begin{bmatrix}1 & 7\\2 & 9\end{bmatrix} = \begin{bmatrix}5*1+4*2 & 5*7+4*9\\3*1+2*2 & 3*7+2*9\end{bmatrix}$$

Thus C[0] [0]= A[0] [0] *B[0] [0] + A[0] [1] *B[1] [0]

The formula for an element of matrix C is as follows:

C[I] [J] =A[I] [K] * B[K] [J] for K varying from 0 to r-1

/*this program multiplies two matrices A & B and stores the result in matrix C*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int A[10][10],B[10][10], C[10][10];
int i,j,k;          /* The array indices*/
int m,n,r;          /* order of matrices*/
printf("\n Enter the order of the matrices m,r,n:");
```

```
scanf("%d%d%d", &m,&r, &n);
printf("\n Enter the elements of matrix A:");
for(i=0;i<m;i++)
for(j=0;j<r;j++)
{
printf("\n Enter an element:");
scanf("%d", &A[i][j]);
}
printf("\n Enter the elements of matrix B:");
for(i=0;i<r;i++)
for(j=0;j<n;j++)
{
printf("\n Enter an element:");
scanf("%d", &B[i][j]);
}
/* multiplies two matrices*/
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
C[i][j]=0;
for(k=0;k<r;k++)
{
C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
```

```
}
}
printf("\n The resultant matrix is ....\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
printf("%d",C[i][j]);
}
printf("\n");
}
}
```

Output –

Enter the order of the matrices m, r, n: 2

2

2

Enter the elements of matrix A:

Enter an element: 5

Enter an element: 4

Enter an element: 3

Enter an element: 2

Enter the elements of matrix B:

Enter an element: 1

Enter an element: 7

Enter an element: 2

Enter an element: 9

The resultant matrix is ….

13

71

7

39

That is,

$$\begin{bmatrix}5 & 4\\ 3 & 2\end{bmatrix} * \begin{bmatrix}1 & 7\\ 2 & 9\end{bmatrix} = \begin{bmatrix}5*1+4*2 & 5*7+4*9\\ 3*1+2*2 & 3*7+2*9\end{bmatrix} = \begin{bmatrix}13 & 71\\ 7 & 39\end{bmatrix}$$

7.6 SUMMARY

- An array is a fixed-length data structure that can contain multiple objects of the same type. An array can contain any type of object, including arrays.
- Arrays are declared with a statement of the form *array-type name[];*
- Arrays are objects. Therefore, an array must be instantiated before it can be used.
- The subscript of an array is enclosed in square brackets and always starts at 0 in C as C is 0 indexed languages.
- Every array knows its own length arrayname.length can be used to determine the size of the array.
- When use a for loop to go through all elements in an array, the index should start at 0 and ends at length-1. For example,

  for (int i = 0; i < a.length; i++)

  or

  for (int i = 0; i <= a.length-1; i++)
- The index can be a numerical expression, but the result of the expression must be of integer type.

- Referencing elements beyond the bounds of an array causes a run-time error.
- Size of an array must be a constant.
- In memory multidimensional arrays are represented in two ways, as Row major order and Column major order.
- Strings are used to manipulate text such as words and sentences.
- The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'.

7.7 KEYWORDS

Array - Array is a data structure that represents a collection of the same types of data.

Index – A non negative integer that refers to the cardinal position of a member variable in an array.

One-dimensional array – An array having a single index.

Multi-dimensional array – An array having more than one index.

Strings - A string is a sequence of characters treated as a group.

String variable - Allocate an array of a size large enough to hold the string (plus 1 extra value for the delimiter).

String Input & String Output – It require %s field specification in scanf to read string and %s field specification in printf to write a string respectively.

7.8 REVIEW QUESTIONS

1. What do you mean by arrays? How its declaration and initialization is being done and what are the applications of array?
2. Describe the different types of array. Explain with example.
3. Differentiate the following :
    - Single Dimension and Multi Dimension Array
    - Row major and column major array.
4. What are strings? Explain it with example?
5. Explain the following string functions:

- Strcpy()
- Strlen()
- Strcmp()
- Strcat()

7.9 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma, *Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press, 1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 8	
FUNCTIONS IN 'C'	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

8.1 OBJECTIVE

The main objective of this lesson is to aware students about functions and their different types in C with their syntax. How execution control takes place between called and calling function is explained. Functions are small manageable units defined as the building blocks of any program. In this lesson, we will see that by using arguments how we can pass parameters between called and calling function. Any function can be called either call by value method or by using pointers method called as call by

reference. Different categories of functions on how functions return values are discussed. At last, recursive functions are explained with example.

8.2 INTRODUCTION

A computer program cannot handle all the tasks by it self. Instead its requests other program like entities – called functions in C – to get its tasks done. A function is a self contained block of statements that perform a coherent task of same kind. We can divide a long C program into small blocks which can perform a certain task.

In C programming, all executable code resides within a function. Functions are a powerful programming tool. A function is a named block of code that performs a task and then returns control to a caller. Other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method" -- in C, these are all functions. C functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them. Actually, Collection of these functions creates a C program. A function is a group of statements that together perform a task. Every C program has at least one function which is main (). A function is like a black box. It takes in input, does something with it, and then spits out an answer as shown in Figure 8.2.1.

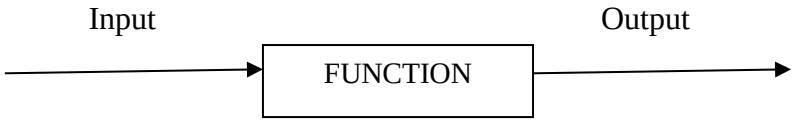


Figure 8.2.1: Overview of a Function

There are 3 aspects in each C function. They are,

- Function declaration or prototype - This informs compiler about the function name, function parameters, their order, their type and return value's data type.

  return_type function_name (argument list);

- Function call – This calls the actual function.

  function_name (arguments list);

- Function definition – This contains all the statements to be executed.

```
return_type function_name ( arguments list )
{
        Body of function;
}
```

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

- Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters. Functions with variable-length argument lists are functions that can take a varying number of arguments. An example in the C standard library is the printf function, which can take any number of arguments depending on how the programmer wants to use it.
- Function Body: The function body contains a collection of statements that define what the function does.

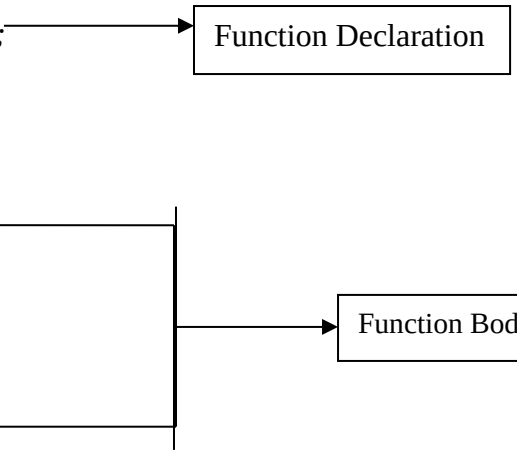
A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call. As a basic example, suppose you are writing code to print out the first 5 squares of numbers, do some intermediate processing, and then print the first 5 squares again. We could write it like this:

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

  for(i=1; i <= 5; i++)
  {
    printf("%d ", i*i);
  }
  return 0;
}
```

We have to write the same loop twice. We may want to somehow put this code in a separate place and simply jump to this code when we want to use it. This would look like as shown in below code.

```
#include <stdio.h>
void_print_squares(void);
void print_squares(void)
{
  int i;
  for(i=1; i <=5; i++)
  {
    printf("%d ", i*i);
```



```
  }
}
int main(void)
{
  print_squares();
  print_squares();
  return 0;
}
```

Function call

Calling a function in C program

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program as shown in figure 8.2.3.

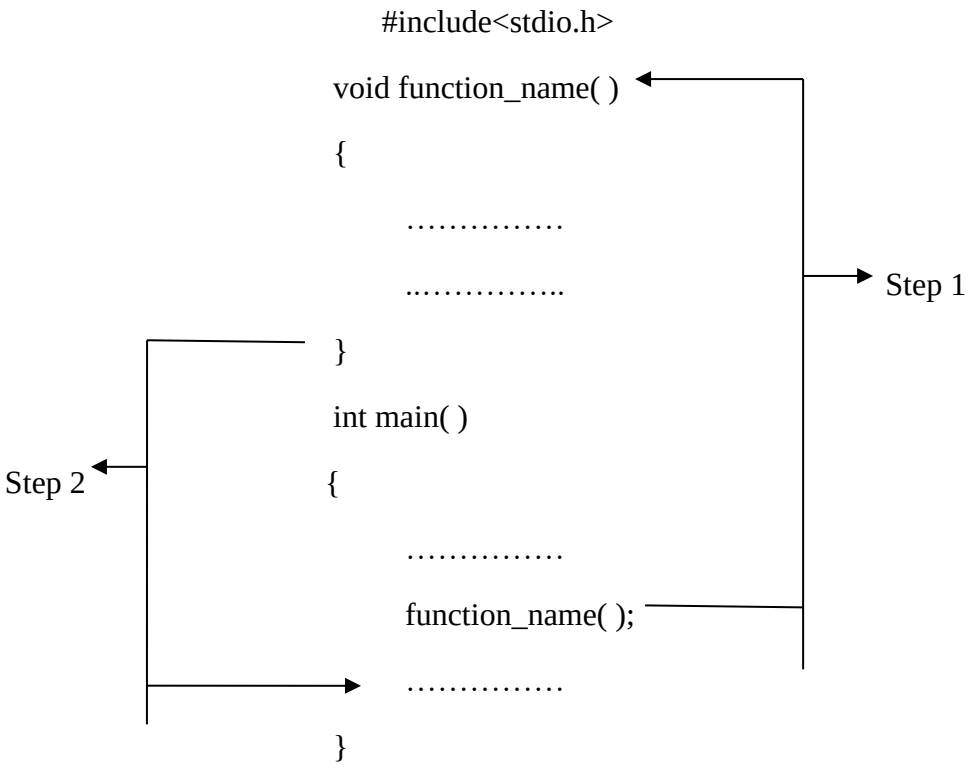


Figure 8.2.3: Function Calling

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. For example:

```
#include <stdio.h>
int max(int num1, int num2);          /* function declaration */
int main ()                           /*main function*/
{
   int a = 100;                       /* local variable definition */
   int b = 200;
   int ret;
   ret = max(a, b);                   /* calling a function to get max value */
   printf( "Max value is : %d\n", ret );
}
int max(int num1, int num2)           /* function returning the max between two numbers */
{
int result;                           /* local variable declaration */
    if (num1 > num2)
      result = num1;
   else
      result = num2;
    return result;
}
```

Output:

Max value is: 200

Within the main body, when control reaches ret = max (a, b); statement, then max ( ) function is called.

As soon as function is called, control gets transferred to called function body max ( ), where calculations is performed and result is transferred back to the statement following ret = max (a, b); i.e. printf("Max value is : %d\n", ret); and result is displayed on user screen.

Function Arguments

Function is good programming style in which we can write reusable code that can be called whenever require. Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called argument. If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

- call by value
- call by reference

Call by value: If data is passed by value, the data is copied from the variable used in for example main () to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function. Let's take a look at a call by value example:

Program - Write a program to illustrate calling a function by call by value.

```
#include <stdio.h>
void call_by_value(int x)
{
        printf("Inside call_by_value x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside call_by_value x = %d after adding 10.\n", x);
}
int main( )
{
```

```
    int a=10;
    printf("a = %d before function call_by_value.\n", a);
    call_by_value(a);
    printf("a = %d after function call_by_value.\n", a);
    return 0;
}
```

Output:

a = 10 before function call_by_value.

Inside call_by_value x = 10 before adding 10.

Inside call_by_value x = 20 after adding 10.

a = 10 after function call_by_value.

In the main () we create a integer that has the value of 10. We print some information at every stage, beginning by printing our variable a. Then function call_by_value is called and we input the variable a. This variable (a) is then copied to the function variable x. In the function we add 10 to x (and also call some print statements). Then when the next statement is called in main () the value of variable a is printed. We can see that the value of variable a isn't changed by the call of the function call_by_value().

Call by value method 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 uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Let us consider another program of interchange two numbers to understand call by value.

Program - Write a program to interchange two numbers by call by value.

```
#include<stdio.h>
void interchange(int number1, int number2)
{
   int temp;
```

```
    temp = number1;
    number1 = number2;
    number2 = temp;
}
int main( )
{
    int num1=50,num2=70;
    interchange(num1,num2);
    printf("\nNumber 1 : %d",num1);
    printf("\nNumber 2 : %d",num2);
    return(0);
}
```

Output:

Number 1: 50

Numb 2: 70

Figure 8.2.4 will help you to better understand how parameters are passed by value.

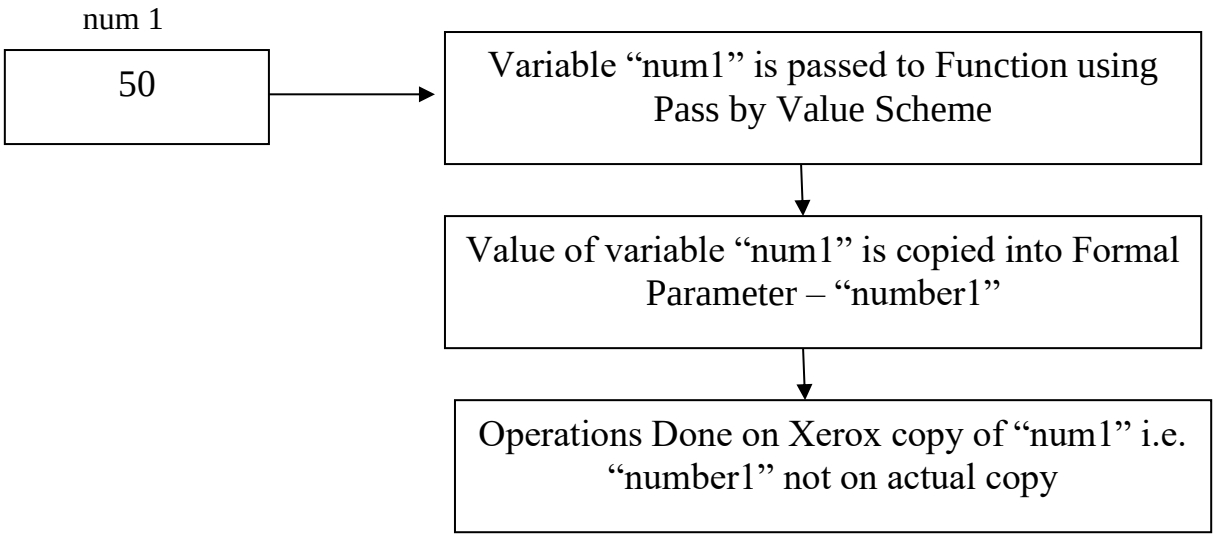


Figure 8.2.4: Passing Parameters by Value

Call by reference: If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main(). Let's take a look at a code example.

Program - Write a program to illustrate calling a function by call by reference.

```
#include <stdio.h>
void call_by_reference(int *y)
{
        printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
int main( )
{
        int b=10;
        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);
        return 0;
}
```

Output:

b = 10 before function call_by_reference.

Inside call_by_reference y = 10 before adding 10.

Inside call_by_reference y = 20 after adding 10.

b = 20 after function call_by_reference.

In this example, we start with an integer b that has the value 10. The function call_by_reference() is called and the address of the variable b is passed to this function. Inside the function there is some before and after print statement done and there is 10 added to the value at the memory pointed by y. Therefore at the end of the function the value is 20. Then in main () we again print the variable b and as you can see the value is changed (as expected) to 20.

Call by reference method 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. This means that changes made to the parameter affect the argument. Let us understand the program of interchange by call by reference.

Program - Write a program to interchange two numbers by call by reference.

```
#include<stdio.h>
void interchange(int *num1,int *num2)
{
   int temp;
   temp  = *num1;
   *num1 = *num2;
   *num2 = *temp;
}
int main()
{
   int num1=50,num2=70;
   interchange(&num1,&num2);
   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);
   return(0);
```

}

Output:

Number 1: 70

Number 2: 50

Figure 8.2.5 will help you to better understand how parameters are passed by reference.

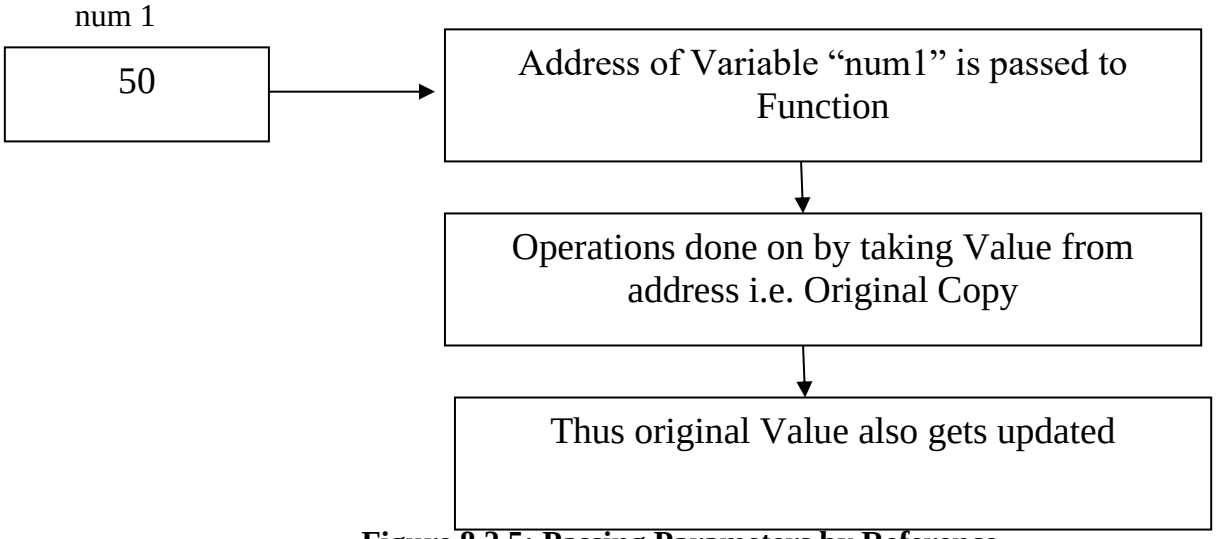


Figure 8.2.5: Passing Parameters by Reference

One advantage of the call by reference method is that it is using pointers, so there is no doubling of the memory used by the variables (as with the copy of the call by value method). This is of course great, lowering the memory footprint is always a good thing. So why don’t we just make all the parameters call by reference?

There are two reasons why this is not a good idea and that you (the programmer) need to choose between call by value and call by reference. The reason are: side effects and privacy. Unwanted side effects are usually caused by inadvertently changes that are made to a call by reference parameter. Also in most cases you want the data to be private and that someone calling a function only be able to change if you want it. So it is better to use a call by value by default and only use call by reference if data changes are expected.

8.3 TYPES OF FUNCTION

Function in programming is a segment that groups a number of program statements to perform specific

task.

A C program has at least one function main( ). Without main() function, there is technically no C program. Basically, there are two types of functions in C on basis of whether it is defined by user or not.

- Library function
- User defined function

8.3.1 C standard library functions

Library functions are inbuilt functions which are grouped together and placed in a common place called library. Each library function performs specific operation. We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.

While the C language doesn't itself contain functions, it is usually linked with the C Standard Library. To use this library, you need to add a #include directive at the top of the C file, which may be one of the following as shown in Table 8.3.1.1.

Table 8.3.1.1: #include Directives

<assert.h>	<limits.h>	<signal.h>	<stdlib.h>
<ctype.h>	<locale.h>	<stdarg.h>	<string.h>
<float.h>	<math.h>	<stdio.h>	<complex.h>
<errno.h>	<setjmp.h>	<stdef.h>	

These library functions are created by the persons who designed and created C compilers. Example: printf(), scanf(), strcpy() etc. Function prototype and data definitions of these functions are written in their respective header file. For example: If you want to use printf() function, the header file <stdio.h> should be included.

```
#include <stdio.h>
void main( )
{
    printf("Catch me if you can.");
}
```

If you write printf() statement without including header file, this program will show error.

There is at least one function in any C program, i.e., the main() function (which is also a library function). This program is called at program starts. There are many library functions available in C programming to help the programmer to write a good efficient program. Suppose, you want to find the square root of a number. You can write your own piece of code to find square root but, this process is time consuming and the code you have written may not be the most efficient process to find square root. But, in C programming you can find the square root by just using sqrt() function which is defined under header file "math.h".

Program - Write a program to calculate square root of a number by using math.h file.

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

List of Standard Library Functions under Different Header Files in C Programming.

- <ctype.h> Header File

Header file "ctype.h" includes numerous standard library functions to handle characters (especially test characters). Table 8.3.1.2 shows list of function included in <ctype.h>.

Table 8.3.1.2: List of functions included in <ctype.h>

Function	Description

isalnum	Tests whether a character is alphanumeric or not
isalpha	Tests whether a character is aplhabetic or not
iscntrl	Tests whether a character is control or not
isdigit	Tests whether a character is digit or not
isgraph	Tests whether a character is grahic or not
islower	Tests whether a character is lowercase or not
isprint	Tests whether a character is printable or not punctuation or not
ispunct	Tests whether a character is punctuation or not
ispace	Tests whether a character is white space or not
isupper	Tests whether a character is uppercase or not
isxdigit	Tests whether a character is hexadecimal or not
tolower	Converts to lowercase if the character is in uppercase
toupper	Converts to uppercase if the character is in lowercase

- <math.h>

math.h is a header file which is commonly used for mathematical operations. Some functions of this header file uses floating point numbers. The functions which accepts angle are accepted in terms of radians. Table 8.3.1.3 shows list of functions included in <math,.h>.

Table 8.3.1.3: List of functions included in <math.h>

Function	Description
acos	Computes arc cosine of the argument
acosh	Computes hyperbolic arc cosine of the argument
asin	Computes arc sine of the argument
asinh	Computes hyperbolic arc sine of the argument

atan	Computes arc tangent of the argument
atanh	Computes hyperbolic arc tangent of the argument
atan2	Computes arc tangent and determine the quadrant using sign
cbrt	Computes cube root of the argument
ceil	Returns nearest integer greater than argument passed
cos	Computes the cosine of the argument
cosh	Computes the hyperbolic cosine of the argument
exp	Computes the e raised to given power
fabs	Computes absolute argument of floating point argument
floor	Returns nearest integer lower than the argument passed.
hypot	Computes square root of sum of two arguments (Computes hypotenuse)
log	Computes natural logarithm
log10	Computes logarithm of base argument 10
pow	Computes the number raised to given power
sin	Computes sine of the argument
sinh	Computes hyperbolic sine of the argument
sqrt	Computes square root of the argument
tan	Computes tangent of the argument
tanh	Computes hyperbolic tangent of the argument

- <stdio.h>

stdio.h refers to standard input/output header file. it is header file in C's standard library which contains constants, macros definitions and declarations of functions. It includes types used for various standard input and output operations. Table 8.3.1.4 shows list of functions included in <stdio.h>.

Table 8.3.1.4: List of functions included in <stdio.h>

Function	Description
scanf	Used to take input from the standard input stream
gets	Read characters from stdin while a new line is inserted
printf	Prints to the standard output stream
putc	Writes and returns a character to a stream
putchar	It works as same of putc(stdout)
puts	Outputs a character string to stdout
fopen	Opens a file to read or write
fwrite	Writes data to a file
fputs	Writes a string to a file
fread	Reads data from a file
fseek	Seeks file
fclose	Closes a file
remove	Deletes or removes a file
rename	Renames a file
rewind	Adjusts the specified file so that the next I/O operation will take place at the beginning of the file. "rewind" is equivalent to fseek(f,0L,SEEK_SET);

- <string.h>

'string.h' is a header file which includes the declarations, functions, constants of string handling utilities. These string functions are widely used today by many programmers to deal with string operations. Table 8.3.1.5 shows list of functions included in <string.h>.

Table 8.3.1.5: List of functions included in <string.h>

Function	Description
strlen	Returns the length of a string.

strlwr	Returns upper case letter to lower case.
strupr	Returns lower case letter to upper case.
strcat	Concatenates two string.
strcmp	Compares two strings.
strrev	Returns length of a string.
strcpy	Copies a string from source to destination.

8.3.2 User defined functions

C provides programmer to define their own function according to their requirement known as user defined functions. As mentioned earlier, every C program begins from main() and program starts executing the codes inside main() function. When the control of program reaches to function_name() inside main() function. The control of program jumps to void function_name() and executes the codes inside it. When, all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function_name() from where it is called. Remember, the function name is an identifier and should be unique. Analyze the figure 8.2.3 for understanding the concept of function. The functions which are developed by user at the time of writing a program are called user defined functions. So, user defined functions are functions created and developed by user. When not using user defined functions, for a large program the tasks of debugging, compiling etc may become difficult in general. That's why user defined functions are extremely necessary for complex programs. The necessities or advantages are as follows:

- It facilitates top-down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
- The length of a source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function for further investigations.
- A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratc

Elements of user defined functions:

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition
2. Function call
3. Function declaration

These elements are discussed in detail in introduction part of function above. Let us discuss these elements in brief.

The function definition is an independent program module that is specially written to implement to the requirements of the function. A function definition, also known as function implementation shall include the following elements

1. Function name
2. Function type
3. List of parameters
4. Local variable declarations
5. Function statements
6. A return statement

All the six elements are grouped into two parts; namely,

- Function header (First three elements)
- Function body (Second three elements)

The function header consists of three parts; function type, function name and list of parameter.

(a) Function Type

The function type specifies the type of value (like float or double) that the function is expected to return to the calling program. If the return type is not explicitly specified, C will assume that it is an integer type.

(b) Function name

The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function.

(c) List of Parameter

The parameter list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task.

Example:

float mul (float x, float y) {….}

sum (int a, int b) {….}

The function body is enclosed in braces, contains three parts, in the order given below:

1. Local variable declaration: Local variable declarations are statements that specify the variables needed by the function.

2. Function Statements: Function statements are statements that perform the task of the function.

3. Return Statements: A return statement is a statement that returns the value evaluated by the function to the calling program. If a function does not return any value, one can omit the return statement.

Advantages of user defined functions:

- User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

8.4 SOME IMPORTANT THINGS ABOUT FUNCTION

Key points to remember while writing functions in C:

- All C programs contain main () function which is mandatory.
- Main () function is the function from where every C program is started to execute.
- Name of the function is unique in a C program.
- C Functions can be invoked from anywhere within a C program.
- There can any number of functions be created in a program. There is no limit on this.

- There is no limit in calling C functions in a program.
- All functions are called in sequence manner specified in main() function.
- One function can be called within another function.
- C functions can be called with or without arguments/parameters. These arguments are nothing but inputs to the functions.
- C functions may or may not return values to calling functions. These values are nothing but output of the functions.
- When a function completes its task, program control is returned to the function from where it is called.
- There can be functions within functions.
- Before calling and defining a function, we have to declare function prototype in order to inform the compiler about the function name, function parameters and return value type.
- C function can return only one value to the calling function.
- When return data type of a function is "void", then, it won't return any values
- When return data type of a function is other than void such as "int, float, double", it returns value to the calling function.
- main() program comes to an end when there is no functions or commands to execute.

Uses of C functions

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

8.5 CATEGORIES OF FUNCTION

C function arguments and return values:

All C functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function. Now, we will see simple example C

programs for each one of the below.

- C function without arguments (parameters) and without return value
- C function with arguments (parameters) and without return value
- C function with arguments (parameters) and with return value
- C function without arguments (parameters) and with return value

Function with no arguments and no Return Value in C

```
#include<stdio.h>
void area();  // Prototype Declaration
void main()
{
        area();
}
void area()
{
   float area_circle;
   float rad;
   printf("\nEnter the radius : ");
   scanf("%f",&rad);
   area_circle = 3.14 * rad * rad ;
   printf("Area of Circle = %f",area_circle);
}
```

Output :

Enter the radius : 3

Area of Circle = 28.260000

Consider main function -

```
void main()
{
        area();
}
```

We have just called a function, we can see that there is no variable or anything specified between the pair of round brackets.

void area();

Now in the prototype definition (line No 3) of the function we can see the return value as Void. Void means it does not return anything to the calling function.

Function with arguments and no Return Value:

```
#include<stdio.h>
#include<conio.h>
 void area(float rad);  // Prototype Declaration
void main()
{
float rad;
   printf("nEnter the radius : ");
   scanf("%f",&rad);
   area(rad);
getch();
}
void area(float rad)
{
float ar;
ar = 3.14 * rad * rad ;
```

```
printf("Area of Circle = %f",ar);
}
```

Output:

Enter the radius: 3

Area of Circle = 28.260000

Function accepts argument but it does not return a value back to the calling Program. It is Single (One-way) Type Communication. Generally Output is printed in the Called function. Here area is called function and main is calling function.

Function with argument and return type

```
#include<stdio.h>
float calculate_area(int);
int main()
{
   int radius;
   float area;
   printf("\nEnter the radius of the circle : ");
   scanf("%d",&radius);
   area = calculate_area(radius);
   printf("\nArea of Circle : ",area);
   return(0);
}
float calculate_area(int radius)
{
   float areaOfCircle;
   areaOfCircle = 3.14 * radius * radius;
```

```
    return(areaOfCircle);
}
```

Output :

Enter the radius of the circle : 2

Area of Circle : 12.56

In the above program we can see that inside main function we are calling a user defined calculate_area() function. We are passing integer argument to the function which after area calculation returns floating point area value.

Function without arguments and with return type

```
#include<stdio.h>
int sum();
int main()
{
    int addition;
    addition = sum();
    printf("\nSum of two given values = %d", addition);
    return 0;
}
int sum( )
{
      int a = 50, b = 80, sum;
      sum = a + b;
      return sum;
}
```

Output:

Sum of two given values = 130

In this program, no arguments are passed in statement addition = sum (); in main function but sum (); function is returning integer type calculated sum to main function.

Always, only one value can be returned from a function. If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement. For example, if you use “return a, b, c” in your function, value for c only will be returned and values a, b won’t be returned to the program. In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

8.6 RECURSION

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping.

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

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

While using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go in infinite loop. Consider an example of recursion in C to find out sum of n natural

numbers.

Program - Write a program to calculate sum of n natural numbers by using recursion.

```
#include <stdio.h>
int sum(int n);
int main(){
   int num,add;
   printf("Enter a positive integer:\n");
   scanf("%d",&num);
   add=sum(num);
   printf("sum=%d",add);
}
int sum(int n){
   if(n==0)
      return n;
   else
      return n+sum(n-1);    /*self call  to function sum() */
}
```

Output

Enter a positive integer:

5

15

In, this simple C program, sum () function is invoked from the same function. If n is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, n is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

sum(5)

=5+sum(4)

=5+4+sum(3)

=5+4+3+sum(2)

=5+4+3+2+sum(1)

=5+4+3+2+1+sum(0)

=5+4+3+2+1+0

=5+4+3+2+1

=5+4+3+3

=5+4+6

=5+10

=15

Every recursive function must be provided with a way to end the recursion. In this example when, n is equal to 0, there is no recursive call and recursion ends.

Recursive functions are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series etc.

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

Factorial is represented using '!', so five factorial will be written as (5!), n factorial as (n!). Also

n! = n*(n-1)*(n-2)*(n-3)...3.2.1 and zero factorial is defined as one i.e. 0! = 1.

Program - Write a program to calculate factorial of a natural numbers by using recursion.

```
#include <stdio.h>
int factorial(unsigned int i)
{
  if(i <= 1)
```

```
  {
    return 1;
  }
  return i * factorial(i - 1);
}
int  main()
{
   int i = 15;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

Output:

Factorial of 15 is 2004310016

Following is another example which generates fibonacci series for a given number using a recursive function.

```
#include <stdio.h>
int fibonaci(int i)
{
  if(i == 0)
  {
    return 0;
  }
  if(i == 1)
  {
    return 1;
```

```
  }
  return fibonaci(i-1) + fibonaci(i-2);
}
int  main()
{
   int i;
   for (i = 0; i < 10; i++)
   {
     printf("%d\t%n", fibonaci(i));
   }
   return 0;
}
```

Output:

0 1 1 2 3 5 8 13 21 34

The main benefits of using recursion as a programming technique are that recursive functions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.

The main disadvantage of programming recursively is that, while it makes it easier to write simple and elegant programs, it also makes it easier to write inefficient ones.

8.7 SUMMARY

- The best way to develop and maintain a large program is to construct it from small piece or modules each of which itself is a complete unit.
- A function is named, independent section of C code that performs a specific task and optionally returns a value to calling program.
- A function can be without parameters, in which case parameter list is empty.

- There are three important parts of user defined function: function prototype, function definition, function calling.
- Before calling and defining a function, we have to declare function prototype in order to inform the compiler about the function name, function parameters and return value type.
- A function can be called any number of times in main () or in any other function.
- There are two types to invoke functions: call by value and call by reference.
- In call by value, a copy of arguments value is made and passed to called function.
- In call by reference, address of argument is passed to the function, instead of passing the arguments directly.
- Recursion is a programming technique that allows the programmer to express operations in terms of themselves.

8.8 KEYWORDS

Function- A group of statements that accepts specified number of values and returns a single computed value to caller.

Function declaration or prototype- This informs compiler about the function name, function parameters and return value's data type.

Call by value- If data is passed by value, the data is copied from the variable used in for example main () to a variable used by the function.

Call by reference- If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value.

Recursion- An act of a function calling itself repeatedly directly or indirectly until some condition becomes false.

8.9 REVIEW QUESTIONS

1. What do you mean by a function? List down different kinds of function in C.
2. What is difference between call by value and call by reference? Explain with examples.
3. What are the different categories of functions? Explain each of them by using examples.
4. Write a program to swap two numbers using function swap ().
5. Write a program to print all the prime numbers less than or equal to the given integer as input.

6. What are the important points that should be kept in mind while dealing with functions in C?
7. What do you mean by recursion? Explain it with the help of C code.
8. Write a program to print table of given integer by using recursion.

8.10 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language,* Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma, *Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press, 1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 9	
STORAGE CLASSES IN 'C'	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

9.1 OBJECTIVE

The objective of this lesson is to aware students about various scope rules in C. It gives the students an idea about automatic storage class, register storage class, use of static storage class and how the external storage class is being used in C.

9.2 INTRODUCTION

The range of code in a program over which a variable has a meaning is called as **scope** of the variable. Scope can also be defined as to how a variable can access by different parts of the program.

There are three types of scopes in C:

1. Local Scope
2. Function Scope
3. File Scope

Local Scope:

Every C code is surrounded by a pair of curly braces '{', '}'. We can declare variables inside the block. Such variables, called as local, can be referenced only within the body of the block.

When the control of program execution reaches the opening braces, these variables come into existence and get destroy as soon as the control leaves the block through the closing brace.

Example –

```
while (flag)
{
        int x;
        if(i>x)
                {
                int j;
                …..
                …..
                …..
                …..
                }
}
```

The scope of variable x is the while blocks and that of variable j is if-block. Now the variable x can also be referenced to in the if-block because this block is completely enclosed in the while block. On the other hand, the scope of variable j is only the if-block. Since a function is a block in itself, all the variables declared in, it has the local scope.

Function Scope

The function scope pertains to the variables declared in a function in the sense that a variables can be used anywhere in the function in which it is declared. Thus, we can use different variables with same names in different functions.

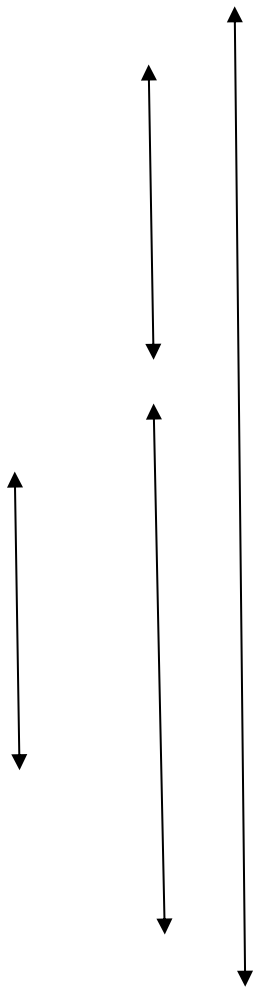
File scope

Since global variables are declared outside all blocks and functions, they are available to all the blocks and functions. Thus, the scope of a global variable encompasses the entire program file. This type of scope is known as file scope.

Example–

Global variable rate and some have file scope.

```
int rate;
int some;
main()
{
int Total, Net
……..
……..
}
int any()
{
        char ch
                {
                int i;
                …….
                ……
                }
}
```



The scope rules can be summarized as:

1. A local variable can be accessed only within the block in which it is declared. The local variable declared in a function exists only during the execution of the function. Therefore, these variables are also called automatic variables because they are automatically created and destroyed.

2. Since global variables are not declared in a particular function, they are available to all the functions. Thus, the scope of a global variable encompasses the entire program.

3. The scope of formal parameters is local to the function in which it is defined.

Scope tells the limit of variable in a program i.e. till where a variable can be accessed. And a, storage class defines the scope (visibility) and life time of variables and/or functions within a C program. This specifies precede the type that they modify.

What is a storage class?

Whenever a variable is created, the value of the variable is stored in either two locations. It might be stored in the CPU registers or memory. Either of these two locations always interacts with the compiler to save the variable. The storage class states the location of the storage of the variable of where it is stored. If a storage class is not mentioned by the user than the compiler allocates a default storage class to the variable.

A storage class basically states the life of a variable (lifetime). It even states the scope of the variable inside a function. It will even state the initial value of a variable if it is not assigned and most importantly it will state where the variable will be stored.

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

- auto
- register
- static
- extern

9.3 AUTOMATIC STORAGE CLASS

Local variable can be defined by *auto* specifier. However, this specifier is seldom used because we know that any variable declared within a block or function is a local variable and its scope lies within

that block. The features of this storage class are very simple. The variable of this storage class are always saved in **memory.** The default initial value of a variable is always a **garbage value**. A garbage value is an unpredicted value that is illogical to the user. The scope of the variable is local to the body of the function. The life of the variable remains until the control remains in the block where the variable is defined. The keyword "**auto**" is used to declare a variable of automatic storage class.

Thus, a local variable is by default auto. And, *auto* storage class is the default storage class for local variables.

{

int mount;

auto int month;

}

The example above defines two variables (mount and month) with the same storage class. Now the variable mount is local and by default automatic whereas the variable month has been explicitly defined as auto. Both are equivalent as far as scope and storage class is concerned.

Therefore, there is no use of auto specifier except that it removes confusion in certain situations. Auto can only be used within functions, i.e. local variables.

Program - Write a program to show example of automatic storage class.

```
#include<stdio.h>
#include<conio.h>
void main()
{
auto int a;
printf("%d",a)
}
```

Output:

1285

As seen above, the output is garbage value.

9.4 REGISTER STORAGE CLASS

What is a variable? A variable is nothing but a memory location in the main memory of the computer and the CPU has to access the main memory to manipulate the variable. This activity slows down the execution of the program because main memory is about 10 times slower than the CPU. Since, a number of registers are available in CPU itself and all of them are not used all the time, a few of them can be used to store some frequently used variables so that the execution becomes faster. C allows some variables to declare as register variables.

The register storage class is used to define local variables that should be stored in a register instead of RAM. The reserved word *register* can be prefixed to an ordinary variable declaration for the purpose of declaring it as register variable. This means that the variable has a maximum size equal to the register size (usually one word)

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

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

Main difference between auto and register is that variable declared as auto is stored in memory whereas variable declared as register is stored in CPU register. Since the variable is stored in CPU register, it takes very less time to access that variable. Hence it becomes very time efficient.
It is not necessary that variable declared as register would be stored in CPU registers. The number of CPU registers is limited. If the CPU register is busy doing some other task then variable might act as automatic variable.

Note –

1. We cannot apply the operator '&' on a register variable.
2. We cannot declare an array or structure of type register i.e. it has to be a scalar type.

9.5 STATIC STORAGE CLASS

If one desires that the local variables should retain its value even after the execution of the function in which it is declared, then the variable must be declared as *static*. The static storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. According to static storage class for variable the initial value of variable is zero. The scope depends where the variable is declared i.e. inside or outside the class. Static variables persist throughout the execution of process and it gets memory on heap section of the process. In C we can declare variable as well as function as static. Static modifier has different effects upon local variables and global variables.

Static Local Variables

When you apply the *static* modifier to a local variable, the compiler creates permanent storage for it, much as it creates storage for a global variable. The key difference between a *static* local variable and *static* global variable is that the *static* local variable remains known only to the block in which it is declared. In simple terms, a *static* local variable is a local variable that retains its value between function calls. Therefore, making local variables static allows them to maintain their values between function calls and the value of a static variable is not destroyed when the function terminates.

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

Static Global Variable -

Applying the specifier *static* to a global variable instructs the compiler to create a global variable known only to the file in which it is declared. Thus, a static global variable has internal linkage (will be described under the *extern* statement). This means that even though the variable is global, routines in other files have no knowledge of it and cannot alter its contents directly, keeping it free from side effects. For the few situations where a local *static* cannot do the job, you can create a small file that contains only the functions that need the global *static* variable, separately compile that file, and use it without fear of side effects.

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

Program – Write a program to show example of static storage class.

```
#include<stdio.h>
void Check();
int main()
{
 Check();
Check();
 Check();
 }
void Check()
{
static int c=0;
printf("%d\t",c);
 c+=5;
}
```

Output –

0

5

10

During first function call, it will display 0. Then, during second function call, variable c will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call.

If variable c had been automatic variable, the output would have been:

0

0

0

Let's see another example of C program that uses static storage class.

Program - Write a program that makes use of static storage class.

```
#include <stdio.h>
/* function declaration */
void func(void);
static int count = 5; /* global variable */
main()
{
while(count--)
{
func();
}
return 0;
}
/* function definition */
void func( void )
{
static int i = 5; /* local static variable */
i++;
printf("i is %d and count is %d\n", i, count);
}
```

Output –

i is 6 and count is 4

i is 7 and count is 3

i is 8 and count is 2

i is 9 and count is 1

i is 10 and count is 0

9.6 EXTERNAL STORAGE CLASS

C defines three categories of linkage: external, internal, and none. In general, functions and global variables have external linkage. This means they are available to all files that constitute a program. File scope objects declared as *static* (described in above section) 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. When the size of the program becomes very large then it has to be divided into parts and each part is stored in a separate file. The extern storage class is used to give a reference of a global variable that is visible to all the program files. In most cases, variable declarations are also definitions. However, by preceding a variable name with the extern specifier, you can declare a variable without defining it. Thus, when you need to refer to a variable that is defined in another part of your program, you can declare that variable using *extern.* When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

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

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

First File: main.c

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

Second File: write.c

```
#include <stdio.h>

extern int count;

void write_extern(void)

{

count = 5;

printf("count is %d\n", count);

}
```

Here extern keyword is being used to declare count in the second file where as it has its definition in the first file. Now compile these two files as follows:

$ gcc main.c write.c

This will produce a.out executable program, when this program is executed, it produces following

Output –

5

In real-world, multiple-file programs, extern declarations are normally contained in a header file that is simply included with each source code file. This is both easier and less error prone than manually duplicating extern declarations in each file.

Table 9.6.1 shows basic difference between different storage classes.

Table 9.6.1: Difference between different Storage Classes.

Storage Type	Created	Initialized	Scope	Purpose
auto	Each time the function or block is called	Can be initialized at the time of declaration	Within the block or function	As variable within a block.
static	First time when the function is called	Initialized at the time of declaration	Within the block or function	As variables which retain value even after

				the termination of a function
register	Same as auto	Same as auto	Same as auto	As most frequently used variables
extern	Created in the file where it has been declared as global.	Also initialized in the file where it has been declared as global.	Both the files where declared as global and where declared as extern.	As variables used by multiple files.

Programming Example –

Program – Write a program for example of C auto variable.

The scope of this auto variable is within the function only. It is equivalent to local variable. All local variables are auto variables by default.

```
#include<stdio.h>
void increment(void);
 int main()
{
increment();
increment();
increment();
increment();
}
void increment(void)
{
auto int i = 0 ;
```

```
printf ( "%d \n", i ) ;
i++;
 }
```

Output –

```
0
0
0
0
```

Program – Write a program for example of C static variable.

Static variables retain the value of the variable between different function calls.

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

Output –

0

1

2

3

*Program – Write a program for **e**xample of extern variable.*

The scope of this extern variable is throughout the main program. It is equivalent to global variable. Definition for extern variable might be anywhere in the C program.

```
#include<stdio.h>
int x = 10 ;
int main( )
{
extern int y ;
printf ( "The value of x is %d \n", x ) ;
printf ( "The value of y is %d",y ) ;
}
int y = 50 ;
```

Output:

The value of x is 10
The value of y is 50

*Program – Write a program for **e**xample of register variable.*

Register variables are also local variables, but stored in register memory, whereas auto variables are stored in main CPU memory.

Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory. But, only limited variables can be used as register since register size is very low. (16bits, 32 bits or 64 bits)

```
#include<stdio.h>
int main()
{
register  int i, arr[5];          // declaring array
 arr[0] = 10;     // Initializing array
 arr[1] = 20;
 arr[2] = 30;
 arr[3] = 40;
 arr[4] = 50;
 for (i=0;i<5;i++)
 {
 printf("value of arr[%d] is %d \n", i, arr[i]);              // Accessing each variable
 }
}
```

Output –

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

9.7 SUMMARY

- **Automatic storage class:**

The keyword used for Automatic storage class is 'auto'. The variable declared as auto is stored in the memory. Default value of that variable is garbage value. Scope of that variable is local to the block in which the variable is defined. Variable is alive till the control remains within the

block in which the variable is defined.

- **Register storage class:**

  The keyword used for Register storage class is 'register'. The variable declared as register is stored in the CPU register. Default value of that variable is garbage value. Scope of that variable is local to the block in which the variable is defined. Variable is alive till the control remains within the block in which the variable id defined.

- **Static storage class:**

  The keyword used for Static storage class is 'static'. The variable declared as static is stored in the memory. Default value of that variable is zero. Scope of that variable is local to the block in which the variable is defined. Life of variable persists between different function calls.

- **External storage class:**

  The keyword used for External storage class is 'extern'. The variable declared as static is stored in the memory. Default value of that variable is zero. Scope of that variable is global. Variable is alive as long as the program's execution doesn't come to an end. External variable can be declared outside all the functions or inside function using 'extern' keyword.

- Let's summarized all these storage specifier with their Storage place, Initial/Default value, its scope and the life of the storage specifier.

S.No.	Storage Specifier	Storage place	Initial / default value	Scope	Life
1	**auto**	CPU Memory	Garbage value	local	Within the function
2	**extern**	CPU memory	Zero	Global	Till end of the main program. Variable definition might be anywhere in the C program

3	**static**	CPU memory	Zero	local	Retains the value of the variable between different function calls.
4	**register**	Register memory	Garbage value	local	Within the function

9.8 KEYWORDS

Scope - The scope of a variable determines over what part(s) of the program a variable is actually available for use (active).

Longevity - It refers to the period during which a variables retains a given value during execution of a program (alive).

Local (internal) variables - Are those which are declared within a particular function.

Global (external) variables - Are those which are declared outside any function. The principal use of extern is to specify that an object is declared with external linkage elsewhere in the program.

Automatic Variables - Are declared inside a function in which they are to be utilized and are created when the function is called and destroyed automatically when the function is exited.

Register Storage Class in C Language - The register storage class is defined with the help of **register** keyword. The register storage class states that the variable is stored in CPU registers. The initial value of the variables is assigned with garbage values, same as automatic storage class. Similar to automatic storage class, the scope of the variable is local to the block and the life of the variable ends when the control leaves the block where it is defined.

Static Storage Class in C Language - The static storage class is defined by the **static** keyword. The storage of the variables of this type is stored in **memory**. The default initial value of the variable is assigned to **zero**. The value of the variable changes with different function calls, not limited to its defined block. The scope of the variable is local to the block.

External Storage Class in C Language - The external storage class is defined by the keyword **extern**. The variables are stored in the **memory** and the initial value by default is set to **zero** for a variable. The

scope of a variable inside a block is **global** to it and the life of the variable remains as long as the **program terminates.** Extern variables are always declared outside the body of a function because of its global scope and all the functions can use it efficiently.

9.9 REVIEW QUESTIONS

1. What do you mean by storage classes in C.?
2. What do you mean by automatic variables? Explain with example?
3. Explain register type of scope rule with example?
4. Differentiate between External and Internal variables?
5. Explain extern type of scope declaration with example?
6. What is the use of static type scope rule explain with example?

9.10 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma,*Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press,1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 10	
POINTERS	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

10.1 OBJECTIVE

Objective of this lesson is to make students learn the concept of pointers, why pointers are useful element in C. We will study the operators used by pointers i.e. (&, *). How the pointer variables can be initialized and how they can be used in C.

What all operations can be done on pointers? How the NULL pointer can be initialized. Therefore, the objective of this lesson is to link the concept of arrays and functions with pointer in topic pointer to

function and Array of pointers in further lesson.

10.2 THE CONCEPT OF POINTERS

Pointers in C are easy and fun to learn. Pointers are one of C's most powerful features, and they are used for a wide variety of purposes. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

A pointer is the memory address of an object. A pointer variable is a variable that is specifically declared to hold a pointer to an object of its specified type.

Applications –

1. They can provide a fast means of referencing array elements.
2. They allow functions to modify their calling parameters.
3. They support linked lists, binary trees, and other dynamic data structures.

Consider the following example which will print the address of the variables defined.

Program - Write a program to print address of defined variables.

```
#include <stdio.h>
int main ()
{
int var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

So you understood what memory address is and how to access it, so base of the concept is over. Now let us see what a pointer is.

10.3 DEFINITION OF A POINTER

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second as shown in Figure 10.3.1.

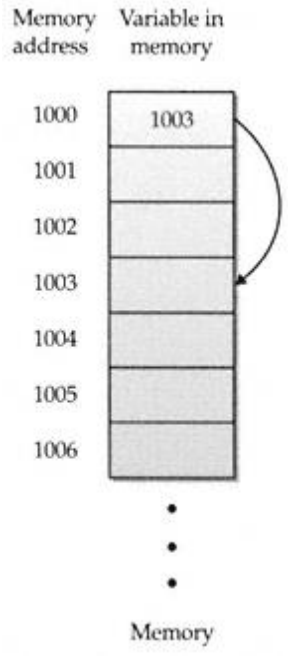


Figure.10.3.1. One variable points to another

A pointer is a variable whose value is the address of another variable i.e. direct address of the memory location.

The pointer has wide variety of uses in program:

1. To pass address of variable from one function to another.
2. To return more than one value from a function to the calling function.
3. For the creation of linked structures such as: linked lists and trees.

10.4 DEFINING A POINTER

This topic defines briefly the two operators that are used to manipulate pointers.

The first pointer operator is **&**, a unary operator that returns the memory address of its operand.

(Remember, a unary operator requires only one operand).

The '&' operator –

When a variable x is declared in a program, a storage location in main memory is made available by the compiler as shown below:

int x; = x

Thus, x is the name associated by the compiler to a location in the memory of the computer. Let us assume that, at the time of execution, the physical address of this memory location (called x) is 2712. Now please note that this memory location is viewed by the programmer as variable x and by the operating system as address 2712. The address of variable x can be obtained by '&', an address of operator. This operator when applied to a variable gives the physical memory address of the variable. Thus**,** &x will provide the address 2712.

Let's see this with the help of a program.

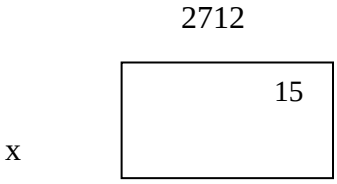
Program – Write a program to show the use of '&' operator.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x=15;
printf("\n Value of x=%d",x);
printf("\n Address of x=%d",&x);
getch();
}
```

Output –

Value of x=15

Address of x=2712



For Example –

m=&count

This places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. We can say that & means "the address of." Therefore, the preceding assignment statement means "m receives the address of variable count."

To better understand this assignment, assume that the variable count is at memory location 2000.

Also assume that count has a value of 100. Then, after the previous assignment, m will have the value 2000.

The '*' operator

The second pointer operator is *, which is the complement of &. The * is a unary operator that returns the value of the object located at the address that follows it. The '*' is an indirection operator. * means 'value at address operator'. In simple words, we can say that if address of a variable is known then the '*' operator provides a way of accessing the contents of the variable.

For example, if m contains the memory address of the variable count, (in above example)

q = *m;

This statement places the value of count into q. Now q has the value 100 because 100 is stored at location 2000, the memory address that was stored in m. Think of * as meaning "at address." In this case, you could read the statement as "q receives the value at address m." Let's see this with the help of a program.

Program – Write a program to show the use of '*' operator.

```
#include<stdio.h>
main()
{
int x=15;
printf("\n Value of x=%d",x);
printf("\n Address of x=%u", &x);
printf("\n Value at address %d=%d",&x,*(&x));
}
```

Output –

Value of x=15

Address of x=2712

Value at address 2712=15

Note - Both & and * have a higher precedence than all other arithmetic operators except the unary minus, with which they share equal precedence.

How to use Pointers?

To use pointers we have to declare the variable of pointer type. What are these Pointer type variables?

Pointer Variables –

Like any variable or constant, you must declare a pointer before you can use it to store any variable address. If a variable is going to be a pointer, it must be declared as such, the general form of a pointer variable declaration is:

type *var-name;

Here,

Type: Pointer's base type;(it must be a valid C data type).

var-name : name of the pointer variable.

*(Asterisk) : used to designate a variable as a pointer.

You used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is used to differentiate a variable as pointer type.

For example, when you declare a pointer to be of type int *, the compiler assumes that any address that it holds points to an integer— whether it actually does or not. That is, an int ***** pointer always assumes that it points to an int object, no matter what that piece of memory actually contains. Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.

Following are the valid pointer declaration:

1. int *ip; /* pointer to an integer */
2. double *dp; /* pointer to a double */
3. float *fp; /* pointer to a float */
4. char *ch /* pointer to a character */
5. char *I,*k;
6. int ** ptr; /*means the ptr is a pointer to a location which itself is a pointer to a variable of type integer*/

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are few important operations which we will do with the help of pointers very frequently. (a) We define a pointer variable

(b) Assign the address of a variable to a pointer and

(c) Finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations.

```
#include <stdio.h>

int main ()
```

```
{
int var = 20; /* actual variable declaration */
int *p=5224;
int *ip; /* pointer variable declaration */
ip = &var; /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var );
/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );
/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
/* Increment pointer*/
*p++;
printf("Value after increment of p:%d",*p);
return 0;
}
```

Output:

Address of var variable: bffd8b3c

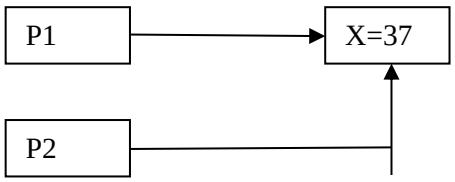
Address stored in ip variable: bffd8b3c

*Value of *ip variable: 20*

Value after increment of p: 5226

Note – More than one pointer can point to the same location. Example –

```
int x=36;
int *p1,*p2;
p1=&x;
p2=p1;
```



The content of variable x is now reachable by both pointers p1 and p2.

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have 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;

}
```

Output:

The value of ptr is 0

On 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.

To check for a null pointer you can use an if statement as follows:

if (ptr) /* succeeds if p is not null */

if (!ptr) /* succeeds if p is null */

10.5 PASSING POINTER TO A FUNCTION

C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type. Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <stdio.h>
#include <time.h>
void getSeconds(unsigned long *par);
int main ()
{
unsigned long sec;
getSeconds(&sec );
/* print the actual value */
printf("Number of seconds: %ld\n", sec );
return 0;
}
void getSeconds(unsigned long *par)
{
/* get the current number of seconds */
*par = time (NULL);
return;
}
```

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

Number of seconds: 1294450468

The function which can accept a pointer, can also accept an array as shown in the following example:

```
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);
int main ()
{
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
/* pass pointer to the array as an argument */
avg = getAverage( balance, 5 ) ;
/* output the returned value */
printf("Average value is: %f\n", avg );
return 0;
}
double getAverage(int *arr, int size)
{
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = (double)sum / size;
return avg;
}
```

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

Average value is: 214.40000

Return pointer from functions -

As we have seen in last chapter how C programming language allows to return an array from a function, similar way C allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:

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

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as static variable.

Now consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer i.e. address of first array element.

```
#include <stdio.h>
#include <time.h>
/* function to generate and retrun random numbers. */
int * getRandom( )
{
static int r[10];
int i;
/* set the seed */
srand( (unsigned)time( NULL ) );
for ( i = 0; i < 10; ++i)
{
```

```
r[i] = rand();
printf("%d\n", r[i] );
}
return r;
}
/* main function to call above defined function */
int main ()
{
/* a pointer to an int */
int *p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf("*(p + [%d]) : %d\n", i, *(p + i) );
}
return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows:

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
```

123250484

106932140

1604461820

149169022

**(p + [0]): 1523198053*

**(p + [1]): 1187214107*

**(p + [2]): 1108300978*

**(p + [3]): 430494959*

**(p + [4]): 1421301276*

**(p + [5]): 930971084*

**(p + [6]): 123250484*

**(p + [7]): 106932140*

**(p + [8]): 1604461820*

**(p + [9]): 149169022*

10.6 OPERATIONS ON POINTERS

Pointer Arithmetic –

As we have studied that C pointer is an address which is a numeric value. Therefore, we can perform arithmetic operations on a pointer just as we do on numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and - .

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

ptr++ ;

Now after the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory

location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer –

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array.

Program - Write a program to illustrate pointer increment operations.

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;
/* let us have array address in pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++;
}
return 0;
}
```

Output:

Address of var [0] = bf882b30

Value of var [0] = 10

Address of var [1] = bf882b34

Value of var [1] = 100

Address of var [2] = bf882b38

Value of var [2] = 200

Decrementing a Pointer -

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below.

Program - Write a program to illustrate pointer decrement operation.

```
#include <stdio.h>

const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i, *ptr;

/* let us have array address in pointer */

ptr = &var[MAX-1];

for ( i = MAX; i > 0; i--)

{

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, *ptr );

/* move to the previous location */

ptr--;

}
```

return 0;

}

Output:

Address of var[3] = bfedbcd8

Value of var[3] = 200

Address of var[2] = bfedbcd4

Value of var[2] = 100

Address of var[1] = bfedbcd0

Value of var[1] = 10

It may be noted here that a pointer can also be decremented or incremented to point to an immediately previous or next location of its type respectively. For Example, if the contents of a pointer p of type integer are 5224 then the content of p++ will be 5266 instead of 5225.The reason being an int is always 2 bytes size and therefore stored in two memory locations. The amount of storage taken by various types of data is tabulated in following Table10.6.1.

Table10.6.1: Various Data types Memory requirement

Type	Amount of storage
Character	1 byte
Integer	2 byte
Float	4 byte
Long	4 byte
Double	8 byte

Pointer Comparisons -

Generally, pointer comparisons are useful only when two pointers point to a common object, such as an array. 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.

You can compare two pointers in a relational expression. For instance, given two pointers p and q, the following statement is perfectly valid:

if(p < q) printf("p points to lower memory than q\n");

Program – Write a program that increment the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is & var [MAX - 1]:

```
#include <stdio.h>

const int MAX = 3;

int main ()

{

int var [] = {10, 100, 200};

int i, *ptr;

/* let us have address of the first element in pointer */

ptr = var;

i = 0;

while ( ptr <= &var[MAX - 1] )

{

printf ("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, *ptr );

/* point to the previous location */

ptr++;

i++;

}

return 0;

}
```

Output:

Address of var [0] = bfdbcb20

Value of var [0] = 10

Address of var [1] = bfdbcb24

Value of var [1] = 100

Address of var [2] = bfdbcb28

Value of var [2] = 200

10.7 POINTER AND ARRAYS

There is a close relationship between pointers and arrays. Consider this program fragment:

*char name[80], *p1;*

p1 = name;

Here, p1 has been set to the address of the first array element in *name*. To access the fifth element in *name*, you could write

name[4]

or

*(p1+4)

Both statements will return the fifth element (Remember, arrays start at 0). To access the fifth element, you must use 4 to index *name*. You also add 4 to the pointer p1 to access the fifth element because p1 currently points to the first element of *name*. (Recall that an array name without an index returns the starting address of the array, which is the address of the first element.)

The preceding example can be generalized. In essence, C provides two methods of accessing array elements:

- Pointer arithmetic
- Array indexing

Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C programmers often use pointers to access array elements.

These two versions of putstr( )— one with array indexing and one with pointers— illustrate how you can use pointers in place of array indexing. The putstr( ) function writes a string to the standard output device one character at a time.

Program - Write a program to illustrate indexes as an array.

```
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t)
putchar(s[t]);
}
/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

Pointer to Pointer

A pointer to a pointer is a form of multiple indirections, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example.

```
#include <stdio.h>
int main ()
{
int var;
int *ptr;
int **pptr;
var = 3000;
/* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

Output:

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

10.8 ARRAY OF POINTERS

Before we understand the concept of arrays of pointers, let us consider the following example which makes use of an array of 3 integers.

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

```
const int MAX = 3;

int main ()

{

int var[] = {10, 100, 200};

int i;

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

{

printf("Value of var[%d] = %d\n", i, var[i] );

}

return 0;

}
```

Output:

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

There may be a situation when we want to maintain an array which can store pointers to an int or char or any other data type available. Pointers can be arrayed like any other data type. The declaration for an int pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called var to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of var, write

```
*x[2]
```

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays i.e. simply call the function with the array name without any subscripts. For example, a function that can receive array x looks like this:

```
void display_array(int *q[])
{
int t;
for(t=0; t<10; t++)
printf("%d ", *q[t]);
}
```

Remember, q is not a pointer to integers, but rather a pointer to an array of pointers to integers.

Therefore you need to declare the parameter q as an array of integer pointers. You cannot declare q simply as an integer pointer because that is not what it is.

Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows.

```
#include <stdio.h>
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] );
}
return 0;
}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows:

```
#include <stdio.h>
const int MAX = 4;
int main ()
{
char *names[] = {
"Zara Ali",
"Hina Ali",
"Nuha Ali",
"Sara Ali",
};
int i = 0;
for ( i = 0; i < MAX; i++)
{
printf("Value of names[%d] = %s\n", i, names[i] );
}
```

```
return 0;
}
```

Output:

Value of names[0] = Zara Ali

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali

Programming Examples –

Program – Write a program to add two numbers using pointer.

```
#include<stdio.h>
int main()
{
int first,second,*p,*q,sum;
printf("Enter two integers to add \n");
scanf("%d%d", & first,&second);
p=&first;
q=&second;
sum=*p + *q;
printf("Sum of entered numbers = %d\n", sum);
return 0;
}
```

Output –

Enter two integers to add

4

5

Sum of entered numbers = 9

Program – Write a program to demonstrate, handling of pointers in C program.

```
#include<stdio.h>
int main()
{
int *pc,c;
 c=22;
 printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
pc=&c;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
c=11;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
return 0;
}
```

Output

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

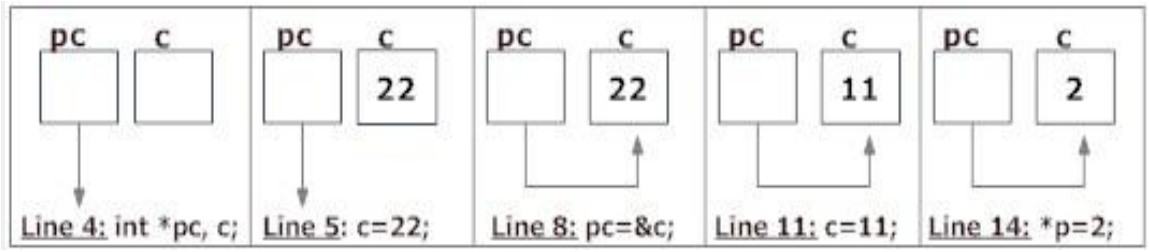
Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2



Explanation of program and figure

1. Code `int *pc, p;` creates a pointer *pc* and a variable *c*. Pointer *pc* points to some address and that address has garbage value. Similarly, variable *c* also has garbage value at this point.

2. Code `c=22;` makes the value of c equal to 22, i.e.,22 is stored in the memory location of variable *c*.

3. Code pc=&c; makes pointer, point to address of c. Note that, `&c` is the address of variable *c* (because *c* is normal variable) and *pc* is the address of *pc* (because pc is the pointer variable). Since the address of pc and address of c is same, `*pc` (value of pointer pc) will be equal to the value of *c*.

4. Code `c=11;` makes the value of *c*, 11. Since, pointer *pc* is pointing to address of *c*. Value of **pc* will also be 11.

5. Code `*pc=2;` change the address pointed by pointer *pc* to change to 2. Since, address of pointer *pc* is same as address of *c*, value of *c* also changes to 2.

Program - Write a program to accept a string and find out whether it is palindrome or not using pointer.

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

```
{
 char str[30];
 char *p,*t;
 printf("Enter any string : ");
 gets(str);
 for(p=str ; *p!=NULL ; p++);
  for(t=str, p-- ; p>=t; )
  {
    if(*p==*t)
    {
        p--;
        t++; }
    else
        break;
  }
  if(t>p)
      printf("\nString is palindrome");
  else
 printf("\nString is Not palindrome");
 return 0;
  getch();
}
```

Output –

Enter any string: SHAREMARKET

String is not palindrome

Enter any string: MADAM

String is palindrome

Program – Write a program to compute the sum of all elements stored in an array using pointers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a[10];
 int i,sum=0;
 int *ptr;
printf("Enter 10 elements:n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
ptr = a;          /* a=&a[0] */
for(i=0;i<10;i++)
{
sum = sum + *ptr;    //*p=content pointed by 'ptr'
ptr++;
}
printf("The sum of array elements is %d",sum);
}
```

Output –

```
Enter 10 elements: 11 12 13 14 15 16 17 18 19 20

The sum of array elements is 155
```

Explanation for above program –

Accept the 10 elements from the user in the array.

```
for(i=0;i<10;i++)
    scanf("%d",&a[i]);
```

We are storing the address of the array into the pointer.

```
ptr = a;
```

Now in the for loop we are fetching the value from the location pointer by pointer variable. Using De-referencing pointer we are able to get the value at address.

```
for (i=0;i<10;i++)
  {
  sum = sum + *ptr;
  ptr++;
  }
```

Suppose we have 2000 as starting address of the array. Then in the first loop we are fetching the value at 2000. i.e

```
sum = sum + (value at 2000)
  = 0   + 11
  = 11
```

In the Second iteration we will have following calculation -

```
sum = sum + (value at 2002)
  = 11  + 12
  = 23
```

10.9 SUMMARY

- Pointers are variables that contain as their values addresses of other variables.
- Pointer must be defined before they can be used.
- The definition

int *ptr;

defnes ptr to be a pointer to an object of type int and is read, “ptr is a pointer to int”. The * as used here indicates that the variable is a pointer.

- There are three values that can be used to initialize a pointer; 0, NULL, or an address. Initializing a pointer to 0 and initializing that same pointer to NULL are identical.
- The only integer that can be assigned to a pointer is 0.
- The &(address) operator returns the address of its operand.
- The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions, or to variables declared with the storage class register.
- The * operator, referred to as the indirection or dereferencing operator, returns the value of the object that its operand points to in memory. This is called dereferencing the pointer.
- When calling a function with an argument that the caller wants the called function to modify the address of the argument is passed. The called function then uses the indirection operator (*) to modify the value of the argument in the calling function.
- A function receiving an address as an argument that the caller wants the called function to modify, the address of the argument is passed. The called function then uses the indirection operator (*) to modify the value of the argument in the calling function.
- Arrays are automatically passed by reference because the value of the array name is the address of the array.
- To pass a single element of an array to a function by reference, the address of the specific array element must be passed.
- The arithmetic operations that may be performed on pointers are incrementing (++) a pointer, decrementing (--) a pointer, adding (+ or +=) a pointer and an integer, subtracting (- or -=) a pointer and an integer, and subtracting one pointer from another.
- It is possible to have array of pointers.
- It is possible to have pointers to functions.

- A pointer to a function is the address where the code for the function resides.
- Pointers to function can be passed to functions, returned from functions, stored in arrays and assigned to other pointer.

10.10 KEYWORDS

Pointer - A pointer stores the address (memory location) of another entity.

Address-of operator (&) – It gets the address of an entity.

De-reference operator (*) - It makes a reference to the referee of a pointer.

Pointer to function – A pointer to a function contains the address of the function in memory.

Arrays of Pointers - Array may contain pointers. A common use of an array of pointers is to form an array of strings, referred to simply as a string array.

10.11 REVIEW QUESTIONS

1. What is meant by a pointer?
2. Differentiate between an uninitialized pointer and a NULL pointer?
3. “Pointers always contain Integers” Comment.
4. Differentiate between (&) and (*) operator.
5. Write a short note on “Array of pointers”.
6. Write a short note on “Pointers to function”. Explain with example.
7. What is null pointer? When it is used?
8. What will be output of following program?
    a) #include<stdio.h>

```
int main(){
int a = 320;
char *ptr;
ptr =( char *)&a;
printf("%d ",*ptr);
return 0;
}
```

b)
```
#include<stdio.h>
#include<conio.h>
int main(){
void (*p)();
int (*q)();
int (*r)();
p = clrscr;
q = getch;
r = puts;
(*p)();
(*r)("cquestionbank.blogspot.com");
(*q)();
return 0;
}
```

c)
```
#include<stdio.h>
int main(){
int huge *a =(int huge *)0x59990005;
int huge *b =(int huge *)0x59980015;
if(a == b)
printf("power of pointer");
else
printf("power of c");
return 0;
}
```

9. Write a program to subtract, multiply and divide two numbers using pointer.

10. Write a program to insert and delete an element in array using pointer.

10.12 Further Readings

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[3] R.Hutchison, *Programming in C*, Tata McGraw Hill, 1990.

[4] A.K.Sharma,*Fundamentals of Computers & Programming with C*, Ganpat Rai Publications.

[5] T.D.Brown, *C for Basic Programmers*, Silicon Press, 1987.

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 11	
STRUCTURES AND UNIONS	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

11.1 OBJECTIVE

The main objective of this lesson is to introduce students about and unions and its uses in C. Difference between a variable, a array and structure is discussed. What is structure? How structure is declared? and how to define structure variables? A structure can be declared in five ways. Each of these ways is explained in detail with examples. How to make array of structures and how to use structure within structure? is explained under processing of structures. Three ways of passing structure to a functions namely passing by value, passing by address and passing as global variable is discussed. At the end of chapter union is discussed and how the memory is allocated to union along with accessing members of union is explained with examples. Difference between union and structure is done.

11.2 INTRODUCTION

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds. Structures are like arrays except that they allow many variables of different types grouped together under the same name. Structure is user defined data type which is used to store heterogeneous data under unique name. Keyword 'struct' is used to declare structure. The variables which are declared inside the structure are called as 'members of structure'.

Basically structure is a collection of variables that are functionally related to each other. Each variable that is a member of the structure has a specific type. Different members of the structure may have either the same or different types. The elements of an array, which must all be of one type. 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

Now in this book structure there are four attributes. These attributes are of different kinds. Title, author and subject is of string type and book ID is of int type. We cannot access this kind of data types together as single entity by using array only.

The entire structure may bear a name. Each member of the structure must [also] have a name. The scope of the name of a structure member is limited to the structure itself and also to any variable declared to be of the structure's type. Therefore, different structures may contain members having the same name; these may be of the same or of different types.

Uses of C structures:

- C Structures can be used to store huge data. Structures act as a database.
- C Structures can be used to send data to the printer.
- C Structures can interact with keyboard and mouse to store the data.
- C Structures can be used in drawing and floppy formatting.

- C Structures can be used to clear output screen contents.
- C Structures can be used to check computer's memory size etc.

Difference between C variable, C array and C structure:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types
- Data types can be int, char, float, double and long double etc.

Datatype	C variable		C array		C structure	
	Syntax	Example	Syntax	Example	Syntax	Example

int	int a	a = 20	int a[3]	a[0] = 10
				a[1] = 20
				a[2] = 30
				a[3] = '\0'
					struct student
					{
					int a;
					char b[10];
					}
						a = 10
						b = "Hello"

char	char b	b='Z'	char b [10]	b="Hello"		

Figure 11.2.1: Difference between C variable, C array and C structure

Defining a structure:

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

struct [structure tag]

{

```
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

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

```
struct books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

11.3 DECLARING C STRUCTURE:

Declaration of the structure merely defines the new data type; space is NOT reserved in memory as a result of the declaration. However, declaration of the structure does define how much memory is needed to store each variable subsequently declared to be of the type of the defined structure.

We can declare a structure variable in different ways:

- Declaration: Alternative 1

```
struct nameOfThisStructureType
    {
        typeOfFirstMember nameOfFirstMember;
```

```
        typeOfSecondMember nameOfSecondMember;
        typeOfThirdMember nameOfThirdMember;
        . . .
    };
struct nameOfThisStructureType variable1OfThisStructureType,
variable2OfThisStructureType,
. . . ;
```

Each such declaration MUST include the keyword struct AND the name of the user-defined structure type AND the variable name(s). Additional variable declarations can subsequently be made for this structure type.

Example:

```
struct student
{
    char first_name[20];
    char last_name[20];
    char middle_name[20];
    int student_number;
    int entrance_year;
}
struct student under_graduate_student, graduate_student;
struct student special_student;
```

- Declaration: Alternative 2

Basic named definition of the structure is effected same as for Alternative 1. In ADDITION, one or more variables can be declared within the declaration of the structure type to be of the defined type. Other variables may also be declared subsequently to be of the same type of this structure, using the

keyword struct together with the tag name and the variable names.

```
struct nameOfThisStructureType
{
        typeOfFirstMember nameOfFirstMember;
        typeOfSecondMember nameOfSecondMember;
        typeOfThirdMember nameOfThirdMember;
        . . .
} variable1OfThisStructureType, variable2OfThisStructureType, . . .;
struct nameOfThisStructureType  variable3OfThisStructureType,
variable4OfThisStructureType;
```

Example:

```
struct student
{
        char first_name[20];
        char last_name[20];
        char middle_name[20];
        int student_number;
        int entrance_year;
} under_graduate_student, graduate_student;
struct student special_student;
```

- Declaration: Alternative 3

Tag name is not assigned to the structure when the type is declared. Variables are specified within the structure declaration to be of the defined structure type. Because of the absence of a tag name for the structure type, there is no means available to ever be able to declare any other variables to be of this

same type.

```
struct /* No Name Assigned To The Type */
{
        typeOfFirstMember nameOfFirstMember;
        typeOfSecondMember nameOfSecondMember;
        typeOfThirdMember nameOfThirdMember;
        . . .
} variable1OfThisStructureType, variable2OfThisStructureType, . . .;
```

Example:

```
struct student
{
        char first_name[20];
        char last_name[20];
        char middle_name[20];
        int student_number;
        int entrance_year;
} under_graduate_student, graduate_student, special_student;
```

- Declaration: Alternative 4

Complete definition of the structure, including assignment to it of a tag name. Subsequently, the tag name is used in a typedef declaration to assign a second name (i.e., an alias) to the structure. The alias can then be used in declaring

a variable the same way as a native C type name is used, that is, without the keyword struct, i.e., just like int, char, float, etc.

```
struct nameOfThisStructureType
{
```

```
        typeOfFirstMember nameOfFirstMember;
        typeOfSecondMember nameOfSecondMember;
        typeOfThirdMember nameOfThirdMember;
        . . .
};
typedef struct nameOfThisStructureType AliasForThisStructureType;
AliasForThisStructureType variable1OfThisStructureType,
variable2OfThisStructureType, . . . ;
```

Example:

```
struct student
{
        char first_name[20];
        char last_name[20];
        char middle_name[20];
        int student_number;
        int entrance_year;
}  under_graduate_student, graduate_student;
typedef struct student StudentType;
StudentType special_student;
```

- Declaration: alternative 5

Complete definition of the structure without assignment of a tag name.The keyword typedef is used within the declaration of the structure to assign a name (i.e., an alias) to the structure. The structure itself is anonymous, and has only the alias name. The alias can be used in the same way as a native C type name is used , that is, without the keyword struct, i.e., just like int, char, float, etc.

```
typedef struct
{
        typeOfFirstMember nameOfFirstMember;
        typeOfSecondMember nameOfSecondMember;
        typeOfThirdMember nameOfThirdMember;
        . . .
} AliasForThisStructureType;
AliasForThisStructureType variable1OfThisStructureType,
variable2OfThisStructureType, . . . ;
```

Example:

```
struct student
{
        char first_name[20];
        char last_name[20];
        char middle_name[20];
        int student_number;
        int entrance_year;
}  StudentType;
StudentType under_graduate_student, graduate_student, special_student;
```

Alternative 3 is useful (example 3) because it forces all variables to be declared at structure definition time. Alternative 5 is useful (example 5) because it enables variable declarations to be made to the

structure type without use of the keyword struct.

Accessing structure members:

To access any member of a structure, we use the member access operator – dot operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure.

Program - Write a program to illustrate structure in C.

```
#include <stdio.h>
#include <string.h>
 struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
 int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */
    /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
```

```
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
    /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);
   return 0;
}
```

Output:

Book 1 title : C Programming

Book 1 author: Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700

We can also access structure member by using pointers. You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

struct Books *struct_pointer;

Now you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

struct_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

struct_pointer->title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

Program - Write a program to illustrate structure using pointers.

```
#include <stdio.h>
#include <string.h>
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};
/* function declaration */
void printBook( struct Books *book );
```

```
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */
   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );
   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );
   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
```

printf("Book book_id : %d\n", book->book_id);

}

Output:

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book_id : 6495700

11.4 PROCESSING ON STRUCTURE

Array of Structures

As you know, C Structure is collection of different datatypes(variables) which are grouped together. Whereas, array of structures is nothing but collection of structures. This is also called as structure array in C. Like Array , Array of Structure can be initialized at compile time. Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Example :

```
struct Bookinfo
{
    char[20] bname;

 int pages;
 int price;
```

}Book[100];

Here Book structure is used to Store the information of one Book. In case, if we need to store the Information of 100 books then array of Structure is used b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books. book [3] is shown in figure 11.4.1.:

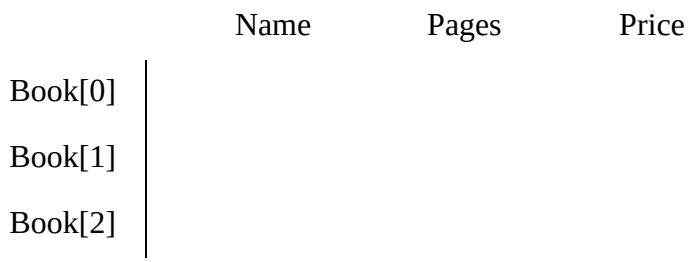


Figure 11.4.1: Memory Allocation Array of structure

Program - Write a program to illustrate array of structure.

```
#include <stdio.h>
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}book[3];
int main(int argc, char *argv[])
{
int i;
for(i=0;i<3;i++)
  {
  printf("\nEnter the Name of Book    : ");
```

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

Output:

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

--------- Book Details ----------

Name of Book : ABC

Number of Pages : 100

Price of Book : 200

Name of Book : EFG

Number of Pages : 200

Price of Book : 300

Name of Book : HIJ

Number of Pages : 300

Price of Book : 500

Structure within structure

Nested structure in C nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure. The structure variables can be normal structure variable or a pointer variable to access the data. You can learn below concepts in this section.

- Structure within structure in C using normal variable
- Structure within structure in C using pointer variable

Structure within structure in C using normal variable

This program explains how to use structure within structure in C using normal variable. "student_collage_detail' structure is declared inside "student_detail" structure in this program. Both structure variable are normal structure variables. Please note that members of "student_collage_detail" structure are accessed b 2 dots(.) operator and members of "student_detail" structure are accessed by single dot(.) operator.

Program - Write a program to illustrate structure within structure.

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

```
{
Int collage_id;
Char collage_name[50];
};
struct student_detail
 {
Int id;
Char name[20]
Float percentage;
/*structure within structure*/
struct student_collage_detail clg_data;
}stu_data;
int main()
{
struct student_detail stu_data ={1,"Raju",90.5,71145,"Anna University"};
printf(" Id is: %d \n", stu_data.id);
printf(" Name is: %s \n", stu_data.name);
printf(" Percentage is: %f \n\n", stu_data.percentage);
printf(" Collage Id is: %d \n", stu_data.clg_datacollage_id);
printf(" Collage Name is: %s \n", stu_data.clg_data.collage_name);
return 0;
}
```

Output

Id is: 1

Name is: Raju

Percentage is: 90.500000

Collage Id is: 71145

Collage Name is: Anna University

Structure within structure in C using pointer variable

This program explains how to use structure within structure in C using pointer variable. "student_collage_detail' structure is declared inside "student_detail" structure in this program. one normal structure variable and one pointer structure variable is used in this program.

Please note that combination of .(dot) and ->(arrow) operators are used to access the structure members which is declared inside the structure.

Program - Write a program to illustrate structure within structure using pointer variable.

```
#include <stdio.h>
 #include <string.h>
struct student_collage_detail
{
int collage_id;
char collage_name[50];
};
struct student_detail
{
int  id;
char name [20];
float percentage;
 /*structure within structure*/
struct student_collage_detail clg_data;
```

```
}stu_data, *stu_data_ptr;
int main ()
{
struct student_detail stu_data={1,"Raju",90.5, 71145,"Anna University"};
stu_data_ptre=&stu_data;printf("ID is: %d \n",stu_data_ptr->id);
printf(" Name is: %s \n", stu_data_ptr->name);
printf(" Percentage is: %f \n\n", stu_data_ptr->percentage);
printf(" Collage Id is: %d \n", stu_data_ptr->clg_data.collage_id);
printf(" Collage Name is: %s \n", stu_data_ptr->clg_data.collage_name);
return 0;
}
```

Output

Id is: 1

Name is: Raju

Percentage is: 90.500000

Collage Id is: 71145

Collage Name is: Anna University

11.5 PASSING STRUCTURE TO FUNCTION

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

Passing structure to function in C 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

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

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

Program - Write a program, to illustrate passing structure to function in C by value.

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

```
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.p ercentage);
}
```

Output:Id is: 1

Name is: Raju

Percentage is: 86.500000

- Passing structure to function in C by address:

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

Program - Write a program to illustrate passing structure to function in C by reference.

```
#include <stdio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
void func(struct student *record);
 int main()
{
```

```
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    func(&record);
    return 0;
}
void func(struct student *record)
{
    printf(" Id is: %d \n", record->id);
    printf(" Name is: %s \n", record->name);
    printf(" Percentage is: %f \n", record->percentage);
}
```

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

- Declare structure variable globally

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.

Program - Write a program to illustrate passing the structure globally.

```
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
struct student record; // Global declaration of structure
void structure_demo();
int main()
{
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
        structure_demo();
        return 0;
}
void structure_demo()
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

11.6 UNIONS

In C Programming we have came across Structures. C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member. Unions are similar to structure syntactically. Syntax of both is almost similar.

```
union stud              struct stud
{                       {
  int roll;               int roll;
  char name[4];           char name[4];
  int marks;              int marks;
}s1;                    }s1;
```

If we look at the two examples then we can say that both structure and union are same except Keyword. Union and structure in C are same in concepts, except allocating memory for their members. In case of union, we have collected three variables (roll, name[4], marks) of different data type under same name together whereas structure allocates storage space for all its members separately.

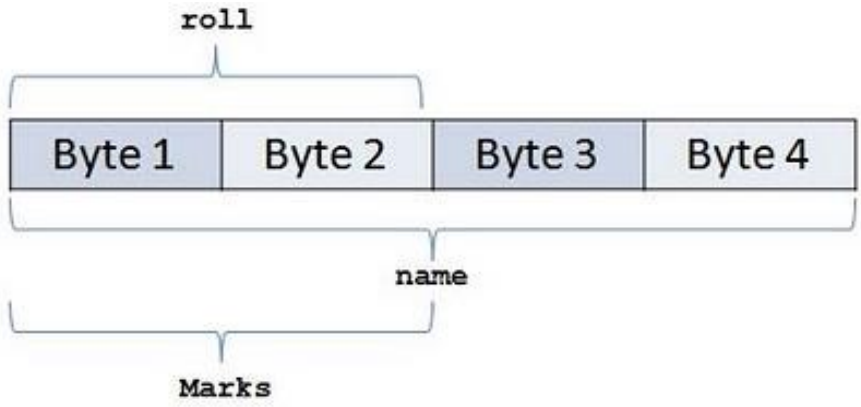


Figure 11.6.1 shows the memory allocation for different variables in a union. For the union maximum

memory allocated will be equal to the data member with maximum size. In the example character array 'name' have maximum size thus maximum memory of the union will be 4 Bytes.

Maximum Memory of Union = Maximum Memory of Union Data Member

Suppose we are accessing one of the data member of union then we cannot access other data member since we can access single data member of union because each data member shares same memory. By Using Union we can Save Lot of Valuable Space. Though unions are similar to structure in so many ways, the difference between them is crucial to understand. This can be demonstrated by this example.

Program - Write a program to print size of structure and union.

```
union job {        /*defining a union*/
char name[32];
float salary;
float worker_no;
}u;
struct job1
{
  char name[32];
  float salary;
 float  worker_no;
}s;
int main()
{
  printf("size of union = %d",sizeof(u));
  printf("\nsize of structure = %d", sizeof(s));
  return 0;
}
```

Output:

size of union = 32

size of structure = 38

There is difference in memory allocation between union and structure as suggested in above example. The amount of memory required to store a structure variables is the sum of memory size of all members as shown in Figure 11.6.2 and storage of members in unions is shown in Figure 11.6.3.

Fig11.6.2: Memory allocation in structure

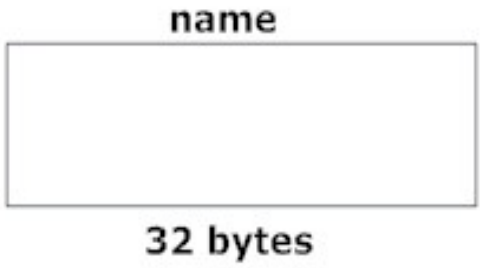


Figure 11.6.3: Memory allocation in union

Accessing Members of Union in C Programming

While accessing union, we can have access to single data member at a time. all members of structure can be accessed at any time. But, only one member of union can be accessed at a time in case of union and other members will contain garbage value. We can access single union member using following two Operators -

- Using DOT Operator (.)
- Using ARROW Operator (->)

We can access structure member using the dot operator. DOT operator is used inside printf and scanf statement to get/set value from/of union member location.

Syntax:

Variable_name.Member

OR

Pointer_Variable_name->Member

Example:

```
union emp
{
        int id;
        char name[20];
}e1;
```

id can be Accessed by – [union_Variable.member]

e1.id

Instead of maintaing the union variable suppose we store union at particular address then we can access the members of the union using pointer to the union and arrow operator.

```
union emp
{
        int id;
        char name[20];
}*e1;
```

id can be Accessed by – [union_Variable->member]

e1->id

Consider an program to understand how members of union can be accessed.

Program - Write a program to access members of unions.

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

```
union student
{
        char name[20];
        char subject[20];
        float percentage;
};
int main()
{
   union student record1;
   union student record2;
   // assigning values to record1 union variable
      strcpy(record1.name, "Raju");
      strcpy(record1.subject, "Maths");
      record1.percentage = 86.50;
      printf("Union record1 values example\n");
      printf(" Name       : %s \n", record1. name);
      printf(" Subject    : %s \n", record1.subject);
      printf(" Percentage : %f \n\n", record1.percentage);
   // assigning values to record2 union variable
      printf("Union record2 values example\n");
      strcpy(record2.name, "Mani");
      printf(" Name       : %s \n", record2.name);
      strcpy(record2.subject, "Physics");
      printf(" Subject    : %s \n", record2.subject);
      record2.percentage = 99.50;
```

```
    printf(" Percentage : %f \n", record2.percentage);
    return 0;
}
```

Output:

Union record1 values example

Name :

Subject :

Percentage : 86.500000

Union record2 values example

Name : Mani

Subject : Physics

Percentage : 99.500000

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

For record1 union variable:

“Raju” is assigned to union member “record1.name” . The memory location name is “record1.name” and the value stored in this location is “Raju”. Then, “Maths” is assigned to union member “record1.subject”. Now, memory location name is changed to “record1.subject” with the value “Maths”. (Union can hold only one member at a time). Then, “86.50” is assigned to union member “record1.percentage”. Now, memory location name is changed to “record1.percentage” with value “86.50”. Like this, name and value of union member is replaced every time on the common storage space. So, we can always access only one union member for which value is assigned at last. We can’t access other member values. That is the reason, we got only the record1.percentage value displayed in output while record1.name and record1.percentage are empty.

For record2 union variable:

If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.

Each union member are accessed in record2 example immediately after assigning values to them. If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.

Difference between structure and union:

- Structure allocates storage space for all its members separately whereas Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space.
- Structure occupies higher memory space whereas Union occupies lower memory space over structure.
- We can access all members of structure at a time but only one member of union at a time.
- Memory allocated by structure is sum of memory allocated to individual data type whereas in union total memory allocated is maximum memory of any union data member.

11.7 SUMMARY

In this lesson we have studied:

- Structure is user defined data type which is used to store heterogeneous data under unique name. Keywords 'struct' is used to declared structure.
- The scope of the name of a structure member is limited to the structure itself and also to any variable declared to be of the structure's type.
- Different structure may contain members having the same name; these may be of the same or of different types.
- typedef defines and names a new type, allowing its used throughout the program. typedefs usually occur just after the #define and #include statements in a file.
- To access the members of a structure using a pointer to that structure, you must use the ->operator.
- A structure can be passed to any function from main function or from any sub function.

- C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member. Unions are similar to structure syntactically.

11.8 KEYWORDS

Structure- Structure is user defined data type which is used to store heterogeneous data under unique name.

Struct- Keywords 'struct' is used to declare structure.

Typedef- Typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Union- C Union is also like structure, i.e. collection of different data type which are grouped together. For the union maximum memory allocated will be equal to the data members with maximum size.

11.9 REVIEW QUSTIONS

1. What do you mean by structure? how it can be declared and initialized in C?
2. Explain the usefulness of structure and union in C with their differences.
3. What do you mean by union and how it can be declared and initialized in C.
4. Differentiate between following:
   - 4.1 Array and Structure
   - 4.2 Array of structures and structure within structure
   - 4.3 Local and Global structure.
5. Write a program using structure that receives input for coordinates of a rectangles and calculates its area.
6. States with reason whether the following structure assignment is correct or not. In case it is correct, correct it.

struct pincode

```
{
        char name [5];
        int number;
} pincode;
Pincode= {{‘H’,’A’,’R’,Y’,A’,’N’,’A’},125001};
```

7. Write a C function that takes a structure variable of the following type and returns the same doubling the income value.

```
struct salary
{
        char name[14];
folat income;
};
```

11.10 FURTHER READINGS

[1] E.Balaguruswamy, *Introduction to C,* Tata McGraw Hill.

[2] T.D.Brown, *C for Basic Programmers*, Silicon Press, 1987.

[3] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[4] Peter Prinz, tony Crawford, O’reilly C *in a Nutshell*

SUBJECT: COMPUTER FUNDAMENTALS AND PROBLEM SOLVING THROUGH C	
COURSE CODE: DCA-11-T	AUTHOR: SAKSHI DHINGRA
LESSON NO. 12	
FILES IN 'C'	
REVISED / UPDATED SLM BY VINOD GOYAL	

STRUCTURE

12.1 OBJECTIVE

The basis objective of this lesson is to introduce students about file handling in C. File handling consists writing a new file, reading a pre existing file, appending matter in a file, deleting a file etc. All these operations on file are discussed in this lesson. Function for opening and closing file i.e. fopen() and fclose() respectively are discussed. Students will come ti know about different types of files. At the end of lesson, some functions to access random position in a file i.e. fseek(), ftell() and rewind are discussed.

12.2 INTRODUCTION

Storage of data in array or variables is temporary. All such data stored in arrays or variables lost when program is terminated. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, these information can be accessed using few commands. Computer stores files on secondary storage devices, especially disk storage devices. Chapter 4 explained about standard input and output devices handled by C programming language. This chapter we will see how C programmers can create, open, close text or binary files for their data storage. A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

In this lesson, we will discuss about files which are very important for storing information permanently. We store information in files for many purposes, like data processing by our programs.

12.3 TYPES OF FILE

The files can contains any type of information means they can Store the text, any Images or Pictures or any data in any Format. So that there must be Some Mechanism those are used for Storing the information, Accessing the information and also Performing Some Operations on the files.

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage. Essentially there are two kinds of files that programmers deal with:

- Text files
- Binary files

These two classes of files will be discussed in the following sections.

ASCII text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

- No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
- C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files.

They a generally processed using read and write operations simultaneously. For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many

binary files, but are rarely found in applications that process text files.

12.4 HANDLING FILES IN C

We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files. Files are not only used for data. Our programs are also stored in files. The editor which you use to enter your program and save it, simply manipulates files for you. The Unix commands cat, cp, cmp are all programs which process your files. In order to use files we have to learn about ***File I/O*** i.e. how to write information to a file and how to read information from a file. We will see that file I/O is almost identical to the terminal I/O that we have being using so far.

The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use. As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use.

Reading from or writing to a file in C requires 3 basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

Following are described the functions needed to accomplish each step.

File Operations

- Creating a new file
- Opening an existing file
- Reading from and writing information to a file
- Closing a file

Opening file

Specifying the file you wish to use is referred to as ***opening*** the file. You can use the fopen( ) function to create a new file or to open an existing file, this call will initialize an object of the type FILE, which

contains all the information necessary to control the stream. When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both. Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer.** Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable. Following is the prototype of this function call:

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

Here filename is string literal which you will use to name your file, you may use any name you wish. The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. The file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the file name.The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character '\\' must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as "\\PROJECTS\\TENLINES.TXT".

Access mode for reading and writing the files are shown in Table 12.3.1.

Table 12.3.1: Various Access Modes to Access File

Modes	Description
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 reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it

	exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"

The file <stdio.h> contains declarations for the Standard I/O library and should always be **include**d at the very beginning of C programs using files. Constants such as FILE, EOF and NULL are defined in <stdio.h>. You should note that a file pointer is simply a variable like an integer or character. It does **not** *point* to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to. A file number is used in the Basic language and a unit number is used in Fortran for the same purpose.

The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

fp = fopen(“prog.c”, “r”) ;

The above statement **opens** a file called prog.c for **reading** and associates the file pointer fp with the file. When we wish to access this file for I/O, we use the file pointer variable fp to refer to it. You can have up to about 20 files open in your program - you need one file pointer for each file you intend to use.

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:

```
FILE *fp;

fp=fopen("c:\\test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b

to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose(). The prototype of this function is:

int fclose(FILE *fp);

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. It is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program. The following example shows the usage of fclose() function.

Program – Write a program to illustrate fclose().

```
#include <stdio.h>
int main()
{
   FILE *fp;
   fp = fopen("file.txt", "w");
   fprintf(fp, "%s", "This is example of fclose");
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program, this will create a file file.txt, second it will write text line “This is example of fclose” and finally it will close the file using fclose() function.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File: fprintf( ), fputc( ), fputs( )

The functions fprintf() is the file version of printf(). The only difference while using fprintf() and is that, the first argument is a pointer to the structure FILE. fprintf() is used to write data's to a file from memory. Syntax for fprintf( ):

fprintf(file pointer,"control string",variables);

eg:- fprintf(fp,"%s",name);

The fprintf() function outputs the values of the arguments that makes up the argument list as specified in the format string to the stream pointed to by stream. The operations of the format control string and command are identical to those in printf().The return value of fprintf() is the number of characters outputted, or a negative number if an error occurs. The following example shows the usage of fputc() function.

Program – Write a program to illustrate fprint().

```
#include <stdio.h>
int main()
{
  int n;
  FILE *fptr;
  fptr=fopen("C:\\program.txt","w");
  if(fptr==NULL){
    printf("Error!");
    exit(1);
  }
  printf("Enter n: ");
```

```
  scanf("%d",&n);
  fprintf(fptr,"%d",n);
  fclose(fptr);
  return 0;
}
```

This program takes the number from user and stores in file. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open that file, you can see the integer you entered.

fputc( ): writing a character

Following is the simplest function to write individual characters to a stream:

int fputc(int c, FILE *fp);

This function converts c to an unsigned char, writes c to the output stream pointed to by stream at the current position, and advances the file position appropriately. The fputc function is identical to the putc function, but is always a function because it is not available as a macro. If the stream is opened with one of the append modes, the character is appended to the end of the stream regardless of the current file position. The following example shows the usage of fputc() function.

Program – Write a program to illustrate fputc() function.

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  int ch;
  fp = fopen("file.txt", "w+");
  for( ch = 33 ; ch <= 100; ch++ )
  {
    fputc(ch, fp);
```

```
   }


   fclose(fp);
   return(0);
}
```

Let us compile and run the above program, this will create a file file.txt in the current directory which will have following content:

!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd

fputs( ): writing a string

You can use the following functions to write a null-terminated string to a stream:

int fputs(const char *s, FILE *fp);

The function fputs() writes the string pointed to by the s to the output stream referenced by fp. It does not write the terminating \0 at the end of the string. It returns a non-negative value on success, otherwise EOF is returned in case of any error. The fputs function has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. You can use int fprintf(FILE *fp,const char *format, ...) function as well to write a string into a file. Try the following example.

Program – Write a program to illustrate fputs().

```
#include <stdio.h>
main()
{
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
```

```
  fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file test.txt in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File: fscanf( ), fgetc( ), fgets( )

Following is the simplest function to read data from the stream and stores them into location pointed by file pointer.

int fscanf (FILE * stream, const char * format, ...);

Reads data from the stream and stores them according to the parameter format into the locations pointed by the additional arguments. The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the format string. The format-string controls the interpretation of the input fields and has the same form and function as the format-string argument for the scanf() function.

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the end-of-file. If a reading error happens or the end-of-file is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

Program – Write a program to illustrate fscanf().

```
#include <stdio.h>
int main ()
{
  char str [80];
  float f;
  FILE * pFile;
  pFile = fopen ("myfile.txt","w+");
  fprintf (pFile, "%f %s", 3.1416, "PI");
```

```
  rewind (pFile);
  fscanf (pFile, "%f", &f);
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %f and %s \n",f,str);
  return 0;
}
```

This sample code creates a file called myfile.txt and writes a float number and a string to it. Then, the stream is rewinded and both values are read with fscanf. It finally produces an output similar to:I have read: 3.141600 and PI

fgetc( ): read single character from a file

Following is the simplest function to read a single character from a file:

int fgetc(FILE * fp);

Returns the character currently pointed by the internal file position indicator of the specified file pointer fp. The internal file position indicator is then advanced to the next character. If the stream is at the end-of-file when called, the function returns EOF and sets the end-of-file indicator for the stream (feof).

If a read error occurs, the function returns EOF and sets the error indicator for the stream (ferror). fgetc and getc are equivalent, except that getc may be implemented as a macro in some libraries.

Program – Write a program to illustrate fgetc().

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  int c;
  int n = 0;
  fp=fopen ("myfile.txt","r");
```

```
 if (fp==NULL) perror ("Error opening file");
 else
 {
  do {
    c = fgetc (fp);
    if (c == '$') n++;
  } while (c != EOF);
  fclose (fp);
  printf ("The file contains %d dollar sign characters ($).\n",n);
 }
 return 0;
}
```

This program reads an existing file called myfile.txt character by character and uses the n variable to count how many dollar characters ($) the file contains.

fgets( ): read string from a file

The following functions allow you to read a string from a stream:

char *fgets(char *buf, int n, 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 new line character. Notice that fgets is quite different from gets: not only fgets accepts a stream argument, but also allows to specify the maximum size of str and includes in the string any ending newline character.

Program – Write a program to illustrate fgets().

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

```
int main()
{
  FILE * fp;
  char mystring [100];
  fp = fopen ("myfile.txt" , "r");
  if (fp == NULL) perror ("Error opening file");
  else {
   if ( fgets (mystring , 100 , fp) != NULL )
     puts (mystring);
   fclose (fp);
  }
  return 0;
}
```

This example reads the first line of myfile.txt or the first 99 characters, whichever comes first, and prints them on the screen.

You can also use int fscanf(FILE *fp, const char *format, ...) function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>
main()
{
  FILE *fp;
  char buff[255];
  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buff);
  printf("1 : %s\n", buff );
```

```
   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );
   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in previous section and produces following result:

1 : *This*

2: is testing for fprintf...

3: This is testing for fputs...

In this program, first fscanf() method read just This because after that it encountered a space, second call is for fgets() which read the remaining line till it encountered end of line. Finally last call fgets() read second line completely.

A buffer is a temporary holding area in memory which acts as an intermediary between a program and a file or other I/0 device. Information can be transferred between a buffer and a file using large chunks of data of the size most efficiently handled by devices like disc drives. Typically, devices like discs transfer information in blocks of 512 bytes or more, while program often processes information one byte at a time. The buffer helps match these two desperate rates of information transfer. On output, a program first fills the buffer and then transfers the entire block of data to a hard disc, thus clearing the buffer for the next batch of output.

The various example which we have discussed till now, represents sequential access in which reading and writing is according to sequence. But sometime it is required to access the random positions of the file. This can be done by mainly three functions:

- fseek( )
- ftell( )

- rewind( )

Let us discuss each of these functions in detail.

fseek( )

The C library function fseek() sets the file position of the stream to the given offset. Following is the declaration for fseek() function:

int fseek(FILE *stream, long int offset, int whence)

where:

- stream -- This is the pointer to a FILE object that identifies the stream.
- offset -- This is the number of bytes to offset from whence.
- whence -- This is the position from where offset is added. It is specified by one of the following constants shown in Table 12.4.1.

Table 12.4.1: Constants that specifies whence

Constant	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

This function returns zero if successful, else it returns nonzero value.

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

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

  FILE *fp;
  fp = fopen("file.txt","w+");
```

```
   fputs("This is tutorialspoint.com", fp);
   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Langauge", fp);
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program, this will create a file file.txt with the following content. Initially program creates file and writes This is tutorialspoint.com but later we reset the write pointer at 7th position from the begining and use puts() statement which over-write the file with the following output:

This is C Programming Langauge

ftell( ):

The C library function long int ftell(FILE *stream) returns the current file position of the given stream.

Following is the declaration for ftell() function.

long int ftell(FILE *stream)

where stream is the pointer to a FILE object that identifies the stream.

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

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

```
#include <stdio.h>
int main ()
{
   FILE *fp;
   int len;
   fp = fopen("file.txt", "r");
```

```
  if( fp == NULL )
  {
    perror ("Error opening file");
    return(-1);
  }
  fseek(fp, 0, SEEK_END);
  len = ftell(fp);
  fclose(fp);
  printf("Total size of file.txt = %d bytes\n", len);
  return(0);
}
```

Assuming we have a text file file.txt, which has the following content:

This is C programing

Let us compile and run the above program, this will produce the following result:

Total size of file.txt = 27 bytes

rewind( )

The C library function void rewind(FILE *stream) sets the file position to the beginning of the file of the given stream. Following is the declaration for rewind() function.

void rewind(FILE *stream)

where stream is the pointer to a FILE object that identifies the stream.

This function does not return any value.

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

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

```
{
   FILE *fp;
   int ch;
   fp = fopen("file.txt", "r");
   if( fp != NULL )
   {
      while( !feof(fp) )
      {
         ch = fgetc(fp);
         printf("%c", ch);
      }
      rewind(fp);
      while( !feof(fp) )
      {
         ch = fgetc(fp);
         printf("%c", ch);
      }
      fclose(fp);
   }
   return(0);
}
```

Assuming we have a text file file.txt having the following content:

This is C programming

Now let us compile and run the above program, this will produce the following result:

This is C programming

This is C programming

12.5 THE STANDARD HEADER FILES

A header is a file with extension.h which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files:

- Files written by the programmer
- Files that are installed in compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive #include like you have seen inclusion of stdio.h header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source file, specially if we have multiple source file comprising program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototype in header files and include that header file wherever it is required.

Both user and system header files are included using the preprocessing directive #include. It has following two forms:

#include <file>

This form is used for system header files. It searches for a file named file in a slandered list of system directories. You can prepend directories to this with the –I option while compiling your source code.

#include “file”

This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the –I option while compiling your source code.

The #include directive works by directing the C preprocessor to scan the specified files as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the including file, followed by the output that comes from the text after the #include directive. For example, if you have a header file header.h as follows:

char*test (void);

And a main program called program.c that uses the header file, like this:

int x;

#include "header.h"

int main (void)

{

puts (test ());

}

The compiler will see the same token stream as it would if program.c read

int x;

*char *test (void);*

int main (void)

{

puts (test());

}

Once –Only Headers

If a header file happens to be included twice, the compiler will process its contest twice and will result an error. The standard way to prevent this is to enclose the entire real contest of the file in a conditional, like this;

#ifndef HEADER_FILE

#define HEADER_FILE

The construct is commonly known as a wrapper #indef. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometime it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could this with series of conditional as follows:

#if SYSTEM_1

include 'system_1.h"

#elif SYSTEM_2

#include "system_2.h"

#elifSYSTEM_3

….

#endif

But as it grows, it becomes tedious instead the preprocessor offers the to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of#include, you simply put a macro name there instead:

#define SYSTEM_H"system_1.h"

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the #include had been written that way originally. SYSTEM_H could be defined by your Makefile with a –D option.

12.6 UNFORMATTED DATA FILES

Some applications involve the use of data files to store block of data, where each block consists of a fixed number of contiguous bytes. Each block will generally represent a complex data structure, such as a structure or an array. For example, a data file may consist of multiple structure having the same composition, or it may contain multiple arrays of the same type and size. For such applicators it may be

desirable to read the entire block from the data, or write the entire block to the data file, rather than reading or writing the individual components (i.e., structure members of array elements) within each block separately. The library functions fread and fwrite and intended to be used in situations of this type. There functions are often referred to as unformatted read and write functions. Similarly, data files of this type are often referred to as unformatted data file.

Let us discuss both these data files in detail.

fread ()

In the C Programming Language, the fread function read nmemb elements (each element is the number of bytes indicated by size) from the stream pointed to by stream and stores them in ptr. Following is the declaration for fread() function:

Size_t fread(void*ptr,size_t size, size_t nmemb, FILE*stream)

where

ptr -This is the pointer to a block of memory with a minimum size of size*nmemb bytes.

size -This is the size in bytes of each element to be read.

nmemb - This is the number of elements, each one with a size of size bytes.

stream - This is the pointer to a FILE object that specifies an input stream

The total number of elements successfully read is returned as a size_t object, which is an integral data type. IF this number differs from the nmemb parameters, either an error occurred or the End Of File was reached. The required header for the fread function is #include<stdio.h>.The following example shows the usage of fread() function.

Program: Write a program to illustrate fread() function

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

```
FILE *fp;
char c[] =”this is computer programming “;
char buffer[20];
fp = fopen(“file.text”,”w+”);          /*open file for both reading and writing*/
fwrite(c,strlen(c ) +1,1,fp);          /*Write data to the file */
fseek(fp, SEEK_SET, 0);                /*Seek to the beginning of the file*/
fread(buffer, strlen (C )              /*Read and display data */
printf(“%s/n”, buffer);
fclose(fp);
return(0);
}
```

Let us compile and run the above program, this will create a file. Text and write content. Next we use fseek() function to reset writing pointer to the beginning of the file and ready the file content which is as follows:

This is computer programming

fwrite()

In the C Programming Language, the fwrite function writes nmemb elements (each element is the number of bytes indicated by size) from ptr to the stream pointed to by stream.Delcaration

Following is the declaration for fwrite() function.

```
size_t fwrite(const void*ptr,size_t size, size_t nmemb, FILE *stream)
```

Where

ptr – This is the pointer to the array of elements to be written.

size —This is the size in bytes of each elements to be written.

nmemb — This is the number of elements, each one with a size of size bytes.

stream — This is the pointer to a FILE object that specifies an output stream.

This function returns the total number of elements successfully written is returned as a size _t object, which is an integral data type. If this number differs from the nmemb parameters, it will show an error.

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

Program: Write a program to illustrate fwrite() function.

```
#include<stdio.h>
int main ()
{
FILE *fp;
char str[] =”This is computr programming”;
fp =fopen(“file.text” , “w”);
fwrite (str ,1 , sixeof(str) ,fp);
fclose(fp);
return(0)
}
```

Let us compile and run the above program, this will create a file.text which will have following content:

This is computer programming

12.7 SUMMARY

- A file represents a sequence of bytes, does not matter if it is a text file or binary file.
- A text file can be a stream of character that a computer can process sequentially.
- The primary difference between manipulation files and doing terminal I/O is that we must specify in our programs which files we wish to use.
- fopen() function is used to create a new file or to open an existing file.
- The file <stdio.h> contains declarations for the Standard I/O library and should always be included at the very beginning of C programs using files.

- Closing a file is performed using library function fclose().
- fprintf() is used to write data's to a file from memory. fprintf() is the version of prinf(). The only difference while using fprintf() is that the first argument is a pointer to the structure FILE.
- fputc() is simplest function to write individual character to a stream.
- fputs() functions is used to write a null-terminated string to a stream.
- fscanf() is function to read to data from the stream and store them into location pointed by file pointer.
- fgetc() is the function to read a single character from a file.
- fgets() function reads a string from file and copies it in a string variable lying in memory.
- fseek() function is used to point character pointer to given location.
- ftell () function is used to return the current position to the character pointer in the given file.
- rewind() function is used to move the character pointer to be beginning of file.
- fread() function reads nmemb elements from the stream pointed to by stream and store them in ptr.
- fwrite function write nmemb elements from ptr to the stream pointed to by stream.

12.8 KEYWORDS

File - is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind.

<stdio.h>- contains declarations for the Standard I/O library and should always be included at very beginning of C programs using files.

ferror() -If a reading errors happens or the end-of-file is reached while reading, the proper indicator is set (feof or ferror).

Buffer- A buffer is a temporary holding area in memory which acts as an intermediary between a program and a file or other I/O device.

#include directive- It works by directing the C preprocessor to scan the specified files as input before continuing with the rest of the current source file.

12.9 REVIEW QUESTIONS

1. What do you mean by file handling? Explain the concept of file handling in C.
2. Explain the various types of file and their access mechanism.
3. Explain:
   - 3.1 fprintf( ),fpuctc( ), fputs( )
   - 3.2 fscanf(), fgetc(), fgets()
4. Write a C program to emulate a database management system. The program should allow the user to create tables, insert records in them, delete records from them and view records.
5. Write a C program that reads a data file one character at a time and prints the frequency of letters in the file.
6. Write a C program to demonstrate the difference between opening a data file in following pair of modes.
   - 6.1 “w” and “w++”
   - 6.2 “a” and “ab”
7. Explain the following functions with example:

   fseek(), ftell(), rewind(), fgets(), fputs(), fgetc(), fputc().

12.10 FURTHER READING

[1] Brain W.Keringhan and Dennis M.Ritchie, The C Programming Language, Prentice Hall, 1988

[2] Peter Prinz, Tony Crawford, O’Reilly C in a Nutsheel

[3] Greg W Scragg, Genesco Suny, Problem Solving with computers, jones and Bartlett, 1997

[4] R.Hutchison, Programming in C, tata McGraw Hill. 1990

NOTES

NOTES

NOTES