

JUST BEFORE CAMPUS
JBC – T1
(Developing Programming Logic)



Anukampa Behera

CITZEN

JUST BEFORE CAMPUS
JBC – T1
(Developing Programming Logic)

By
Anukampa Behera
CITZEN

First Edition 2016

Copyright © CITZEN

All Rights Reserved. No part of this publication can be stored in any retrieval system or reproduced in any form or by any means without the prior written permission of the publishers.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The Author and Publisher of this book have tried their best to ensure that the programmes, procedures and functions described in the book are correct. However, the author and the publishers make no warranty of any kind, expressed or implied, with regard to these programmes or the documentation contained in the book. The author and publishers shall not be liable in any event of any damages, incidental or consequential, in connection with, or arising out of the furnishing, performance or use of these programmes. Product names mentioned are used for identification purposes only and may be trademarks of their respective companies.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Published by :

CITZEN

787/A, Lane-III, Jayadev Vihar, Bhubaneswar

Preface

This book covers 3 subjects - C, C++ and Data structure and is not for first time learners. It has been written to guide students preparing to seek job in IT sector. A basic knowledge of the subjects is a prerequisite to effectively use this book. The many years that I have spent in Training the students to successfully get placed has given me the necessary insight and the motivation to write this book.

These three subjects challenge the students in three forms. (1) In the form of MCQs (2) write a program (3) in the form of conceptual questions in verbal interview. In each chapter while discussing the various concepts I have tried to address all the three types of challenges that a student is expected to handle.

While planning for this book, my thought went to the glut of books available in the market teaching programming in C, C++. While many of those books are very good and they teach the language, all right, but most of them I find wanting in developing the programming skills amongst the students. We need to understand here that when we seek job in IT sector as a developer, our primary responsibility is to be able to write reliable and maintainable codes that can be easily understood and followed by others. I have tried to cover the different types of programming techniques explaining them with examples. The exercises have a variety of questions that I expect the readers to solve by themselves and gain the necessary confidence in writing clean and good codes.

This book is to be treated as a handbook based upon which a student can do a thorough preparation towards IT fresher's recruitments. It has been designed to assist and guide students in developing an in-depth understanding of the subjects and the confidence to write programs. While writing this book, my assumption has been that the reader has a basic knowledge of the subjects and programming, though that shouldn't be considered as a restriction. I encourage the readers to analyse and understand each topic and while doing so, any queries that they may have, please mail them to me and get them clarified.

Anukampa Behera

anukampa@citizen.net

About the Author

Anukampa Behera is a Software Trainer by profession. She founded her Organisation, CITZEN, in the year 2004 where she has been Training & Developing students into IT Professionals who are successfully performing in the Industry. She also happens to be the 1st Pre-Placement IT Trainer of Odisha specializing in training students to crack the Campus Interviews. With over 18 years of experience, of which more than 12 years in coaching students for IT recruitments, she brings into her class a wealth of experience and expertise that can very seldom be matched.

She regularly guides students in their Project Work in various Technologies like Android, Perl, Python, Java, Oracle, PHP, C and C++. Simply knowing to write a program is not enough. One must know how to design and develop the Project scientifically in a professional environment. Training in her classes goes beyond writing programs and designing click buttons. Her classes teach the students the most effective use of technologies for designing Projects with greater ease and efficiency.

She has MTECH degree and is an Oracle Certified Professional. She writes blogs on various topics and has published a book for the Oracle developers called “Oracle 10g: A HandBook”.

Content

Sl. no	Content	Page no
C Programming		
1	Introduction to C Language	20
	1.1. Structure of a C Program	
	1.2. Some simple programs	
	1.3. Desirable program characteristics	
2	Elements of C Programming	25
	2.1 Basics of a typical C environment	
	2.2 Anatomy of a C Program	
	2.3 Basic Conversion Characters for Data Output	
	2.4 Expression	
	2.5 Statements	
	2.6 Operators	
3	Control Statements	48
	3.1 Conditional Execution of Statements	
	3.2 Iterative Control Statements	
4	Functions and storage class	61
	4.1 Basics of function	
	4.2 Calling a function – call by value	
	4.3 Calling a function – passing reference argument	
	4.4 Storage classes	
	4.5 Recursion	
5	Arrays	82
	5.1 Defining an array	
	5.2 Processing an array	
	5.3 Sorting	
	5.4 Multidimensional arrays	
	5.5 Arrays and strings	
6	Pointer	99
	6.1 Declaration of pointers	

6.2	Pointers and one dimensional arrays	
6.3	Dynamic memory allocation	
6.4	Multi-dimensional arrays using pointers	
6.5	Command line arguments	
6.6	Pointers to functions	
7	User defined datatype	119
7.1	Declaring a structure	
7.2	Accessing structure elements	
7.3	Nested structures	
7.4	Self referential structure	
7.5	Single Linked list	
7.6	Union	
7.7	Enumerated data type	
8	Input and output in C	142
8.1	Disk I/O Functions	
8.2	File opening modes	
8.3	Program to copy a file	
8.4	Formatted disk I/O	
8.5	Text versus binary file	
C++ Programming		
9	Introduction to C ++ Language	170
9.1	C Vs. C++	
10	Programming in C++	175
10.1	Basics of a typical C++ environment	
10.2	Anatomy of a C++ program	
10.3	Data types	
10.4	Keywords	
10.5	Constants	
10.6	Expressions	
10.7	Operators	
11	Conditional and Iterative Construct	191
11.1	if...else	
11.2	switch...case	

11.3	looping	
11.4	Break and Continue	
12	Function and storage types	197
12.1	Basics of function	
12.2	Calling a function	
12.3	default values	
12.4	Inline functions	
12.5	Recursion	
12.6	Function overloading	
12.7	Type safe linkage	
12.8	Linkage Specification	
12.9	Scope/life of a variable	
12.10	Storage Classes	
12.11	Type Qualifier	
12.12	Dynamic memory allocation	
13	OOPs	216
13.1	Object Oriented Programming Structure	
12.4	Constructor	
12.4	Destructor	
14	Inheritance	224
14.1	Implementation of inheritance in C++	
14.2	overloading base class member function	
14.3	multiple inheritance	
14.4	Ambiguity in multiple inheritance	
14.5	Ambiguity in hybrid inheritance	
14.6	Multipath inheritance	
15	Binding	238
15.1	Binding	
15.2	Virtual Function	
15.3	Pure Virtual Function	
15.4	Abstract class	
15.5	void pointer	
16	Polymorphism	243
16.1	Concept of polymorphism	

16.2 Static polymorphism	
16.3 Dynamic polymorphism	
16.4 Friend function	
16.5 Friend class	
16.6 Operator overloading	
17 File Handling	250
17.1 Data transmission in a program	
17.2 class hierarchy	
17.3 Single file stream working on separate files	
17.4 File input and output using Abstract datatypes	
17.5 Binary I/O	
17.6 Random Access	

Prologue

What is programming?

We define *programming*, as a general human activity, to mean the act of extending or changing a system's functionality.

Program development cycle –

1. Understand the problem
2. Plan the logic
3. Code the program
4. Translate the program into machine language using software (a compiler and/or interpreter)
5. Test the program
6. Deploy the program (make available for use)
7. Maintain the program

A detailed Description:

1. Understanding the Problem

Professional computer programmers write programs to satisfy the needs of others, called users or end users. Examples of end users include a Human Resources department that needs a printed list of all employees, a Billing department that wants a list of clients who are 30 or more days overdue on their payments, and an Order department that needs a Web site to provide buyers with an online shopping cart. Because programmers are providing a service to these users, programmers must first understand what the users want. When a program runs, you usually think of the logic as a cycle of input-processing-output operations, but when you plan a program, you think of the output first. After you understand what the desired result is, you can plan the input and processing steps to achieve it.

Suppose the director of Human Resources says to a programmer, "Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner." On the surface, this seems like a simple request. An experienced programmer, however, will know that the request is incomplete. For example, you might not know the answers to the following questions about which employees to include:

- Does the director want a list of full-time employees only, or a list of full- and part-time employees together?
- Does she want to include people who have worked for the company on a month-to-month contractual basis over the past five years, or only regular, permanent employees?
- Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cut-off date?
- What about an employee who worked three years, took a two-year leave of absence, and has been back for three years?
- The programmer cannot make any of these decisions; the user (in this case, the Human Resources director) must address these questions.

- More decisions still might be required. For example: What data should be included for each listed employee? Should the list contain both first and last names? Social Security numbers? Phone numbers? Addresses?
- Should the list be in alphabetical order? Employee ID number order? Length-of-service order? Some other order?
- Should the employees be grouped by any criteria, such as department number or years of service?

Several pieces of documentation are often provided to help the programmer understand the problem. Documentation consists of all the supporting paperwork for a program; it might include items such as original requests for the program from users, sample output, and descriptions of the data items available for input.

Fully understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, users may not really know what they want, and users who think they know frequently change their minds after seeing sample output. A good programmer is often part counsellor, part detective!

In short, understanding the problem is

- One of the most difficult aspects of programming
- Users (end users) – People for whom program is written
- Documentation – Supporting paperwork for a program
- flowchart / pseudo code
- hierarchy chart (aka structure chart or VTOC)
- screen / printer spacing chart
- end user instructions

2. Plan the Logic

The heart of the programming process lies in planning the program's logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include and how to order them. You can plan the solution to a problem in many ways. The two most common planning tools are flowcharts and pseudocode. Both tools involve writing the steps of the program in English, much as you would plan a trip on paper before getting into the car or plan a party theme before shopping for food and favours.

You may hear programmers refer to planning a program as “developing an algorithm.” An algorithm is the sequence of steps necessary to solve any problem. In addition to flowcharts and pseudocode, programmers use a variety of other tools to help in program development. One such tool is an IPO chart, which delineates input, processing, and output tasks. Some object-oriented programmers also use TOE charts, which list tasks, objects, and events. The programmer shouldn't worry about the syntax of any particular language during the planning stage, but should focus on figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario. The process of walking through a program's logic on paper before you actually write the program is called desk-checking. You will learn more about planning the logic.

In short, plan the logic is

- Heart of the programming process
- Most common logic planning tools – Flowcharts – Pseudocode – hierarchy chart
- Desk-checking – Walking through a program's logic on paper before you actually write the program

Code the Program

After the logic is developed, only then can the programmer write the source code for a program. Hundreds of programming languages are available. Programmers choose particular languages because some have built-in capabilities that make them more efficient than others at handling certain types of operations. Despite their differences, programming languages are quite alike in their basic capabilities—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with proper punctuation and the correct spelling of commands—in other words, using the correct syntax.

Some experienced programmers can successfully combine logic planning and program coding in one step. This may work for planning and writing a very simple program, just as you can plan and write a postcard to a friend using one step. A good term paper or a Hollywood screenplay, however, needs planning before writing—and so do most programs.

Which step is harder: planning the logic or coding the program? Right now, it may seem to you that writing in a programming language is a very difficult task, considering all the spelling and syntax rules you must learn. However, the planning step is actually more difficult. Which is more difficult: thinking up the twists and turns to the plot of a best-selling mystery novel, or writing a translation of an existing novel from English to Spanish? And who do you think gets paid more, the writer who creates the plot or the translator?

In short, code the program involves the following:

- Hundreds of programming languages are available – Choose based on:
- features
- organizational requirements
- Most languages are similar in their basic capabilities
- Easier than planning step

3. Using Software to Translate the Program into Machine Language

Even though there are many programming languages, each computer knows only one language—its machine language, which consists of 1s and 0s. Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively. Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer's English-like high-level programming language into the low-level machine language that the computer understands. When you learn the syntax of a programming language, the commands work on any machine on which the language software has been installed. However, your commands then are translated to machine language, which differs in various computer makes and models

If you write a programming statement incorrectly (for example, by misspelling a word, using a word that doesn't exist in the language, or using "illegal" grammar), the translator program doesn't know how to proceed and issues an error message identifying a syntax error.

Although making errors is never desirable, syntax errors are not a major concern to programmers, because the compiler or interpreter catches every syntax error and displays a message that notifies you of the problem. The computer will not execute a program that contains even one syntax error.

Typically, a programmer develops logic, writes the code, and compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors and compiles the program again. Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler. For example, if you could use an English compiler and submit the sentence—"The dg chase the cat.", the compiler at first might point out only one syntax error. The second word, dg, is illegal because it is not part of the English language.

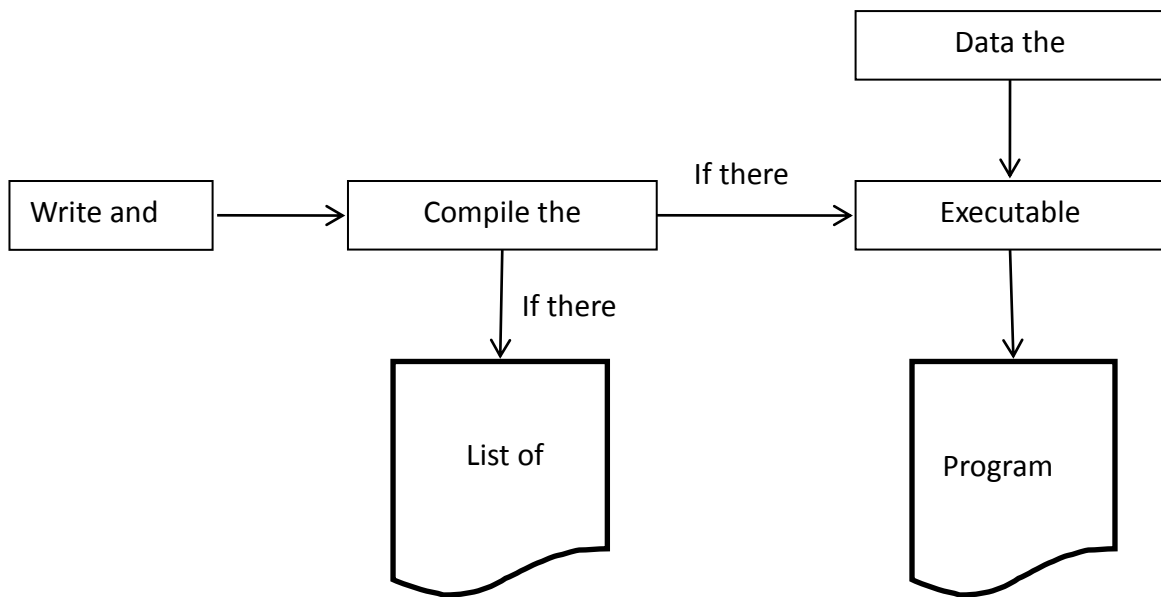
Only after you corrected the word to dog would the compiler find another syntax error on the third word, chase, because it is the wrong verb form for the subject dog. This doesn't mean chase is necessarily the wrong word. Maybe dog is wrong; perhaps the subject should be dogs, in which case chase is right. Compilers don't always know exactly what you mean, nor do they know what the proper correction should be, but they do know when something is wrong with your syntax.

In short, Using Software to Translate the Program into Machine Language is

- Translator program – Compiler and/or interpreter – Changes the programmer's English-like high-level programming language into the low-level machine language
- Syntax error – Misuse of a language's grammar rules – Programmer corrects listed syntax errors – Might need to recompile the code several times
- misspelled variable names
- unmatched curly braces

Languages / File Types

- Source language – Java, C++, Visual Basic, etc. – file types (extensions):
 - java
 - cpp
 - vb
- Compiled language (destination language) – other high-level language (cross compiler) – machine language – virtual machine language (intermediate language)
 - Java class file (.class)
- MSIL (Microsoft Intermediate Language) – files types (extensions):
 - class
 - msil
 - obj
 - exe



4. Test the Program

A program that is free of syntax errors is not necessarily free of logical errors. A logical error results when you use a syntactically correct statement but use the wrong one for the current context. For example, the English sentence The dog chases the cat, although syntactically perfect, is not logically correct if the dog chases a ball or the cat is the aggressor.

Once a program is free of syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct. Let's recall the number-doubling program:

```

input myNumber
set myAnswer = myNumber * 2
output myAnswer

```

If you execute the program, provide the value 2 as input to the program, and the answer 4 is displayed, you have executed one successful test run of the program. However, if the answer 40 is displayed, maybe the program contains a logical error. Maybe the second line of code was mistyped with an extra zero, so that the program reads:

```

input myNumber
set myAnswer = myNumber * 20
output myAnswer

```

Placing 20 instead of 2 in the multiplication statement caused a logical error. Notice that nothing is syntactically wrong with this second program—it is just as reasonable to multiply a number by 20 as by 2—but if the programmer intends only to double myNumber, then a logical error has occurred.

The process of finding and correcting program errors is called debugging. You debug a program by testing it using many sets of data. For example, if you write the program to double a number, then enter 2 and get an output value of 4, that doesn't necessarily mean you have a correct program. Perhaps you have typed this program by mistake:

```

input myNumber
set myAnswer = myNumber + 2
output myAnswer

```

An input of 2 results in an answer of 4, but that doesn't mean your program doubles numbers—it actually only adds 2 to them. If you test your program with additional data and get the wrong answer—for example, if you enter 7 and get an answer of 9—you know there is a problem with your code.

Selecting test data is somewhat of an art in itself, and it should be done carefully. If the Human Resources department wants a list of the names of five-year employees, it would be a mistake to test the program with a small sample file of only long-term employees. If no newer employees are part of the data being used for testing, you do not really know if the program would have eliminated them from the five-year list. Many companies do not know that their software has a problem until an unusual circumstance occurs—for example, the first time an employee has more than nine dependents, the first time a customer orders more than 999 items at a time, or when the Internet runs out of allocated IP addresses, a problem known as IPv4 exhaustion.

In short, testing the program involves:

- **Logical error** – Use a syntactically correct statement but use the wrong one for the current context
- **Run-time error** – program ends abnormally when the user runs the program (sometimes or every time)
- **Test Data** – Execute the program with some sample test data to see whether the results are logically correct

5. Deploy the Program Make the Program Available for Use

Once the program is thoroughly tested and debugged, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user's request for a special list. However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program, users must be trained to understand the output, or existing data in the company must be changed to an entirely new format to accommodate this program. Conversion, the entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.

In short, deployment involves:

- Process depends on program's purpose – May take several months
- Conversion – Entire set of actions an organization must take to switch over to using a new program or set of programs

6. Maintain the Program

After programs are put into production, making necessary changes is called maintenance. Maintenance can be required for many reasons: for example, because new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications. Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work. When you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production. If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

In short, maintenance is:

- Making changes after program is put into production
- Common first programming job – Maintaining previously written programs
- Make changes to existing programs – Repeat the development cycle

Using Pseudocode Statements and Flowchart Symbols

When programmers plan the logic for a solution to a programming problem, they often use one of two tools: pseudocode (pronounced sue-doe-code) or flowcharts. Pseudocode is an English-like representation of the logical steps it takes to solve a problem. Pseudo is a prefix that means false, and to code a program means to put it in a programming language; therefore, pseudocode simply means false code, or sentences that appear to have been written in a computer programming language but do not necessarily follow all the syntax rules of any specific language.

A flowchart is a pictorial representation of the same thing.

Writing Pseudocode

You have already seen examples of statements that represent pseudocode earlier in this chapter, and there is nothing mysterious about them. The following five statements constitute a pseudocode representation of a number-doubling problem:

```
start
input myNumber
set myAnswer = myNumber * 2
output myAnswer
stop
```

Using pseudocode involves writing down all the steps you will use in a program. Usually, programmers preface their pseudocode with a beginning statement like `start` and end it with a terminating statement like `stop`. The statements between `start` and `stop` look like English and are indented slightly so that `start` and `stop` stand out. Most programmers do not bother with punctuation such as periods at the end of pseudocode statements, although it would not be wrong to use them if you prefer that style. Similarly, there is no need to capitalize the first word in a sentence, although you might choose to do so. This book follows the conventions of using lowercase letters for verbs that begin pseudocode statements and omitting periods at the end of statements.

Pseudocode is fairly flexible because it is a planning tool, and not the final product. Therefore, for example, you might prefer any of the following:

Instead of `start` and `stop`, some pseudocode developers would use other terms such as `begin` and `end`. Instead of writing `input myNumber`, some developers would write `get myNumber` or `read myNumber`. Instead of writing `set myAnswer = myNumber * 2`, some developers would write `calculate myAnswer = myNumber times 2` OR `compute myAnswer as myNumber doubled`. Instead of writing `output myAnswer`, many pseudocode developers would write `display myAnswer`, `print myAnswer`, OR `write myAnswer`.

The point is, the pseudocode statements are instructions to retrieve an original number from an input device and store it in memory where it can be used in a calculation, and then to get the calculated answer from memory and send it to an output device so a person can see it. When you eventually convert your pseudocode to a specific programming language, you do not have such flexibility because specific syntax will be required. For example, if you use the C programming language and write the statement to output the answer to the monitor, you will code the following:


```
printf("%d",Write(myAnswer) ;
```

The exact use of words, capitalization, and punctuation are important in the C statement, but not in the pseudocode statement.

Some professional programmers prefer writing pseudocode to drawing flowcharts, because using pseudocode is more similar to writing the final statements in the programming language. Others prefer drawing flowcharts to represent the logical flow, because flowcharts allow programmers to visualize more easily how the program statements will connect. Especially for beginning programmers, flowcharts are an excellent tool to help them visualize how the statements in a program are interrelated.

You can draw a flowchart by hand or use software, such as Microsoft Word and Microsoft PowerPoint, that contains flowcharting tools. You can use several other software programs, such as Visio and Visual Logic, specifically to create flowcharts. When you create a flowchart, you draw geometric shapes that contain the individual statements and that are connected with arrows. (Appendix B contains a summary of all the flowchart symbols you will see in this book.) You use a parallelogram to represent an input symbol, which indicates an input operation.

You write an input statement in English inside the parallelogram, as shown in Figure 1-3.



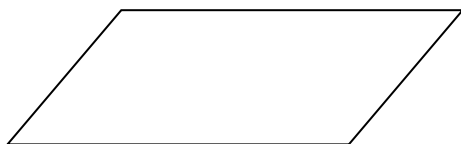
<fig -1.3 >

Arithmetic operation statements are examples of processing. In a flowchart, you use a rectangle as the processing symbol that contains a processing statement, as shown in Figure 1-4.



<fig – 1.4>

To represent an output statement, you use the same symbol as for input statements—the output symbol is a parallelogram, as shown in Figure 1-5.



<fig – 1.5>

Because the parallelogram is used for both input and output, it is often called the input/output symbol or I/O symbol. Some software programs that use flowcharts (such as Visual Logic) use a left-slanted parallelogram to represent output. As long as the flowchart creator and the flowchart reader are communicating, the actual shape used is irrelevant. This book will follow the most standard convention of using the right-slanted parallelogram for both input and output. To show the correct sequence of these statements, you use arrows, or flowlines, to connect the steps. Whenever possible, most of a flowchart should read from top to bottom or from left to right on a page. That's the way we read English, so when flowcharts follow this convention, they are easier for us to understand. To be complete, a flowchart should include two more elements: terminal symbols, or start/stop symbols, at

each end. Often, you place a word like start or begin in the first terminal symbol and a word like end or stop in the other.

Usually, a computer program will be written in some **high-level** language, whose instruction set is more compatible with human languages and human thought processes. Most of these are **general-purpose** languages such as C.

A program that is written in a high-level language must, however, be translated into machine language before it can be executed. This is known as **compilation** or **interpretation**, depending on how it is carried out. (Compilers translate the entire program into machine language before executing any of the instructions.

Interpreters, on the other hand, proceed through a program by translating and then executing single instructions or small groups of instructions.) In either case, the translation is carried out automatically within the computer. In fact, inexperienced programmers may not even be aware that this process is taking place, since they typically see only their original high-level program, the input data, and the calculated results. Most implementations of C operate as compilers.

A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, and generates a corresponding machine-language program as output. The original high-level program is called the source program, and the resulting machine-language program is called the object program. Every computer must have its own compiler or interpreter for a particular high-level language.

Section - I

Programming

with

C

Chapter – 1: Introduction to C Language

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English **keywords** such as **if**, **else**, **for**, **do** and **while**. In this respect C resembles other high-level structured programming languages such as Pascal and Fortran. C also contains certain additional features, however, that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high-level languages. This flexibility allows C to be used for **systems programming** (e.g., for writing communication protocols, kernels etc) as well as for **applications programming** (e.g., for writing a program to solve a complicated system of mathematical equations, or for writing a program to bill customers).

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive **library functions** which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes, and C interpreters are becoming increasingly common. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages. The interpreters are less efficient, though they are easier to use when developing a new program. Many programmers begin with an interpreter, and then switch to a compiler once the program has been debugged (i.e., once all of the programming errors have been removed).

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer-dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer. These library functions are relatively standardized, however, and each individual library function is generally accessed in the same manner from one version of C to another. Therefore, most C programs can be processed on many different computers with little or no alteration.

Structure of a C Program

Every C program consists of one or more modules called functions. One of the functions must be called **main**.

The program will always begin by executing the **main** function, which may access other functions. Any other function definitions must be defined separately, either ahead of or after **main**.

Each function must contain:

1. A function **heading**, which consists of the function name, followed by an optional list of **arguments**, enclosed in parentheses.
2. A list of argument **declarations**, if arguments are included in the heading.
3. A **compound statement**, which comprises the remainder of the function.

The arguments are symbols that represent information being passed between the function and other parts of the program. Arguments are passed when we call a function and **parameters** are variables created to receive these arguments when we define a function.

Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called **expression statements**) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;).

Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., /* **t h i s** is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

A Sample 'C' program:

```
/*Swap the value of two numbers */
/* TITLE (COMMENT) */
#include <stdio.h>                / * LIBRARY FILE ACCESS */
main( )                          / * FUNCTION HEADING */
{
    int inum1, inum2, itemp;      /* VARIABLE DECLARATIONS */
    printf("Enter two numbers:") /* OUTPUT STATEMENT (PROMPT) */
    scanf("%d %d", &inum1, &inum2) ; /* INPUT STATEMENT */
    itemp = inum1; /* ASSIGNMENT STATEMENT */
    inum1 = inum2; /* ASSIGNMENT STATEMENT */
    inum2 = itemp; /* ASSIGNMENT STATEMENT */
    printf ("\n Swapped Data : %d %d ", inum1, inum2) ;
    /* OUTPUT STATEMENT */
}
```

The following features should be pointed out in this last program.

1. The program is typed in lowercase. Either upper- or lowercase can be used, though it is customary to type ordinary instructions in lowercase. Most comments are also typed in lowercase, though comments are sometimes typed in uppercase for emphasis, or to distinguish certain comments from the instructions. (Uppercase and lowercase characters are not equivalent in C. Later in this book we will see some special situations that are characteristically typed in uppercase.)
2. The first line is a comment that identifies the purpose of the program.
3. The second line contains a reference to a special file (called `stdio.h`) which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler.
4. The third line is a heading for the function `main`. The empty parentheses following the name of the function indicate that this function does not include any arguments.

5. The remaining eight lines of the program are indented and enclosed within a pair of braces. These eight lines comprise the compound statement within main.
6. The first indented line is a variable declaration. It establishes the symbolic names inum1 and inum2 as integer variables.
7. The remaining six indented lines are expression statements. The second indented line (printf) generates a request for information (namely, a value for the radius). This value is entered into the computer via the third indented line (scanf).
8. The fourth, fifth and sixth indented line is a particular type of expression statement called an assignment statement. This statement causes the value stored in inum1 and inum2 to be interchanged.
9. The last indented line (printf) causes the calculated value for the area to be displayed. The numerical value will be preceded by a brief label.
10. Notice that each expression statement within the compound statement ends with a semicolon. This is required of all expression statements.
11. Finally, notice the liberal use of spacing and indentation, creating whitespace within the program. The blank lines separate different parts of the program into logically identifiable components, and the indentation indicates subordinate relationships among the various instructions. These features are not grammatically essential, but their presence is strongly encouraged as a matter of good programming practice.

Some Simple Programs

```

/*accept radius, calculate the area of a circle*/
#include <stdio.h>                / * LIBRARY FILE ACCESS */
main( )                          / * FUNCTION HEADING */
{
    int radius, area;            /* VARIABLE DECLARATIONS */
    printf("Enter radius :") /* OUTPUT STATEMENT (PROMPT) */
    scanf("%f" ,&radius) ;      /* INPUT STATEMENT */
    area= 3.14159 * radius * radius; /* ASSIGNMENT STATEMENT */
    printf ("\n Area : %f ",area) ;
    /* OUTPUT STATEMENT */
}

/*accept base, height, calculate the area of a triangle*/
#include <stdio.h>                / * LIBRARY FILE ACCESS */
main( )                          / * FUNCTION HEADING */
{
    float base, height, area;    /* VARIABLE DECLARATIONS */
    printf("Enter base and height :")
    /* OUTPUT STATEMENT (PROMPT) */
    scanf("%f %f" ,&base, &height) ; /* INPUT STATEMENT */
}

```

```

        area= 0.5 * base * height; /* ASSIGNMENT STATEMENT */
        printf ("\n Area : %f ",area) ;
        /* OUTPUT STATEMENT */
    }

/*accept temperature in Fahrenheit and convert it to centigrade*/
#include <stdio.h>                / * LIBRARY FILE ACCESS */
main( )                          / * FUNCTION HEADING */
{
    float ftemp,ctemp;           /* VARIABLE DECLARATIONS */
    printf("Enter temperature in fahrenheit :")
    /* OUTPUT STATEMENT (PROMPT) */
    scanf ("%f" ,&ftemp); /*INPUT STATEMENT*/
    ctemp = (ftemp-32) * 5.0/9; /* ASSIGNMENT STATEMENT */
    printf ("\n Temperature in Centigrade : %f ",ctemp) ;
    /* OUTPUT STATEMENT */
}

/* accept Principal, Rate of interest and Term. Find the simple
interest*/
#include <stdio.h>                / * LIBRARY FILE ACCESS */
main( )                          / * FUNCTION HEADING */
{
    float P,R,N,SI; /* VARIABLE DECLARATIONS */
    printf("Enter principal, rate of interest and term :")
    /* OUTPUT STATEMENT (PROMPT) */
    scanf ("%f %f %f" ,&P, &R, &N) ; /*INPUT STATEMENT*/
    SI = P * R * N / 100; /* ASSIGNMENT STATEMENT */
    printf ("\n Simple Interest : %f ",SI) ;
    /* OUTPUT STATEMENT */
}

```

Desirable Program Characteristics

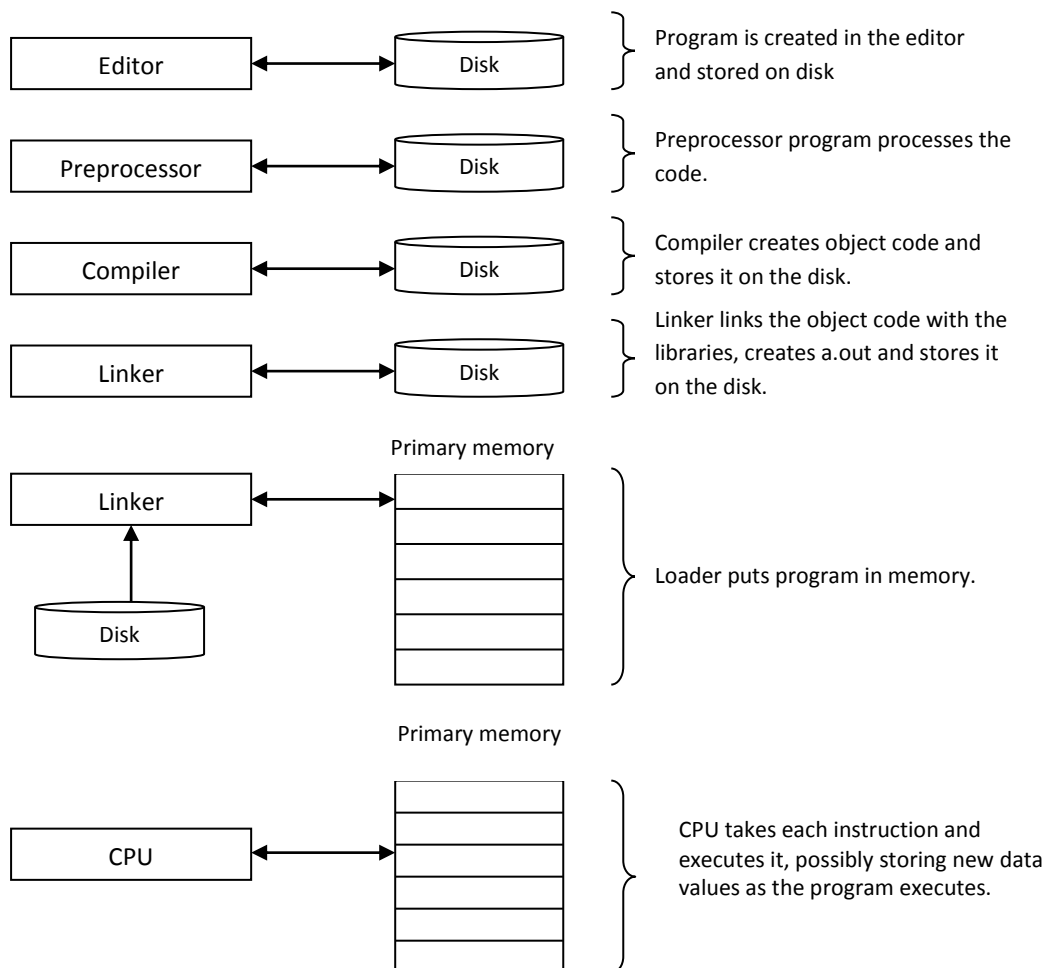
Now, let us discuss, some important characteristics of a good computer programs. These characteristics apply to programs that are written in **any** programming language, not just C. They can provide us with a useful set of guidelines later in this **book**, when we start writing our own C programs.

1. **Integrity.** This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. Thus, the integrity of the calculations is an absolute necessity in any computer program.
2. **Clarity** refers to the overall readability of the program, with particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program logic without undue effort. It should also be possible for the original author to follow his or her own program after being away from the program for an extended period of time. One of the objectives in the design of C is the development of clear, readable programs through an orderly and disciplined approach to programming.
3. **Simplicity.** The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with the overall program objectives. In fact, it may be desirable to sacrifice a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.
4. **Efficiency** is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity. Many complex programs require a trade off between these characteristics. In such situations, experience and common sense are key factors.
5. **Modularity.** Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C, such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.
6. **Generality.** Usually we will want a program to be **as** general as possible, within reasonable limits. For example, we may design a program to read in the values of certain key parameters rather than placing fixed values into the program. **As** a rule, a considerable amount of generality can be obtained with very little additional programming effort.

Chapter – II: Elements of C Programming

After a brief introduction to the C language let's now get into our real objective of writing programs using C. At this point I will be only discussing the basics of the language so that you can start writing effective programs as soon as possible. This chapter will cover the essential elements of the programs i.e. variables & constants, declarations, simple data input and output, expressions, assignments, control flow, operators and their precedence. It is very essential for you to understand these elementary things clearly to be able to write your programs effectively and independently. In the later chapters as you come across the concepts of structures, pointers, control flow statements etc. and master them, would you able to comprehend the power of C and see the beauty, elegance and conciseness of it's programs. As a beginner, let's first understand the C environment.

Basics of a Typical C Environment



Schematic Diagram explaining the execution of a C Program

The diagram in the last page is self-explanatory. When we try to execute a C program, it passes through the following phases:

1. **Edit:** The program is typed/keyed-in using an editor. Some of the compilers have their own editor like VC++. Turbo C/C++, Borland C/C++ etc, where as some compilers use Operating system's editor like ANSI C uses vi editor of UNIX operating system.
2. **Preprocessor:** Preprocessor performs macro substitution, conditional compilation, and inclusion of named files. The Preprocessor does all the prior processing and makes the code ready for compilation.
3. **Compile:** The compiler translates the entire program into Operating System understandable form (machine language) i.e., the source code (file) is compiled and an object file (.obj) is created. The object file is stored on the disk.
4. **Link:** Linker links the object code with the libraries, further divides the object code into single instructions to be executed by the processor and creates the executable (.exe) file, which can be directly executed on the operating system prompt.
5. **Load:** Any program or a set of instruction before getting processed has to be loaded or copied to the internal primary memory (RAM). The loader copies the executable file from the secondary disk to the primary memory (RAM) in order to execute it.
6. **Execute:** In the execution phase, the processor processes individual instructions and stores the results / intermediate results in registers.

Anatomy of a C program

```
#include<stdio.h>      include information about standard library

main( )                Define a function named main

{                      Statements of main are enclosed in
braces

printf("hello, world\n"); Main calls library function printf to
                        print this sequence of characters; \n
                        represents the new line character

}

/*wish.c*/             Comment entry
```

Components of a C Program

- comment
- # include directive
- printf
- scanf
- function
- data type
- variable
- operator

Comments:

Comments are used for better understanding of the program statement. The comment entries start with `/*` and ends with `*/`

Example:

```
/* This is a program in C to automate the Inventory Control System of ABC International Ltd. */
```

include Directive

`#include` is a preprocessor directive (discussed in details in later chapter), which causes the codes of one file to be included in another. Generally the `#include` statement appears at the top of the `main()` function. However, it can appear anywhere in the program. It can be written in two ways:

1. `#include <filename.h>`
2. `#include "filename.h"`

In its 1st form the directive would look for the **file** only the specified list of directories as defined by the search path to the include directory and insert the code of the **file** into the source code at the point where it appears.

In its 2nd form the directive would do the same thing but would first look for the **file** in the current directory failing which it will look for the **file** in the include directory.

printf() and scanf() functions

These functions are C library functions. `scanf()` function is used to enter input data to the computer from any standard input device and `printf()` function is used to write the output data to the standard output devices. The input/output data can be any combination of strings, single

character and numerical values. More about these functions have been discussed in the later part of the chapter.

Function

- ❑ All C programs comprise of one or more function.
- ❑ A function name and a function body define a function.
- ❑ A function name is defined by a word followed by parentheses.
- ❑ All programs must have a function called main ().
- ❑ The execution of a program starts with the main () function.
- ❑ A function body is surrounded by curly braces ({}). The braces delimit a block of program statements. Every function must have a pair of braces.

Fundamental Data Types

The fundamental data type at the lowest level, that is, those, which are used for actual data representation in memory. The fundamental data types and their functions are listed in the following table – Table 1.1.

Data type	No of bytes	Representation
Char	1	Single Character
Int	2	Integers
Float	4	Floating point numbers

Variables

Variables are fundamental to any language. Values can be assigned to variables, which can be changed in the course of program execution. The value assigned to a variable is placed in the memory allocation to that variable. Variables can be created using the keyword char, int, and float.

Identifiers

A C program consists of many elements – variables, functions and so on – all of which have names. The name of a function, variable is called its identifier.

- An identifier consists of letters, digits, and the underscore character.
- An identifier must begin with a letter. (Underscores are allowed in the first character, but leading underscores are reserved for identifiers that the compiler defines.)

- Identifiers are case sensitive. For example: add() and Add() are different.
- An identifier may be any length, but only first 32 characters are significant.
- An identifier may not be one of the reserved C keywords.

Declarations

A declaration associates a group of variables with a specific data type. All variables must be declared before they can appear in executable statements.

A declaration consists of a data type, followed by one more variable name, ending with a semicolon. Each array variable must be followed by a pair of square brackets, containing a positive integer which specifies the size (i.e., no of elements) of the array.

```
int age,houseno;
char choice, answer;
float salary,da,hra;
char name[20];
```

Character Variable

These are used to store single characters enclosed within *single* quotes.

Sample:

```
#include<stdio.h>
main( )
{
    char cvar1='A';
    printf("%c",cvar1);
}
```

Integer Variable

Integer variables are used to store integer numbers.

Sample

```
#include<stdio.h>
main ()
{
    int ivar1=10;
    printf("%d",ivar1);
}
```

Float Variable

Float variables are used to store floating-point (decimal) numbers.

```
#include<stdio.h>
main( )
{
    float fvar1=10.6;
    printf("%f",fvar1); }
```

Defining Strings

In order to store more than one character, strings are defined.

```
char cWord [10];
```

The above declaration tells the compiler to allocate ten continuous bytes to the variable cWord to hold a string of maximum 10 characters.

Example

```
#include<stdio.h>
main()
{
    char svar1[20]="God is Great";
    printf("%s",cvar1);
}
```

Constants

Constants are what some languages call literals and others call immediate values. The constant values are used explicitly in expressions. A constant is distinguished from a variable in two ways.

- It has no apparent compiled place in memory except inside the statement in which it appears.
- The constant can't be addressed; neither the value can be changed.

Character Constants

Character constants specify values that a char variable can contain. A character constant can be coded with an ASCII expression.

```
A1 = 'x';           //ASCII char constant
A2 = '\x2f';        //char constant expressed as hex value
A3 = '\013';        //char constant expresses as octal value
```

Escape Sequence

The '\ ' (backslash) begins an escape sequence. It tells the compiler that something extra is coming. The escape sequences apply to character constants and string constants.

Constant Escape Sequences	
Escape Sequence	Description
\'	Single Quote
*	Double Quote
\\	Backslash
\0	Null (zero) character
\0nnn	Octal number (nnn)
\a	Audible bell character
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\x	Hexadecimal number (nnn)

Data Input and Output

Reading Single Character – The getchar() function

Single characters can be entered into the computer using C library function `getchar()`. `getchar()` is a part of standard C Library language I/O library. It returns a single character from a standard input device. This function does not require any arguments, though a pair of empty parenthesis must follow `getchar`.

Syntax:

Character variable = getchar();

Example:

```
char choice = getchar();
```

Writing Single Character – The putchar() function

Single characters can be displayed on the computer using C library function `putchar()`. `putchar()` is a part of standard C Library language I/O library. It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character

type variable. It must be expressed as an argument to the function, enclosed within parenthesis, following the word putchar.

Syntax:

```
putchar(Character variable);
```

Example:

```
putchar(choice);
```

Reading Input data - The scanf() function

The scanf() function is a standard C library function used to enter data from standard input devices. The input can be any combination of single character, strings or numeric values. The function returns the number of data items that it has written successfully. The syntax of the function is:

```
int scanf(formatting string,arg1,arg2,.....);
```

where,**formatting string** contains the necessary information according to which the scanf() function reads and interprets from the standard input device.

arg1, arg2 etc. are the addresses of the memory locations of the variables into which the function will store the converted values. Actually, these are the pointers (the details of which are discussed in later chapters) to the memory locations, where the converted values are stored.

Basic Conversion Characters for Data Input

Conversion character	Meaning
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating point value
f	Data item is a floating point value
g	Data item is a floating point value
h	Data item is a short integer
i	Data item is a decimal, hexadecimal or octal integer
o	Data item is an octal integer
s	Data item is a string followed by a white space character (the null character \0

u	will automatically be added at the end).
X	Data item is a unsigned decimal integer
[...]	Data item is a hexadecimal integer
	Data item is a string which may include white space characters

The formatting string comprises of single or multiple conversion characters; one character for each input indicating the type of input data. Each character should be preceded by a percent symbol (%).

Example

```
int i;

char x, m[];

scanf("%d %s %c", &i,m,&x);
```

The arguments are written as variables or arrays, whose type must match with the corresponding conversion character in the formatting string. **Each variable name must be preceded with the ampersand (&).** The variable names when preceded by the ‘&’ gives the memory location of the particular location. *However, array names should not be preceded by ‘&’ as the array name itself holds the address of the array* (details in later chapters).

The following example will clarify the use of the use of scanf() function.

```
#include<stdio.h>
main( )
{
    int i,j;
    char x, m[];
    j=scanf("%d %s %c", &i,m,&x);
    printf("%d %s %c %d\n",i,m,x,j);
}
```

In the statement

```
scanf("%d %s %c", &i,m,&x);
```

the first conversion character tells the function that the first input is an integer and it has to be stored in the address of the integer variable ‘i’, similarly the second character signifies that the

second input is a string and the third a character and they are to be stored in the corresponding locations.

scanf() function stops when it exhausts its formatting string or when some input fails to match its format specification. The function returns the number of inputs it has successfully converted and assigned. This value can be trapped and used to check how many inputs the function has found. In the above example this is what is being done by the integer variable 'j'. Please note that scanf() ignores blanks and tabs in its formatting string. It also skips white spaces e.g. blanks, tabs, newline character etc. while looking for input values.

The consecutive non-white space characters that compose a data item collectively define a **field**. It is possible to limit the number of such characters by specifying the maximum **field width** for that data item. To do so, an unsigned integer indicating the field width is placed within the formatting string between the percent sign (%) and the conversion character.

The data item may be composed of fewer characters than the specified field width. However, the number of characters in the actual data item cannot exceed the specified field width. Any characters that extend beyond the specified field will not be read. Such leftover characters may be incorrectly interpreted as the components of the next data item. For example:

```
int i,j;  
scanf("%2d %3d",&i,&j);
```

Here, the field width of 1st input is 2 and the 2nd is 3. Suppose we enter the values 26 and 789. Then, the variables would be assigned as

i=26 and j=789.

However, suppose we enter 267 and 89. Then the assignment will be

i= 26 and j=7.

This because the first 2 values would be given to i as per field width and next input which is '7' will be assigned to j and rest values will be ignored.

Writing Output data - The printf() function

The printf() function is a standard C library function used to write data from computer to the standard output devices. The output can be any combination of single character, strings or numeric values. The function returns the number of data items that it has written successfully. The function is very similar to the scanf() function except that it functions in the reverse manner to it. The syntax of the function is:

```
int printf(formatting string,arg1,arg2,.....);
```

where,

formatting string contains the necessary formatting information.

arg1, arg2 etc. are the individual output data items. These can be constants, single variables, array names, complex expressions or function references. In contrast to the `scanf()` function, the arguments here are not the memory locations, so they should not be preceded by any ampersand (&).

Basic Conversion Characters for Data Output

Conversion character	Meaning
c	Data item is displayed as a single character
d	Data item is displayed as a decimal integer
e	Data item is displayed as a floating point value with an exponent
f	Data item is displayed as a floating point value without an expression
g	Data item is displayed as a floating point value using either e-type or f-type conversion depending on value; trailing zeros, decimal point won't be displayed
i	Data item is displayed as a signed decimal integer
o	Data item is displayed as an octal integer without a leading zero
s	Data item is displayed as a
u	Data item is displayed as a unsigned decimal integer
x	Data item is displayed as a hexadecimal integer without a leading 0x

The formatting string comprises of single or multiple conversion characters; one character for each output indicating the type of output data. Each character should be preceded by a percent symbol (%).

```
#include<stdio.h>
#include<math.h>
main()
{
    float a=40.0,b=3.14159254/4;
    printf("%f %f %f\n",a,a+b,tan(b));
}
```

The example in the demonstrates the use of expression (a+b) and function (tan(b)) as arguments. Notice the white space between the conversion characters in the formatting string. These white

spaces (or any character like \t for tab or \n for new line) will get simply transferred to the output device where they are displayed.

In printf() function, the **minimum** field width of a data value can be fixed (unlike scanf where the maximum field width can be set). This is done by preceding the conversion character with an unsigned integer in the formatting statement. If the output value has fewer characters than the field width then the value is preceded by blank characters to fill the specified field width. However, if the value has more characters then, it is assigned extra space so that the entire data can be displayed. For example:

```
int f;
printf("%2d", f);
```

With printf() function it is possible to limit the maximum field width also. This is called **precision**. By this we can limit the maximum number of decimal places for a floating type value or the maximum number of characters for a string type of value. The precision value is an unsigned integer preceded by a decimal point. If used along with the minimum field width, then it follows the field width specification. Both of them precede the conversion character.

```
#include<stdio.h>
main()
{
    float b=3.14159254;
    printf("value=%14.4f\n",b);
}
```

The above example will give output as:

```
value=          3.1416
```

This is because the precision is 4. So, the decimal value will get rounded off to 4 decimal values. Moreover, since the minimum field width is 14, blank spaces will fill the extra places. That's why the free space preceding the number. Consider another example.

```
#include<stdio.h>
main()
{
    char string[]="microsoft";
    printf("%s %15.5s %.5s\n",string,string,string);
}
```

The program when executed will give the following output:

```
microsoft          micro micro
```

Here notice the 2nd and the 3rd values. The second value is *micro*. This because the precision is 5. So, after the 5th character the string value is truncated. However, the value is preceded by a lot of

empty space. This is because the minimum field width is 15. So, the 1st 10 places are filled with blank space. Similarly in the case of 3rd value, it displays *micro* as the precision is 5 but as there is no minimum field width, the value appears just after one space, which is there in between the conversion characters in the formatting statement.

The gets() and puts() function

C contains a number of other library functions that permit some of data transfer into or out of the computer. Each of these functions accepts a single argument. The argument must be a data item that represents a string. (a character array). The string may include white space characters. In case of gets, the string will be entered from keyboard, and will terminate with a newline character (i.e., the string will end when the user presses ENTER/RETURN key.)

Example:

```
#include<stdio.h>
main()
{
    char name[20];
    gets(name);
    puts(name);
}
```

Expressions

An expression represents a single data item, which can be a number or a character. The expression can be a single entity, such as a constant, a variable, an array or a reference to a function. It may also be the combination of the entities interconnected with the operators. Some examples of expressions are given below.

```
s + a      /* Arithmetic Expression */
x=y        /* Assignment Expression */
m<=n      /* Logical Expression */
# define m 100 /* Constant Expression */
a>b? a : b /* Conditional Expression */
```

Statements

A *statement* is an expression followed by a semicolon ';'. A statement causes the computer to carry out some action. On execution this expression gets evaluated. Statements can be of the following types: -

- Simple or expression statements
- Compound statements or Blocks
- Control Statements

An **expression statement** consists of a simple expression followed by a semi colon. Following are some examples:

```
c=a+b;          /* Assignment Statement*/  
i++;           /* Incrementing Statement*/  
;              /* Null Statement*/
```

The first example is of assignment type statement where the value of the expression (a+b) is assigned to c. Thus, c, which receives the assigned value, is called the *lvalue* as it's on the left side of the operator. And the expression, which provides the value and appears on the right side of the operator, is called the *rvalue*.

The second example is causes the value of 'i' to be incremented 1 i.e. it is same as

```
i=i+1.
```

The last example is of empty expression statement or null statement about which we will discuss in later chapters.

A **compound statement** is a group of individual statements enclosed within the braces {}. These individual statements can again be a expression, compound or control statement. For example,

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

} Each is an individual statement.

Please note that a compound statement doesn't end with a semicolon.

A **control statement** is used to create/add special features in the program like conditional execution, repetitive execution, selective execution etc. About these statements we will discuss in details in the next chapter. A typical control statement is given below.

```
while (a<=10)  
{  
    printf("%d\n",a);  
}
```

OPERATORS

Types of Operators

- Arithmetic Operator
- Assignment Operator
- Relational Operator
- Logical Operator
- Increment & Decrement Operator
- Bitwise Operator
- Unary Operator
- Ternary Operator

Arithmetic Operator

The four arithmetic operators are +, -, *, /. Apart from that C also provides the remainder or *modulus operator* represented by the '%' symbol. The modulo operator returns the remainder when an integer is divided by another.

Sample Program:

```
#include<stdio.h>
main ()
{
    int ivar1=11,ivar2=5,ivar3;
    ivar3=ivar1%ivar2;
    printf("The remainder is %d",ivar3);
}
```

Assignment Operator

The assignment operators are used to form assignment expressions, which assign the value of an expression to an identifier. The most commonly used operator is '=' e.g.

a=b+c;

Apart from that C offers the following arithmetical assignment operators.

+=, -=, *=, /=, %=. The following examples will clarify the usage.

Expression	Meaning
i += 2	i =i+ 2
i -= 2	i =i- 2
i *= (k+2)	i = i * (k+2)
i /= (2 + j)	i = i / (2 + j)
i %= 2	i = i % 2

Relational Operator

Relational operators are used to compare two values. The result of the comparison is either true (1) or false (0). The following table gives the list of relational operators.

Example:

```
/*finding the grade of a student*/
.....
if (marks >= 80)
    grade='A';
else if (marks >=70)
    grade='B';
.....
```

Logical Operator

Logical operators are used to combine two or more test expressions. C provides the AND (&&), the OR (||) and the NOT (!) logical operators.

The && operator combines two condition expressions and evaluates to true only if both conditions are fulfilled.

Example:

```
/* finding the largest of three numbers*/
....
if( a>b && b>c)
    printf("%d is greatest", a);
else if (b>a && b>c)
    printf("%d is greatest", c);
else
    printf("%d is greatest", c);
.....
```

Increment/Decrement Operator

C provides two unusual operators for incrementing and decrementing variables. The increment operator '++' adds 1 to its operand and '--' subtracts 1 from it.

The increment and decrement operators are further of two types:

- **Post increment/decrement:** In post increment the first operation done is, *assignment and then increment*.

Example:

```
int j, x=10;
```

```
j=x++;
```

If we print the value of j and x after the execution of the above statement then value of j will be 10 where as the value of x will be 11.

```
j=x++;
```

is same as

```
store the value of x in register j=x;
```

```
x=x+1;
```

```
j= stored value in register
```

i.e., *first read for assignment and then increment*.

- **Pre increment/decrement:** In pre-increment the first operation done is, *increment and then assignment*.

Example:

```
int j, x=10;
```

```
j=++x;
```

If we print the value of j and x after the execution of the above statement then value of j will be 11 and the value of x will be 11.

```
j=++x;           is same as
```

```
x=x+1;
```

```
j=x;
```

i.e., *first increment and then assignment*.

Bitwise Operator

C provides six operators for bit manipulation; these may only be applied to integral operands, i.e, char, short, int, and long, whether signed or unsigned.

&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

Example:

int x=10;

the binary equivalent is 1010.

If we right shift x then the binary representation is 0101 i.e, the decimal value is 5.

				1	0	1	0
--	--	--	--	---	---	---	---

Decimal value: 10

				0	1	0	1
--	--	--	--	---	---	---	---

Right shift x i.e. >>x

Decimal value: 5

If we left shift x then the binary representation is 10100 i.e, the decimal value is 20.

			1	0	1	0	0
--	--	--	---	---	---	---	---

Left shift x i.e. <<x

Decimal Value: 20

Unary operator

Unary operators are operators that take only one operand.

Example: ++, --, *, &, (type), sizeof, +, -, ~, !

sizeof(char) returns 1

sizeof(int) returns 2 if it's a 16-bit compiler

 returns 4 if it's a 32-bit compiler

sizeof(float) returns 4 if it's a 16-bit compiler

 returns 8 if it's a 32-bit compiler

Ternary Operator

The conditional expression, can be written with the ternary operator '?':.

Syntax: *expr1 ? expr2 : expr3*

The expression `expr1` is evaluated first. If it is non-zero(true), then the expression `expr2` is evaluated, and is the value of conditional expression. Other wise `expr3` is evaluated, and that is the value.

Example:

```
int max, a, b;
max = (a>b) ? a : b;
```

This is same as

```
if (a>b)
    max=a;
else
    max=b;
```

Precedence and Order of Evaluation

The table provides the rules for precedence and associativity of all operators. The operators in the same row have the same precedence and the rows have been arranged in the order of decreasing precedence. Thus the operators `'*', '/', '%'` all have the same precedence, however, higher than `'+', '-', '++', '--'`. The unary operators `+`, `-` and `*` have higher precedence than their corresponding binary operators. The operator `()` refers to a function call, operators `'->'` and `'.'` are used to access the members of structures and `sizeof` operator is used to get the size of an object in bytes. All these operators are discussed in greater details in the later chapters.

Operators	Associativity
<code>() [] -> .</code>	Left to right
<code>! - ++ -- + - * & (type) sizeof</code>	Right to left
<code>* / %</code>	Left to right
<code>+ -</code>	Left to right
<code><<>></code>	Left to right
<code><<= >>=</code>	Left to right
<code>== !=</code>	Left to right
<code>&</code>	Left to right
<code>^</code>	Left to right

	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= &= ^= = <<= >>=	Right to left
,	Left to right

C doesn't specify the order in which the operators are evaluated. (The exceptions are &&, ||, ?: and ,) For example,

```
a= add() + subtract();
```

Here, If the both functions add() and subtract() modify the same variable, then the value of 'a' will depend on their order of evaluation i.e. compiler dependent. Hence, the effective way would be to evaluate the individual functions in a sequence and then calculate 'a'.

```
x= add();
```

```
y= subtract();
```

```
a=x+y;
```

C also fails to specify the order in which the function arguments are to be evaluated. The example below will show you the discrepancy.

```
printf("%d %d\n", ++x, power(2,x));
```

Here, the results would be different in different compilers depending upon whether x is incremented before or after the function is called. The effective code would be

```
++x;
```

```
printf("%d %d\n", x, power(2,x));
```

Function calls, assignment statement, increment/decrement operators often lead to "*side effects*" i.e. situations where some variable gets changed as a result of the evaluation of a certain expression. In such situations there is sometimes a dependency on the order of evaluation for the variables in the expression to be updated. Thus a cautionary measure, it is better to adopt such programming practise where you avoid writing codes that depend on the order of evaluation.

MCQ

- 1) Which is not a character of C?
 - a) \$
 - b) ^
 - c) ~
 - d) |
- 2) An identifier can't start with
 - a) _
 - b) uppercase alphabet
 - c) lowercase alphabet
 - d) #
- 3) Which is not a keyword in C?
 - a) const
 - b) main
 - c) sizeof
 - d) void
- 4) Identify the scalar datatype in C.
 - a) Double
 - b) Union
 - c) Function
 - d) Array
- 5) Identify the derived data type in C
 - a) Int
 - b) Float
 - c) Union
 - d) Array
- 6) Which datatype can either be used to represent a scalar data type or a derived data type?
 - a) Pointer
 - b) double
 - c) structure
 - d) union
- 7) The qualifier that may precede float is
 - a) Signed
 - b) Unsigned
 - c) Long
 - d) None of the above
- 8) The qualifier that may precede char is
 - a) Signed
 - b) Unsigned
 - c) Options a & b
 - d) None of the above
- 9) The qualifier that may precede double is
 - a) Signed
 - b) Unsigned
 - c) Short
 - d) Long
- 10) Printable characters always use
 - a) Negative integers
 - b) Positive integers
 - c) Both positive and negative integers
 - d) -1
- 11) The integral data type includes
 - a) Enum
 - b) Int
 - c) Char
 - d) All of the above
- 12) Which is not a valid integer constant?
 - a) 600000 u
 - b) 534878 ul
 - c) 0Xabpq
 - d) 0X625
- 13) Which is not a valid floating constant
 - a) 4E -6 f
 - b) 4E 12
 - c) 0.08 e -4
 - d) 1.3345 F
- 14) Identify the octal constant
 - a) 627
 - b) 0X25
 - c) -0756
 - d) 06.52
- 15) An octal constant is preceded by
 - a) X
 - b) 0X
 - c) O
 - d) 0
- 16) A hexa constant is preceded by
 - a) 0X
 - b) O
 - c) HX
 - d) 0
- 17) A character constant is written within
 - a) ""
 - b> ''
 - c) options a & b
 - d) none of the above
- 18) String constants are represented within
 - a> ''
 - b> ""
 - c) option a & b
 - d) /* ... */
- 19) Identify the invalid string constant.
 - a) "a+b"
 - b> ""
 - c> " ' "
 - d> ' A '
- 20) Which is not a character constant
 - a) '\60'
 - b) '\012'

- c) '\x24'
 - d) 'SUM'
- 21) Identify the escape sequences
- a) '\0'
 - b) '\n'
 - c) '\f'
 - d) all of the above
- 22) Which is not a white space character
- a) \f
 - b) \v
 - c) \0
 - d) blank
- 23) Identify the white space character
- a) Blank
 - b) \f
 - c) \r
 - d) all of the above
- 24) Identify the invalid string constant
- a) "5\t 10\t 15\t"
 - b) "cost = 90\x24\n"
 - c) "\n Don't care condition"
 - d) 'C is flexible'
- 25) If a constant is given a name it becomes
- a) A string constant
 - b) Manifest
 - c) Const declaration
 - d) Invalid
- 26) Symbolic constants are defined as
- a) #define S1 S2
 - b) #define S1 S2;
 - c) #define S1=S2
 - d) #define S1=S2;
- 27) The symbol # in the #define statement must commence from
- a) Anywhere in a line
 - b) The first column of a line
 - c) The first column of next line
 - d) The first line
- 28) What is an object in C
- a) Constant
 - b) Variable
 - c) Identifier
 - d) Keyword
- 29) Identify the C token
- a) Keywords
 - b) Constants
 - c) Operators
 - d) All of the above
- 30) Statement terminator is represented by
- a) :
 - b) Blank
 - c) ;
 - d) \n

Practice

1. Write a C program to multiply and divide a given number by 2 without using * (multiplication) and / (divide) operators? (Use bit wise operators)
2. Interchange the value of two variables without using a temporary variable.
3. Write a program to find the area of a rectangle by accepting the length and breadth of it.
4. Write a C program to read n and print the sum of the square of the first n natural numbers using the formula $sum = n(n+1)(2n+1)/6$.
5. Write a C program to find the compound interest if principal, rate of interest and time period is given.
6. Write a C program to convert a given no. of days to years, weeks or days.
7. Accept the semester marks of a student for four subjects. Find out the average and total marks scored by the student.
8. For a constant PI value, find out the volume of a cylinder by accepting the radius and height.
9. Accept the time of arrival and departure for employees in a company. Find out the following details: 1. minutes/hours late incoming 2. minutes/hours early outgoing 3. total time spent in office if the office incoming time is 9:30 a.m. and outgoing time is 5:30 p.m.
10. Accept the Basic salary of an employee and display the following details.
 - a. DA - 40%
 - b. HRA - 12.5%
 - c. TAX- 15%
 - d. GROSS Income
 - e. NET Income

Chapter – III: Control Statements

In the last chapter, we discussed how to write programs in C. I am sure by now you would be writing lots of programs. All the programs that we have discussed so far constitute of a set of statements, which get executed one after one in the same order as they have been written and each of them are executed only once. However, in real life when we see around our self we find that most of the time we do something i.e. we execute an action based upon some conditions or situations which means a possible situation tells us the course of action. Let's say going to the college - now this action will depend upon the condition whether the college is open or not. If it's open, we go to the college else not. In real life programming also we require the same type of implementation.

Many programs are there where at certain point some condition is evaluated i.e. a logical statement is executed. The execution of a statement will depend upon the outcome the condition. This type of execution is called **conditional execution**. Another form of conditional execution is that where among the various groups of statements a particular group is selected subject to the fulfilment of some condition. This is called **selection**.

Again, there are many programs where a particular group of statement gets executed repeatedly till a particular is satisfied. This is called **looping**. You can compare the situation with eating. At home you are given a '*thali*' and you go on eating till the '*thali*' is empty. Again we come across situation where it is not known in advance how many times the repetition will take place. This type of situation continues till a particular condition becomes true; like going to the hotel and continue eating till the stomach is full. A third situation comes when beforehand it is known for how many times the codes would be repeatedly executed. This is something like carrying 4 '*samosas*' in Tiffin. So, beforehand you know that you have to execute the action of eating a '*samosa*' 4 and only 4 times.

All these above situations discussed above get implemented in C through control statements. This chapter will be discussing about its various types of implementations.

Control Statements

In the last chapter, I have discussed about the various types of statements of which one is control statement. Control statements are statements, which are used to create special program features such as conditional execution, loops and branches etc. These statements most of the time, require the other form of statements i.e. expression and compound statements to be embedded in side them. At times even one control statement is embedded inside another.

Conditional Execution of Statements

if-else Statement

The if-else statement is used to express decisions. It will carry out a logical test and depending upon its outcome, true or false, it will execute one of the two actions i.e. statements.

Syntax:

if (expression)

statement1;

else

statement2;

The second part i.e. 'else part' is optional in if statement. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (expression is zero) then statement 1 is not executed and if there is an else part, then statement2 is executed instead. Note that the expression must be enclosed within a parenthesis as shown above. The statement 1&2 can be simple or compound. The example below will show the usage.

```
/*Find the maximum value of two numbers*/  
  
#include<stdio.h>  
main()  
{  
    int a,b;  
    printf("Enter two numbers\n");  
    scanf("%d %d",&a,&b);  
    if(a>b)  
    {  
        printf("First number is greater\n");  
    }  
    else  
    {  
        printf("Second number is greater\n");  
    }  
    return 0;  
}
```

Since an 'if condition' simply tests the numeric value of an expression, certain coding shortcuts are possible.

if (expression)

Can be written instead of

if (expression != 0)

Multiple if statement (if – else if statement)

Syntax

```
if (expression)
    statement;
else if (expression)
    statement;
else if (expression)
    statement;
else
    statement;
```

This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. The statements can be either single or compound. The last else part handles the “None of the above”.

/*Depending upon the marks secured by a student finds his gradation. */

```
#include<stdio.h>
main()
{
    int marks;
    printf("Enter the total number of marks you have scored");
    scanf("%d",&marks);
    if(marks>=80)
    {
        printf("\nGrade : A+");
    }
    else if(marks>=70)
    {
        printf("\nGrade : A");
    }
    else if(marks>=60)
    {
        printf("\nGrade : B");
    }
    else
    {
        printf("\nGrade : F");
    }
    return 0;
}
```

Nested if - else statement

Syntax :

```
if (expression1)
{
    if(expression2)
    {
        if(expression3)
            statements1;
        else    statements2;
    }
    else
    {
        if(expression4)
            statements3;
        else    statements4;
    }
}
else
{
    if (expression5)
    {
        if(expression6)
            statements5;
        else    statements6;
    }
    else
    {
        if (expression7)
            statements7;
        else    statements8;
    }
}
```

This sequence of if statements is the most general way of writing a dependent nested decision. The expressions are evaluated in order. If *expression1*, *expression2*, *expression3* are true i.e. they return non-zero value then *statements1* is executed and if *expression3* returns false then *statements2* is executed. If *expression2* returns false then *expression4* is evaluated and depending upon its return value, true or false, *statements3* or *statements4* is executed. If *expression1* returns false then *expression5* gets evaluated and depending upon its return value *expression6* or *expression7* gets evaluated. If *expression6* gets evaluated then its value decides the execution of *statements5* or *statements6* and if *expression7* gets evaluated then its value decides the execution of *statements7* or *statements8*.

Thus in the whole of the nested if-else statement the first expression that returns a non-zero value, its corresponding statements get executed and the rest of the nest will be bypassed. Therefore, the moment a condition is found true, the control is moved out of the entire nest ignoring all other statements of the nest. Here, note that these statements can be simple or compound statements.

Example:

```
/*Accept three numbers and find out the largest amongst them*/
#include<stdio.h>
main()
{
    int x,y,z;
    printf("Enter three numbers");
    scanf("%d %d %d",&x,&y,&z);
    if(x>y)
    {
        if(x>z)
        {
            printf("First number is greatest");
        }
        else
        {
            printf("Third number is greatest");
        }
    }
    else
    {
        if(y>z)
        {
            printf("Second number is greatest");
        }
        else
        {
            printf("Third number is greatest");
        }
        return 0;
    }
}
```

The switch - case Statement

The switch statement is an example of *selection* type of control statements. This statement causes a particular group of statements to be chosen and executed among the various groups available. The choice of the group will depend upon the current value of the expression that gets evaluated at the beginning of the switch statement.

Syntax:

switch(expression)

```
{
    case const-expr: statements

    case const-expr: statements

    default: statements }
```

The switch statement is a multi way decision that tests whether an expression matches one of the numbers of constant integer values, and branches accordingly. The expression can return character constant also. The default keyword takes care of any value that is not matching with any of the cases. Note however that the presence of default group is optional. If the value of the expression doesn't match with any of the cases then the control goes directly to the statement that follows the switch statement. However, if we provide a default group then it gives us a convenient way of generating error message or error correction routine. This default group can appear anywhere in the switch statement and not necessarily at the end.

Example:

```
/*Depending upon the day number find out the name of the week day*/
#include<stdio.h>
main()
{
    int wkdyo;
    printf("Enter day number");
    scanf("%d ", &wkdyo);
    switch(wkdyo)
    {
        case 0: printf("Sunday");
                break;
        case 1: printf("Monday");
                break;
        case 2: printf("Tuesday");
                break;
        case 3: printf("Wednesday");
                break;
        case 4: printf("Thursday");
                break;
        case 5: printf("Friday");
                break;
        case 6: printf("Saturday");
                break;
        default: printf("Invalid day number");
    }
    return 0;
}
```

In the above example note the presence of *break* statement at the end of each case group. The break statement causes the control to go out of the switch statement immediately after executing the chosen case and thus prevent the execution of any further cases following it. About the break statement, more in the later sections.

Iterative Control Statements – loops

The while Statement

The while statement is used to carry out looping operations i.e. repeated execution of statements.

Syntax:

```
while(expression)  
    statements;
```

The *expression* is evaluated. If it is true i.e. it returns a non-zero value then the *statements* get executed and the expression is again evaluated. This process continues repeatedly till the expression becomes false. The expression is usually a logical statement, which return either true (a non-zero value) or false (0 value). The statements can be simple or compound though mostly they are compound statements. The statements must include a mechanism by which the value of the expression gets continuously changed thus providing a stopping condition for the loop.

Example:

```
/*Print 1 ... 100*/  
#include<stdio.h>  
main()  
{  
    int n=0;  
    while (n<=100)  
    {  
        n++;  
        printf("%d\t",n);  
    }  
    return 0;  
}
```

The for Statement

The for statement is probably the most commonly used control statement for looping.

Syntax:

```
for (expression1 ; expression2 ; expression3 )  
    statement;
```

As is shown it comprises of three expressions. *expression1* is used to initialise the parameter, which controls the loop; *expression2* is the condition which must be satisfied i.e. it must return true for the iteration to continue; and *expression3* which provides the mechanism for the value of expression2 to change and thus stop the iteration.

Example:

```

/*Calculate the average of n numbers*/
#include<stdio.h>
main()
{
    int i,n,num;
    float avg,sum;
    printf("\nHow many number's average to be calculated? ");
    scanf("%d",&n);
    printf("\nPlease enter the numbers: - ");
    for(i=1,sum=0.0;i<=n;++i)
    {
        printf("\nNumber %d :\t",i);
        scanf("%d",&num);
        sum+=num;
    }
    avg=sum/n;
    printf("\nThe average of the numbers is : %.2f",avg);
    return 0;
}

```

Please note that any of the three parts can be omitted, though the semicolons must remain.

```

for(    ;    ;    )
{
....
}

```

However, it is important that you understand the importance of the omissions of these expressions. The first and third expressions can be omitted provided in the program some other means have been provided for initialising and altering the controlling parameter. However, if the 2nd parameter is omitted then it implies that the controlling condition is always true and so the loop will continue indefinitely unless, it is terminated by some other means like break or return statements.

The do - while Statement

The while or for loops test the termination condition at the top. The do...while, tests at the bottom after making each pass through the loop body; thus the body is always executed at least once.

Syntax:

```
do  
{  
    statement;  
    .....  
}  
while(expression);
```

Example:

```
/*Print the first 10 whole numbers*/  
#include<stdio.h>  
main()  
{  
    int i=0;  
    do  
    {  
        printf("\n%d",i);  
        i++;  
    }while(i<10);  
    return 0;  
}
```

The break and continue Statement

The usage of break statement has been previously demonstrated in conjunction with switch statement. It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The *break* statement provides an early exit from for, while or do, just as from switch statement.

Syntax

break;

A *break* statement causes the control to move out of the immediate enclosing loop or switch statement and execute the next statement that follows it.

The *continue* statement when executed causes the control to **skip** the remaining statement following it in the loop on that particular pass. However, it doesn't terminate the loop and there lies its difference with break statement which terminates the loop and takes the control out of it whereas a continue statement merely causes the control to skip the remaining statements of the loop in that particular pass and start the pass once again.

Syntax

continue;

Example

```
/* Print the first 10 multiples of 17 */
#include<stdio.h>
main()
{
    int i=0,count=0;
    do
    {
        i++;
        if(i%17!=0)
            continue;
        printf("\n%d",i);
        count++;
    }while(count<10);
    return 0;
}
```

The goto Statement

The goto statement is used to ignore the normal sequence of the program and transfer the control to some other part of the program.

Syntax

goto level;

.....
.....

level:

statement;

level is an identifier used to mark the statement where the control would be transferred. *level* should always end with a ':' followed by statements. One program can have many labels. But, no two labels should have the same name.

MCQ

- 1) Identify the wrong statement
 - a) if (a<b);
 - b) if a<b;
 - c) if (a<b) { ; }
 - d) Options b and c
- 2) Which is syntactically correct?
 - a) if (a := 10) { ... } else if (a<10) { ... }
 - b) if(a== 10) { ... }elseif (a<10){ ... }
 - c) if(a eq 10) { ... }elseif (a<10) { ... }
 - d) if(a .eq. 10) { ... }elseif (a<10){ ... }
- 3) Which is the correct statement?
 - a) printf(" Maximum = % d\n", (x,y)? x:y);
 - b) printf("%s\n". (mark >= 60) ? "FIRST CLASS" : "NOT");
 - c) printf("%s \n". "PASS");
 - d) all the above
- 4) The minimum number of times the while loop is executed is
 - a) 0
 - b) 1
 - c) 2
 - d) cannot be predicted
- 5) The minimum number of times the for loop is executed is
 - a) 0
 - b) 1
 - c) 2
 - d) cannot be predicted
- 6) The minimum number of times the do-while loop executed is
 - a) 0
 - b) 1
 - c) 2
 - d) cannot be predicted
- 7) The do-while loop is terminated when the conditional expression returns
 - a) zero
 - b) 1
 - c) non-zero
 - d) -1
- 8) Which conditional expression always returns false value?
 - a) if (a == 0)
 - b) if (a = 0)
 - c) if (a = 10)
 - d) if(10 == a)
- 9) Which conditional expression always returns true value?
 - a) if(a = 1)
 - b) if (a == 1)
 - c) if (a = 0)
 - d) if(1 == a)
- 10) What is (are) the statements) which results in infinite loop?
 - a) for (i - 0; ; i+ +);
 - b) for (i = 0; ;);
 - c) for (; ;);
 - d) all the above
- 11) In the following loop construct, which one is executed only once always
for (expri; expr2; expr3) { ... }
 - a) expr1
 - b) expr3
 - c) expr1 and expr3
 - d) expr1, expr2 and expr3
- 12) Identify the wrong construct.
 - a) for (expr1; expr2;);
 - b) for(expr1; expr3)
 - c) for (; expr ;)
 - d) for (; ; expr3)
- 13) Infinite loop is
 - a) useful for time delay
 - b) useless
 - c) used to terminate execution
 - d) not possible
- 14) What is the value of x in the expression
x = (a = 10, a*a) ?
 - a) invalid expression
 - b) 0
 - c) 10
 - d) 100
- 15) Comma is used as
 - a) a separator in C
 - b) an operator in C
 - c) a delimiter in C
 - d) terminator in C
- 16) What is the result of a multiple expression separated by commas?
 - a) result of the leftmost expression after evaluation
 - b) result of the rightmost expression after evaluating all other previous expressions i
 - c) no value is returned
 - d) result of arithmetic operations, if any
- 17) Which is the correct statement?
 - a) while loop can be nested
 - b) for loop can be nested
 - c) options a and b
 - d) one type of a loop cannot be nested in another type of loop
- 18) The break statement is used to
 - a) continue the next iteration of a loop construct
 - b) exit the block where it exists and continues further sequentially
 - c) exit the outermost block even if it occurs inside the innermost block and continues further sequentially
 - d) terminate the program

- 19) The continue statement is used to
 - a) continue the next iteration of a loop construct
 - b) exit the block where it exists and continues further
 - c) exit the outermost block even if it occurs inside the innermost
 - d) continue the compilation even an error occurs in a program
- 20) The continue statement is used in
 - a) selective control structures only
 - b) loop control structures only
 - c) options a and b
 - d) go to control structure only
- 21) The break statement is used in
 - a) selective control structures only
 - b) loop control structures only
 - c) options a and b
 - d) switch-case control structures only
- 22) If break statement is omitted in each case statement
 - a) The program executes the statements following the case statement where a match is found and exits the switch-case construct.
 - b) The program executes the statements following the case statement where a match is found and also all the subsequent case statements and default statements
 - c) The program executes default statements only and continues with the remaining code.
 - d) Syntax error is produced.
- 23) If default statement is omitted and there is no match with case labels
 - a) No statement within switch-case will be executed.
 - b) Syntax error is produced.
 - c) Executes all the statements in the switch-case construct.
 - d) Executes the last case statement only.
- 24) When is default statement executed in switch-case construct?
 - a) Whenever there is exactly one match.
 - b) Whenever break statement is omitted in all case statements.
 - c) Whenever there is no match with case labels.
 - d) options b and c.
- 25) Identify the wrong statement where --- represents C code
 - a) label :{---}--- goto label;
 - b) goto end; - - - end: (- - - }
 - c) goto 50; - - - 50: {- - -}
 - d) begin : { - - - } if (a < 10) goto begin;

Practice

1. Write a program to check out a number is Armstrong or not?
2. Write a program to print the following format

1

22

333

4444

.....

3. Write a program to construct a pyramid of digits:

1

2 3 2

3 4 5 4 3

4 5 6 7 6 5 4

4. Create a class called 'student' that contains a name and a student roll no. Include a member function to get data from the user and function to show data.
5. Write a program to compute the exponential value of a given number x using the series.
 $e(x)=1+x+x^2/2!+x^3/3!+.....$
6. Write a program to generate reverse pyramid of digits?
7. Write a program to print multiplication table of any given number?
8. Write a program to compute the area of right triangle and triangle using switch statement. Case 1 find area of right triangle and case 2 find area of triangle?
9. Write a program to find the Fibonacci series up to 50 and find sum and average?
10. Write a program to find the Factorial of a number given by user using recursive method?
11. Write a program to check the input number is palindrome or not?
12. Write a program to convert a decimal number to hexadecimal number?
13. Write a program to accept 0,1 or 2. If 0 is entered by user, accept the necessary parameter to calculate the volume of a cylinder. Inputs 1 and 2 correspond to cylinder and cone , respectively. This process go until the user enters q to terminate the program.(use switch case statement)
14. Write a program to accept a string , sum all of its ASCII values and print it.
15. Write a program to input a line , count the number of space, vowels, tab characters are present.

Chapter – IV: Functions and Storage classes

Basics of Functions

Functions are the building blocks of C programs. A function groups a number of program statements into a single unit whose purpose is to carry out some specific well-defined task. Every C program must have at least one function, which is called the `main()`. The program execution begins by carrying out the instruction in it. All other additional functions will be subordinate to the `main()` and may be to one another. Each of these subordinate functions is a self-contained unit and has to be defined independently. The definition of no function can be embedded inside another. A function can be invoked i.e. called from other parts of the program as many times as is the requirement and from multiple locations. Once a function is called the control goes to it and the codes within it are executed and after it gets completed the control returns back to the same point from where the called function was invoked. Generally a function processes the data that is passed to it. These data are passed to the function through special identifiers called *arguments* or *parameters*.

Need of Functions

When programs become large, a single list of instructions becomes difficult to understand. For this reason, functions are adopted. A function has a clearly defined purpose and a clearly defined interface to other functions in the program. A group of functions together form a larger entity called a module.

Another reason for using functions is to reduce the program size. Any sequence of instructions that is repeated in a program can be grouped together to form a function. The function code is stored in only one place in the memory. The use of function also enables a programmer to develop a customised library of functions containing frequently used instructions. Alternatively these functions can also contain system dependent information. Thus whenever a particular instruction has to be carried out, the programmer will just include the particular function and attach it during compilation. This way the repetitive programming among many programs of the same instruction gets avoided. Moreover, once the system dependent instructions get defined in the functions, the programmer no more has to bother about the environment while writing programs. This system independent feature makes the program portable across the platforms

Function Definition

Syntax

<return type> function name (<input parameters>);

Each function is represented by the function name, the input parameters it accepts and data type of the value that it returns.

The following program illustrates the use of a function:

```

/*This program uses a function to calculate the sum of two numbers*/
#include<stdio.h>
void add ();          /*function prototyping*/
main ()
{
    printf("Calling function add()");
    add();             /*A call to the function add()*/
    printf("A return from add( ) function");
    return 0;
}
/*Function definition for add*/
void add()
{
    int inum1, inum2, inum3;
    printf("Input two numbers");
    scanf("%d",&inum1);
    scanf("%d",&inum2);
    inum3=inum1+inum2;
    printf("The sum of the two numbers is %d", num3;
}

```

The function add() is invoked from main(). The control passes to the function add(), and the body of the function gets executed. The control then returns to main() and the remaining code gets executed

Sample output:

Calling function add()

Input two numbers

10

20

The sum of two numbers is 30 A return from add() function. The function main (), which invokes the function add(), is called the **calling function**, and the function add() is called function

Function Prototyping

Just as any variable is declared in a program before it is used, it is necessary to declare or prototype a function to inform the compiler that the function would be referenced at a later point in the program.

```
void add( );
```

is a function declaration or a prototype.

Defining a Function

The function definition contains the code for the function. The function definition is:

```
void add()
{
    int inum1, inum2, inum3;
    printf("Input two numbers");
    scanf("%d, %d",&inum1, &inum2);
    inum3=inum1+inum2;
    printf("The sum of the two numbers is %d",inum3);
}
```

A function declaration can be avoided if the function definition appears before the first call to the function.

Function and Arguments

Argument(s) of a function is (are) the data that the function receives when called from another function.

It is not always necessary for a function to have arguments. The function add() in last Program did not contain any arguments.

```
#include<stdio.h>
void add(int ivar1,int ivar2)
/* function containing two arguments*/
{
    int ivar3=ivar1+ivar2;
    printf("The sum of the two numbers is %d",var3);
}
main()
{
    int inum1,inum2;
    printf("Input two numbers");
    scanf("%d",&inum1);
    scanf("%d",&inum2);
    add(inum1,inum2); /*passing two values to the function add( )*/
}
```

Calling Function

In C++ programs, functions that have arguments can be invoked by:

- Value
- Reference

Call by value

The following program illustrates the invoking of a function by value:

```
/*This function swaps the value of two variables*/
#include<stdio.h>
void swap(int,int);
void main()
{
    int ivar1,ivar2;
    printf("Input two numbers");
    scanf("%d, %d",&ivar1,&ivar2);
    swap(ivar1,ivar2);
    printf("%d,%d",ivar1,ivar2);
}
void swap(int inum1,int inum2)
{
    int itemp;
    itemp=inum1;
    inum1=inum2;
    inum2=itemp;
    printf("%d,%d",inum1,inum2);
}
```

Sample output of Program

Input two numbers

```
15
25
    25  15
    15  25
```

values entered for the variable ivar1 are passed to the function swap(). When the function swap() is invoked, these values get copied into the memory locations of the parameters inum1 and inum2, respectively,

ivar1	ivar2	ivar1	ivar2
15	25	15	25
15	25	25	15
inum1	inum2	inum1	inum2

When the swap function invoked

After the function executed

Thus the variables in the caller function [main()] are distinct from the variable in the called function [swap()] as they occupy distinct memory locations.

Note:

The values of the variable in the earlier do not get affected when the arguments are passed as value.

Passing Reference Argument

In passing reference arguments, a reference to the variable in the calling program is passed.

When arguments are passed by value, the called function creates a new variable of the same type as the arguments, and copies the argument values into it. Passing arguments by value is useful when the function does not need to modify the values of the original variables in the calling program.

```
/*This program swaps the values in the variable using function
containing reference arguments*/
#include<stdio.h>
void swap(int &,int &);
main()
{
    int ivar1, ivar2;
    printf("Enter two numbers");
    scanf("%d,%d",&ivar1,&ivar2);
    swap(ivar1,ivar2);
    printf("%d,%d",ivar1,ivar2);
}
void swap(int &inum1,int &inum2)
{
    int itemp;
    itemp=inum1;
    inum1=inum2;
    inum2=itemp;
    printf("%d,%d",inum1,inum2);
}
```

Reference arguments are indicated by an ampersand (&) preceding the argument:

```
int &inum1;
```

The ampersand (&) indicates that inum1 is an alias for ivar1 which is passed as an argument. The function declaration must have an ampersand following the data type of the argument.

```
void swap(int &,int &)
```

The ampersand sign is not used during the function call:

```
swap(ivar1,ivar2)
```

Sample output of Program

Enter two numbers

12 24

24 12

24 12

In Program, the values in the variables ivar1 and ivar2 in the calling program are swapped.

ivar1	ivar2	ivar1	ivar2
15	25	25	15
inum1	inum2	inum1	inum2
<u>When the swap function invoked</u>		<u>After the function executed</u>	

Storage Classes

To fully define a variable, one need to mention not only the **data type** but also its **storage class** i.e, not only all variables have a data type but also a storage class. But till yet we have not mentioned about the storage type anywhere, thus we are not giving complete definition for a variable. We were able to do this because storage classes have defaults i.e., if we do not specify the storage class of the variable in its declaration the compiler will assume a storage class depending on the context in which the variable is used.

From C compiler's point of view, a variable name identifies some physical location within the computer where string of bits representing the variable's value is stored. They can be basically stored in two locations: Memory and CPU registers. It is the storage class of the variable, which determines in which of the two locations the value is stored.

A variable's storage class tell us the following details about a variable:

- Where the variable would be stored?
- What will be the initial value of the variable, if the initial value is not specifically assigned (i.e., the default initial value)?
- What is the scope of the variable i.e. in which functions the value of the variable is available?
- What is the life of the variable i.e., how long would the variable exist?

The storage types supported by C are:

- Auto
- Static
- Extern
- Register

Automatic Storage Type

Data pertaining to a function is lost when the function has been executed completely. Variables defined in a function are in memory and retain their value only as long as the function is in execution. In C, such data has been classified as data of storage type auto. So far, all functions have been written using auto type data.

Thus we can summarize the automatic storage class in the following way:

Storage	:	RAM (SS).
Default initial value	:	An unpredictable value, which is often called a garbage value.
Scope	:	Local to the block in which the variable is defined.
Life	:	Till the control remains within the block in which the variable is defined.

Example:

```
int ivar; char cvar; invoice ivar1;
```

are, by default, treated as:

```
auto int ivar1; auto char cvar1; auto invoice ivar1;
```

The following program shows how an automatic storage class variable is declared, and the fact if the variable is not initialised it contains a garbage value.

```
main()
{
    auto int x,y;
    printf("\n%d %d",x,y);
}
```

The output will be : 0 202 which are indeed garbage values.

Please note that the key word used is auto not automatic.

Scope and lifetime of an automatic type of variable is illustrated in the following program.

```
main()
{
    auto int x=1;
    {
        {
            printf("\n%d",x);
        }
        printf("%d",x);
    }
    printf("%d",x);
}
```

The output of the above program is 1 1 1.

This is because all printf() statements occur within the outmost block in which x has been defined. This means the scope of x is local to the block in which it is defined.

The moment the control comes out of the block within which the variable is declared, the variable and its value is irretrievably lost. The following example illustrates this fact. But this program can best run on a C++ compilers as individual block scopes are exhibited in C++.

```
int main()
{
    auto int x=1;
    {
        auto int x=2;
        {
            auto int x=3;
            cout<<"\n Innermost block - \tAddress : "<<&x
                <<"    Value: "<< x;
        }
        cout<<"\n Middle block - \tAddress : "<<&x
            <<"    Value: "<< x;
    }
    cout<<"\n Outer block - \tAddress : "<<&x
        <<"    Value: "<< x;
}
return 0;
```

The output of the program :

Innermost block -	Address :	65520	Value: 3
Middle block -	Address :	65522	Value: 2
Outer block -	Address :	65522	Value: 2

Here, the compiler treats the three x's as totally different three variable as since they are defined in different blocks. You can see the address of the 3x's in the output. Once the control comes out of the innermost block, the value 3, which was in x is lost, and x is recreated for middle block and holds the value 2. Similarly, as the control comes out of the middle block, the value 2, which was in x is lost, and x is recreated for outer block and holds the value 1.

Register Storage Type

A register declaration advises the compiler that the variable will be heavily used for example loop counter. The idea that register variables are to be placed in machine registers, which may result in smaller and faster programs as a value stored in a CPU register can always be accessed faster than the one, which is stored in memory.

Thus we can summarize the register storage class in the following way:

Storage	:	CPU Register/RAM (SS)
Default initial value	:	Garbage value.
Scope	:	Local to the block in which the variable is defined.
Life	:	Till the control remains within the block in which the variable is defined.

Example:

```
main()
{
    register int x;
    for(x=1;x<=50;x++)
        printf("%d", x);
}
```

Here though we have declared x a register type, we can't say for sure that the value of x will be stored in CPU register, as compilers are free to ignore this advice. This is because; the number of CPU registers is limited, and they may be busy doing some other task. So, the case in which compiler ignores the request to make a variable register type, there it treats the variable as auto type. As microcomputers generally use 16-bit registers, they cannot hold a float, double or long value as they require 4/8 bytes of storage space.

Therefore the following declarations are wrong.

```
register float fvar;
register double dvar;
```

```
register long ivar;
```

If we declare like this, then compiler will not give any error message rather it will treat it as auto type of variable.

Static Storage Type

As opposed to auto type data, C also offers static type data, which, as its name suggests, retains its value even after the function to which it belongs has been executed.

We can summarize the static storage class in the following way:

Storage	:	RAM(DS)
Default initial value	:	Zero.
Scope	:	Local to the block in which the variable is defined.
Life	:	Value of the variable persists between different function calls.

The following example illustrates the difference between auto and static type data.

Example:

```
#include<stdio.h>
void dummy(void);
main( )
{
    int ivar;
    for(ivar=0;ivar<3;ivar++)
    {
        dummy();
    }
    return 0;
}
void dummy(void)
{
    int ictr=1;
    printf("Function executed %d times",ictr);
    ictr++;
}/* program 1*/
```

```

/*Program to differentiate between auto and static type data*/
#include<stdio.h>
void dummy();
main()
{
    int ivar;
    for(ivar=0;ivar<3;ivar++)
    {
        dummy( );
    }
    return 0;
}
void dummy(void)
{
    static int ictr=1;
    printf("Function executed %d times", ictr);
    ictr++;
}
/* program 2 */

```

In the programs shown, the function dummy() is used to print the number of times the function has been called. Execution stops when the value of the variable ictr in main reaches three. The output of Program – 1 would be:

Function executed 1 time

Function executed 1 time

Function executed 1 time

Whereas that of Program – 2 would be

Function executed 1 time(s)

Function executed 2 time(s)

Function executed 3 time(s)

This is because, in Program – 1 the variable ictr is declared as auto type data. Therefore, every time the function dummy() is called, the variable gets created and initialized to 1. But the static declaration in Program – 2 ensures that the variable is declared and initialised only once, that is, when the function is invoked the very first time.

It is always advised to avoid using static type of variables unless it really needed. Because, once the variable is declared static, their values are kept in memory when the variable is not active, which means they take up space in memory that could otherwise be used by other variables.

Extern Storage Type

Apart from auto and static types, a variable can also be declared in a manner such that it is available to all functions in a program file, that is, it is a global variable. A variable declared outside all function is called a global variable.

We can summarize the extern storage class in the following way:

Storage	:	RAM(DS)
Default initial value	:	Zero.
Scope	:	Global.
Life	:	As long as the program's execution does not come to an end.

Example 1

An example of this type of data is given below

```
#include<stdio.h>
int x;
main()
{
    printf("\nx = %d",x);
    increment();
    increment();
    decrement();
}
increment()
{
    x++;
    printf("\n Incrementing by 1 the value of x is %d", x);}
decrement()
{
    x--;
    printf("\n Decrementing by 1 the value of x is %d", x);}
```

The output will be:

x=0

Incrementing by 1 the value of x is 1

Incrementing by 1 the value of x is 2

Decrementing by 1 the value of x is 1

So, it is obvious from the output that value of x is available to all the functions. As per the rule, if any variable is declared as extern then only you can access it as demonstrated in the following example. Then, how in the above-mentioned example it works globally!!! This is possible because use of extern inside a function is optional as long as we declare it outside and above that function in the same source code file.

Example – 2

```
#include<stdio.h>
int ival;
main()
{
    printf("Enter value:");
    scanf("%d",&ival);
    disp();
} /*program - 3*/
void disp()
{
    extern int ival; //extern is used to refer to ival declared in
last Program
    printf("Value entered is %d",ival);
}/*program - 4*/
```

In the above example, Program 3 declares a global variable called ival and also accepts a value from the user. Then, it calls a function disp() which is in another program file, namely Program 4. Program 4 refers to the variable ival using the extern declaration since the ival is declared in Program 3 and not in program 4. Sample output after the two programs are compiled together and linked would be as follows:

Enter value: 10

Value entered is 10

The following table summarizes the three storage types.

Use extern storage class for only those variables which are being used by almost all functions, this will avoid unnecessary passing of these variables as arguments when making a function call. Declaring all variables as extern would amount a lot of wastage of memory space because these variables remain active throughout the life of the program.

A Comparison

Storage Type	Created	Initialized	Can be accessed
Auto	Each time the function is invoked	At the time of declaration or later	Only in the function in which it is declared
Static	The first time the function is invoked	At the time of declaration so that the value is not reset; static arrays can also be initialized.	Only in the function in which it is declared.
Extern	Created at the time when declared as global, but not where declared extern.	At the time of declaration or later.	In any function in same/different program file.

Recursion

C functions may be used recursively; that is, a function may call itself directly or indirectly. When a function calls itself recursively, each invocation gets a fresh set of the entire automatic variables, independent of the previous set.

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n * (n-1) * (n-2) * \dots * 1, & \text{if } n>0 \end{cases}$$

The factorial function, whose domain is the natural number, can be defined as

Lets consider the case of 4! Since $n>0$, we use the second clause of the definition.

$$4! = 4 * 3 * 2 * 1 = 24$$

Also

$$3! = 3 * 2 * 1 = 6$$

$$2! = 2 * 1 = 2$$

$$1! = 1$$

$$0! = 1$$

Such definition is called iterative because it calls for the explicit repetition of some process until a certain condition is met.

Lets look at the definition of $n!$

$$4! = 4 * 3 * 2 * 1 = 4 * 3!$$

Also

$$3! = 3 * 2 * 1 = 3 * 2!$$

$$2! = 2 * 1 = 2 * 1!$$

$$1! = 1 * 0!$$

$$n! = \begin{cases} 1; & \text{if } n=0 \\ n * (n-1)!; & \text{if } n>0 \end{cases}$$

Or

Such a definition, which defines an object in terms of a simpler case of itself, is called a recursive definition. Here, $n!$ is defined in terms of $(n-1)!$, which in turn is defined in terms of $(n-2)!$ And so on, until finally $0!$ is reached.

The basic idea, here is to define a function for all its argument values in a constructive manner by using induction. The value of a function for a particular argument value can be computed in a finite number of steps using the recursion definition, where at each step of recursion, we come nearer to the solution.

```
factorial(int n)
{
    if (n==0)
        return (1);
    return (n*factorial(n-1));
}
```

Note that in the second return statement **factorial** calls itself, which is an essential ingredient of a recursive routine.

A procedure that contains a procedure call to itself or a procedure call to a second procedure, which eventually causes the first procedure to be called, is known as a recursive procedure.

There are two important conditions that must be satisfied by any recursive procedure.

- a) A smallest, base case that is processed without recursion and acts as a decision criterion for stopping the process or computation.
- b) A general method that makes a particular case to reach nearer in some sense to the base case.

In the case of factorial function each time the function calls itself, its argument is decremented by 'one', so that the argument is getting smaller, to ultimately reach the value zero which is the base case.

Writing Recursive algorithms:

Recursion is a tool to allow the programmer to concentrate on the key step of algorithm without worrying initially about coupling that step with all others. Some of the important guidelines are listed below.

- a) Solve the base case(s), which is usually the small, special case that is trivial or easy to handle without recursion.
- b) Solve the general case(s) correctly the terms of a smaller case of the same problem. Once you have a simple, small step toward the solution, check whether the remainder of the problem can be solved in the same manner.
- c) Combine the stopping rule and the key step, using an "if" statement to select between them.
- d) Verify that the recursion will always terminate. Check the starting with a general case, the base case will be satisfied in a finite number of steps and the recursion terminates.
- e) The algorithm should be able to return gracefully, if it is called on to do nothing - which may be a stopping rule in the recursive algorithm.

Some More Examples of Recursive Algorithm:

Example – 1: Multiplication of natural numbers

The product $a * b$, where a and b are positive numbers, may be written as a added to itself b times. This definition is iterative. We can write this definition in a recursive way as:

$$a * b = \begin{cases} a & \text{if } b=1 \\ a * (b - 1) + a & \text{if } b>1 \end{cases}$$

To evaluate $5 * 3$, we first evaluate $5 * 2$ and add 5. To evaluate $5 * 2$, we first evaluate $5 * 1$ and then add 5. $5*1$ is equal to 5 due to first part of the definition.

Therefore,

$$5 * 3 = 5 * 2 + 5 = 5 * 1 + 5 + 5 = 5 + 5 + 5 = 15$$

We can represent this definition of multiplication of natural numbers in C as following:

```
multiplication (int x, int y)
{
    if ( y == 1 )
        return x;
    return (multiplication (a, y-1) + a);
}
```

Example – 2: The Fibonacci Sequence

The fibonacci sequence is the sequence of integers

0, 1, 1, 2, 3, 5, 8, 13, 21 ...

Each element here is the sum of the two preceding elements.

0,(0+1),(1+1),(1+2),(2+3)...

We may define this sequence by letting fib(0) = 0 and fib (1) = 1, as follows

$$\text{fib}(n) = \begin{cases} n & \text{if } n = 0, 1 \\ \text{fib}(n-2)+\text{fib}(n-1) & \text{if } n \geq 2 \end{cases}$$

This definition can be modelled as a C program to compute n^{th} fibonacci number as follows:

```
fibonacci(int n)
{
    int a,b;
    if (n <= 1)
        return (n);
    a = fib (n - 1);
    b = fib (n - 2);
    return (a+b);
}
```

Example – 3: Finding The Greatest Common Divisor

The greatest common divisor of two integers is defined as follows by the Euclid's algorithm:

$$\text{GCD}(m,n) = \begin{cases} \text{GCD}(n,m) & \text{if } (n>m) \\ m & \text{if } n = 0 \end{cases}$$

Here mod (m,n) is the remainder on dividing m by n. The base case is when the second argument is zero. In this case the greatest common divisor is the first argument.

If second argument is greater than the first, the order of argument is interchanged. Finally, the GCD is defined in terms of itself. Please note that the size of the argument is getting smaller as mod (m,n) will eventually reduce to zero in a finite number of steps.

A C routine can be written as follows to implement the above principle:

```
GCD(int m,int n)
{
    int x;

    if (n>m)

        return (GCD(n,m) );

    if (n==0)

        return (m) ;

    x=m mod n;

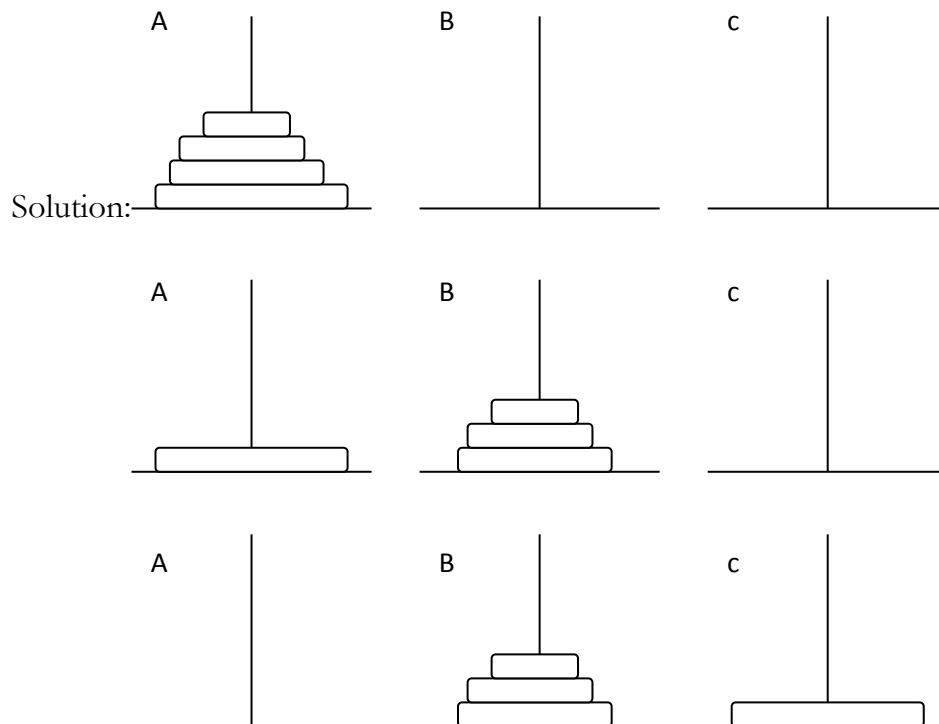
    return (GCD (n,x) );

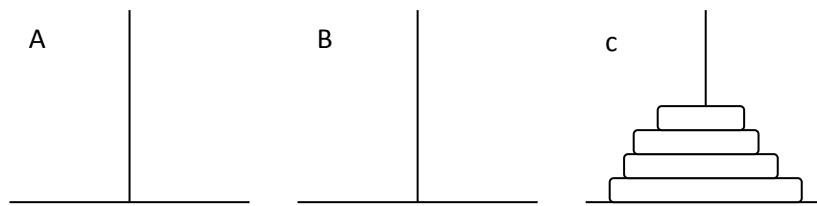
}
```

Example – 4: The Tower of Hanoi Problem

There are n disks of different sizes and there are three needles A, B and C. All the n disks are placed on a needle (A) in such a way so that a larger disk is always below a smaller disk. The other two needles are initially empty. The aim is to move the n disks to the second needle (C) using third needle (B) as a temporary storage. The rules of movement are as follows:

- Only the top disk may be moved at a time.
- A disk may be moved from any needle to any other needle.
- A larger disk may never be placed upon a smaller disk.





Recursive Solution for 4 disks for Tower of Hanoi Problem

```
#include <stdio.h>

void Hanoi (int n, char initial, char final, char temp)
{
    if (n==1)
        printf("Move disk - 1 from %c to %c \n", initial, final);
    return;
    Hanoi (n - 1, initial, temp, final);
    printf("Move disk - %d from %c to %c \n", n, initial, final);
    Hanoi (n - 1, temp, final, initial);
}

main()
{
    int n;
    printf("Enter no. of disks to be moved");
    scanf("%d", &n);
    Hanoi(n, 'A', 'C', 'B');
}
```

MCQ

- 1) The program execution starts from
 - a) The functions which is first declared
 - b) main() function
 - c) the function which is last defined
 - d) the function other than main()
- 2) How many main() functions can be defined in a C program?
 - a) 1
 - b) 2
 - c) 3
 - d) any number of times
- 3) A function is identified by an open parenthesis following
 - a) a keyword
 - b) an identifier other than keywords
 - c) an identifier including keywords
 - d) an operator
- 4) A function with no action
 - a) is an invalid function
 - b) produces syntax error
 - c) is allowed and is known as dummy function
 - d) returns zero
- 5) Parameters are used
 - a) to return values from the called function
 - b) to send values from the calling function
 - c) options a and b
 - d) to specify the data type of the return value
- 6) Identify the correct statement.
 - a) A function can be defined more than once in a program.
 - b) Function definition cannot appear in any order.
 - c) Functions cannot be distributed in many files.
 - d) One function cannot be defined within another function definition.
- 7) The default return data type in function definition is
 - a) Void
 - b) Int
 - c) float
 - d) char
- 8) The parameters in a function call are
 - a) actual parameters
 - b) formal parameters
 - c) dummy parameters
 - d) optional
- 9) The parameters in a function definition are
 - a) actual parameters
 - b) formal parameters
 - c) dummy parameters
 - d) optional
- 10) The parameters passing mechanism used in C is
 - a) call by reference
 - b) call by name
 - c) call by value
 - d) options a and b
- 11) The storage class that can precede return data type in function declaration is
 - a) extern
 - b) static
 - c) options a and b
 - d) register
- 12) Recursive call results when
 - a) A function calls itself
 - b) A function calls another function, which in turn calls the function
 - c) options a and b
 - d) a function calls another function
- 13) The main() function calls in a C program
 - a) allows recursive call
 - b) does not allow recursive call
 - c) is optional
 - d) is a built-in function
- 14) The function main() is
 - a) a built-in function
 - b) a user defined function
 - c) optional
 - d) all the above
- 15) The storage class allowed for parameters is
 - a) auto
 - b) static
 - c) extern
 - d) register
- 16) Functions are assigned by default
 - a) auto storage class
 - b) static storage class
 - c) extern storage class
 - d) register storage class
- 17) Functions have
 - a) file scope
 - b) local scope.
 - c) block scope
 - d) function scope
- 18) The function defined in math.h file for returning the arc tangent of x is
 - a) tan_l(x)
 - b) atan(x)
 - c) tanh(x)
 - d) arctan(x)
- 19) The function ceil(x) defined in math.h
 - a) returns the value rounded down to the next lower integer
 - b) returns the value rounded up to the next higher integer

- c) The next higher value
 - d) the next lower value
- 20) The function floor(x) in math.h
- a) returns the value rounded down to the next lower integer
 - b) returns the value rounded up to the next higher integer
 - c) the next higher value
 - d) the next lower value
- 21) The function strcpy(s1, s2) in string.h
- a) copies s1 to s2
 - b) copies s2 to s1
 - c) appends s1 to end of s2
 - d) appends s2 to end of s1
- 22) The function strcat(s1, s2) in string.h
- a) copies s1 to s2
 - b) copies s2 to s1
 - c) appends s1 to end of s2
 - d) appends s2 to end of s1
- 23) The function strcmp(s1, s2) returns zero
- a) If s1 is lexicographically less than s2
 - b) If s1 is lexicographically greater than s2
 - c) if both s1 and s2 are equal
 - d) if s1 is empty string
- 24) The function toupper(ch) in ctype.h
- a) returns the upper case alphabet of ch
 - b) returns the lower case alphabet of ch
 - c) returns upper case if ch is lower case and lower case if ch is in upper case
 - d) is a user-defined function
- 25) The function tolower(ch) in ctype.h
- a. returns the upper case alphabet of ch
 - b. returns the lower case alphabet of ch
 - c. returns upper case if ch is lower case and lower case if ch is in upper case
 - d. is a user-defined function**
- 26) What function must all C programs have
- a) start()
 - b) main()
 - c) return()
 - d) exit()
- 27) void show();
main()
{
 show();
}
void show(char *s)
{
 printf("%s\n", s);
}

What will happen when the above code is compiled and executed using a strict ANSI C compiler?

- a) It will compile and nothing will be printed when it is executed.
 - b) It will compile, but not link.
 - c) The compiler will generate an error.
 - d) The compiler will generate a warning.
- 28) int i=3, a=1, b=1;
void func()
{
 int a=0;
 static int b = 0;
 a++; b++;
}
- int main()
{
 for (i=0; i < 5; i++)
 {
 func();
 a++; b++;
 }
 printf ("a=%d b=%d\n", a, b);
}

What will be printed from the sample code above?

- a) a=0 b=6
 - b) a=5 b=5
 - c) a=5 b=6
 - d) a=6 b=6
- 29) long factorial (long x)
{
 ????
 return x * factorial(x - 1);
}

What would replace the ??? with, to make the function shown above, return the correct answer?

- a) if (x == 0) return 0;
 - b) if (x == 0) return 1;
 - c) if(x <= 1) return 1;
 - d) return 1;
- 30) if (x ? y : z) doSomething();
Referring to the sample above, which statement correctly identifies when y is evaluated?
- a) y is evaluated only when x == 1.
 - b) y is evaluated only when x >= 1.
 - c) y is evaluated only when x != 0.
 - d) y is evaluated only when x == 0.

Practice

1. Accept a 5-digit integer, write a function to calculate sum of digits of the 5 digit number:
 - a. Using recursion.
 - b. Without using recursion
2. Write a function to create the first 20 fibonacci series.
3. Write a function that will accept a number and produce the following series:
$$1+x+x^2+x^3+\dots+x^n$$
4. Write functions those will accept the start-value, end-value and generate even, odd and prime numbers respectively within the range.
5. Write a function that will accept a number and find the prime factors of the number.

Find out the errors in the following programs.

```
6. main()
{
    int i = 135, a=135, k;
    k=pass(i,a);
    printf("\n%d",k);
}
pass(int j, int b)
int c;
{
    c = j + b;
    return ( c );
}
7. main()
{
    int p = 23, f= 24;
    myfriend(&p, &f);
    printf("\n%d %d",p,f);
}
mtfriend(int q, int g)
{
    q=q+q;
    g=g+g;
}
```

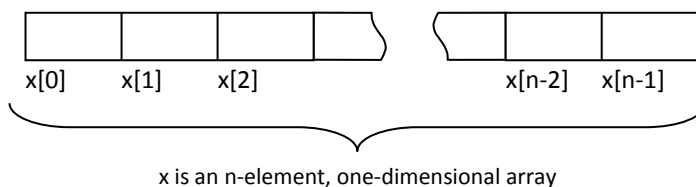
```
8. main()
{
    int k = 35, z;
    z=check(k);
    printf("\n%d", z);
}
check(m)
{
    int m;
    if(m>40)
        return ( 1 );
    else
        return ( 0 );
}
9. main()
{
    int i = 35, *z;
    z=function(&i);
    printf("\n%d", z);
}
function (int *m)
{
    return (m+2);
}
```

Chapter –V: Arrays

Introduction

Many applications require the processing of multiple data items that have common characteristics (e.g., a set of numerical data, represented by $x_1, x_2 \dots x_n$). In such situations it is often convenient to place the data items into an array, where they will share the same name i.e., x in the above example. The individual data items can be characters, integers, floating-point numbers etc. They must all, however, be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets. Each subscript must be expressed as a nonnegative integer. Thus, in the n -element array x , the array elements are $x[0], x[1], x[2], \dots x[n-1]$ as shown in the following figure. The value of each subscript can be expressed as an integer constant, an integer variable or a more complex expression.



The number of subscripts determines the dimensionality of the array. For example: $x[i]$ refers to an element in the one-dimensional array x . Similarly $y[i][j]$ is the j^{th} element of the i^{th} row. Higher dimensional arrays can also be formed, by adding additional subscripts in the same manner like $x[i][j][k]$.

Defining an Array

Arrays are defined in the same manner as any ordinary variable is declared. The difference is each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression in square brackets. The expression is usually written as a positive integer constant.

Syntax:

```
Storage-class data-type array[expression];
```

Where storage-class refers to the storage class of an array, data-type the type of data and array-name is the array-name, and expression is a positive-valued integer expression that indicates that number of array-elements. The storage-class is optional; default values are automatic for arrays defined within a function or a block, and external for arrays defined outside of a function.

Example:

```
int marks[10];
char emp_name[15];
float heights[5];
static char address[50];
```

The first shows that marks is a 10-element integer array, the second is a text of 15 characters, the third one – height is a 5 elements float array and address is a static 50-element character array.

When we declare an array with a definite size i.e., constant value, then at the time of any manipulation the maximum size is mentioned. Now for any change in the maximum array size, all the occurrences of size have to be searched and changed. That is why; it is sometimes convenient to declare the array size in symbolic constants rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all occurrences to the maximum array size (in for loops as well as the definition) can be altered simply by changing the value of the symbolic constant.

Example:

```
/* Conversion of a test from Upper Case to Lower and Vice Versa */
```

```
#include<stdio.h>
#define max 20
main()
{
    char str[max];
    printf("Enter a string");
    scanf("%s",str);
    for(int i=0;i<max;i++)
    {
        if(str[i]>=65 && str[i]<=90)
            str[i]+=32;
        else if(str[i]>=97 && str[i]<=122)
            str[i] -= 32;
    }
    printf("%s",str);
    return 0;
}
```

Now in the above program, if any time the maximum size of the array has to be changed, then only change that is needed to be done is the value defined in the symbolic constant i.e., only the #define value will be changed.

Initializing an Array

Unlike the automatic type of variables the automatic type of array cannot be initialised. However, the external and static array definitions can include the assignment of initial values if desired. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas.

Syntax:

storage-class data-type array[expression]={value 1, value 2, ..., value n};

where value 1 refers to the value of the first array element, value 2 refers to the value of second array element and so on.

For example:

```
char name[10]={ 'C', 'I', 'T', 'Z', 'E', 'N' '\0' };
```

This declaration is same as name[0]='C';name[1]='I';name[2]='T';name[3]='Z';name[4]='E';name[5]='N';

```
int mark[3]={86,79,82};
```

This declaration is same as marks[0]=86; marks[1]=79; marks[2]=82;

```
float salary={5000.95,8900.35,7500.55};
```

This declaration is same as salary[0]=5000.95; salary[1]=8900.35; salary[2]=7500.55;

All individual array elements that are not assigned explicit initial values will automatically be set to zero. This includes the remaining elements of an array in which certain elements have been assigned nonzero values. **Please note that the automatic assignment of 0 only takes place if we initialise a part of an array, otherwise if we declare an array and try to display its value, they may show junk values.**

Example:

Let's consider the output of the following 2 programs.

```
/*Program No- 1*/
/*Displaying the values without initialising the array*/
#include<stdio.h>

main()
{
    int arr[5];
    for(int i=0;i<5;i++)
    {
        printf("%d ",arr[i]);
    }
    return 0;
}
```

Output: 12 98 3 44 3 ----- garbage/junk values

```

/*Program No- 2*/
/*Displaying the values after initialising a part of the array*/
#include<stdio.h>
main()
{
    int arr[5]={23,10,90};
    for(int i=0;i<5;i++)
    {
        printf("%d ",arr[i]);
    }
    return 0;
}

```

Output: 23 10 90 0 0 ----- displays zero in case of un-initialised values

Similarly in case of character array it displays space where un-initialised values appear.

The array size need not be specified explicitly when initial values are included as a part of an array definition. With a numerical array, the array size will automatically be set equal to the number of initial values included within the definition.

Example:

```
int arr[ ]={23,10,90};
```

Processing an Array

Arrays do not behave like natural data type in case of single operations like arithmetic operation, assignment operation, comparison operation etc. i.e., C does not permit single operations involving entire array. These operations must be carried out element-by-element basis, which is done inside a loop, where each pass to the loop processes one array element. Thus the number of passes in the loop will therefore equal to the number of array elements to be processed.

Example:

```

/*calculcate the average of n numbers, then compute the deviation of
each about the average*/
#include<stdio.h>
main()
{
    int n,count;
    float avg,d,sum=0;
    float list[100];

    /*Read in a value for n*/
    printf("\nHow many numbers will be averaged ?");
}

```

```

scanf("%d",&n);
/*Read in the numbers and calculate their sum*/
for(count=0;count<n;count++)
{
    printf("i = %d x = ",count+1);
    scanf("%f",&list[count]);
    sum+=list[count];
}
/*Calculate and write out the average*/
avg=sum/n;
printf("\nThe average is %5.2f\n",avg);
/*calculate and write out the deviation about the average*/
for(count=0;count<n;count++)
{
    d=list[count]-avg;
    printf("\ni = %d x = %5.2f d = %5.2f",count+1,list[count],d);
}
return 0;
}

```

If we declare the array above the main(), then the array becomes an external array. Here we can remove the explicit size declaration of an array as the no. of initialised values will determine its size.

Example:

```

/*calculcate the average of n numbers, then compute the deviation of
each about the average*/
#include<stdio.h>
int n=5;
float list[]={3,-2,12,4.5,6.7}
main()
{
    int count;
    float avg,d,sum=0;
    float list[100];
    /*Calculate and write out the average*/
    for(count=0;count<n;count++)
    {
        sum+=list[count];
    }
    avg=sum/n;
    printf("\nThe average is %5.2f\n",avg);
}

```

```

        /*calculate and write out the deviation about the average*/
        for(count=0;count<n;count++)
        {
            d=list[count]-avg;
            printf("\ni = %d x = %5.2f d = %5.2f",count+1,list[count],d);
        }
        return 0;
    }
}

```

Passing Arrays to a function

An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function. The manner in which the array is passed to the function differs notably, from that of an ordinary variable.

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

Example:

Lets discuss the above-mentioned example by passing an array as argument

```

/*This program outline illustrates the passing of an array from the main
portion of the program to a function*/

#include<stdio.h>
float average(int, float[]);
main()
{
    int n,count;
    float avg, list[100];
    /*Read in a value for n*/
    printf("\nHow many numbers will be averaged ?");
    scanf("%d",&n);
    /*Read in the numbers and calculate their sum*/
    for(count=0;count<n;count++)
    {
        printf("i = %d x = ",count+1);
        scanf("%f",&list[count]);
    }
    avg= average(n,list);
    printf("\n Average of the array is %5.2f\n",avg);
    return 0;
}

```

```

float average(int a, float x[])
{
    int count;
    float avg,sum;
    for(count=0;count<a; count++)
    {
        sum+=x[count];
    }
    avg=sum/a;
    return avg;
}

```

Some care is to be taken when writing function declaration with argument type specification. If any of the arguments is an array, an empty pair of brackets should follow the data type of each array argument, which indicates that the argument is an array. In the function definition similarly an empty pair of braces should follow the name of each array argument.

e.g.,

float average(int, float[]); ----- prototyping / function declaration

float average(int a, float x[]) ----- function definition

Similarly, if the first line of a function definition includes the formal argument declarations, an empty pair of square brackets must follow each array name appearing as a formal argument.

e.g.;

float average (a, x) ----- function definition

int a;

float x [];

From whatever we have discussed we can conclude the fact that arguments are passed to a function by values when the arguments are ordinary variables. When an array is passed to a function, however the values of the array elements are not passed to the function. Rather, the array name is interpreted as the address of the first array element (i.e., address of the memory location containing the first element of an array.). The address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first array element. Arguments passed in this manner are said to be passed by reference rather than by value.

When a reference is made to an array element within the function, the value of the element's subscript is added to the value of the pointer to indicate the address of the specified array element. Therefore, an array element can be accessed from within the function. If an array element is altered within a function, the alteration will be recognized in the calling portion of the program i.e throughout the entire scope of the array definition.

Example: (Bubble Sort)

```
/*Sorting an integer array in ascending order */
#include<stdio.h>
#define size 100
void sort(int,int []);
main()
{
    int i, n, x[size];
    /*Read in a value for n*/
    printf("\n How many numbers will be entered ?");
    scanf("%d", &n);
    /*Read in the list of numbers */
    for(i=0;i<n;i++)
    {
        printf("\n i = %d  x = ",i+1);
        scanf("%d", &x[i]);
    }
    /*reorder all array element*/
    sort(n,x);
    /*display the sorted list of numbers */
    printf("\n\n Recorded list of numbers: \n\n");
    for(i=0;i<n;i++)
    {
        printf("i = %d      x = %d\n", i + 1,x[i]);
    }
    return 0;
}
```

```

void sort(int n,int x[])
{
    int i, item, temp;
    for(item = 0; item < n - 1;item++)
    {
        /*find the smallest of all remaining elements*/
        for(i = item + 1; i < n; i++)
        {
            if(x[i] < x[item])
            {
                /*interchange two elements*/
                temp=x[item];
                x[item]=x[i];
                x[i]= temp;
            }
        }
    }
    return;
}

```

Multidimensional Arrays

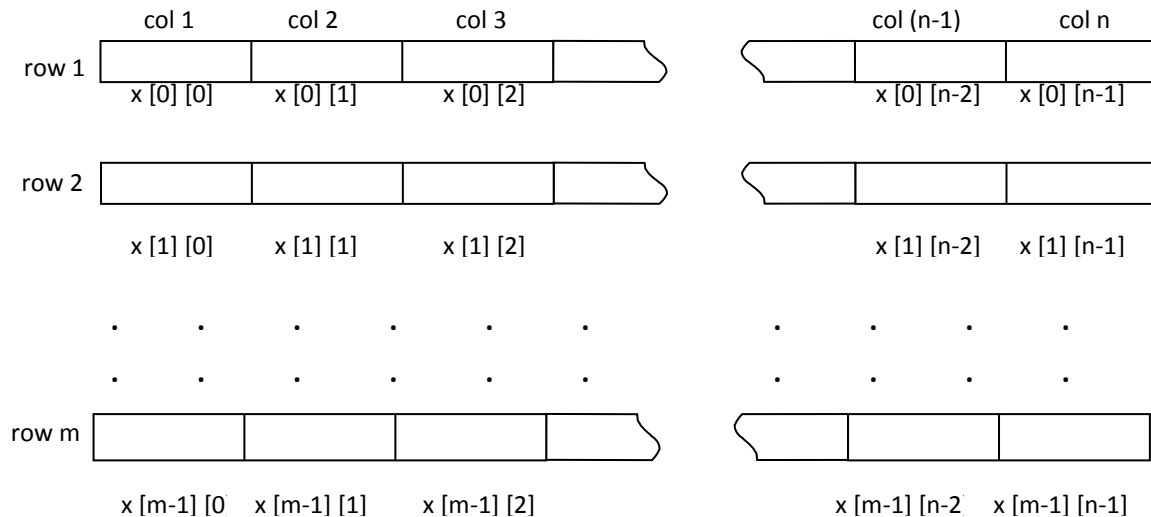
Multidimensional arrays generally represent tables. Multidimensional arrays are represented quite similarly the manner single-dimensional arrays are represented except that a separate pair of square brackets is required for representing each subscript. This way, a two dimensional array will require two pairs of square brackets , a three dimensional array will require three pairs of square brackets and so on.

Definition:

Storage-class data-type array [expression 1][expression 2][expression 3]...[expression n];

Where a storage class refers to the storage class of the array, data type is the data type, array is the array name, and expression 1, expression 2, . . . expression n are positive-valued integer expressions that indicate the number of array elements associated with each subscript. A storage class is optional; the default values are automatic for arrays that are defined inside of a function, and external for arrays defined outside of a function.

As a n-element, one-dimensional array can be thought of as a list of values, similarly, an m x n, two-dimensional array can be thought of as a table of values having m rows and n columns. A three-dimensional array can be visualised as a set of tables and so on.



Example:

```
float marks[4][5];
```

This represents a marks table that contains 4 rows and 5 columns i.e., (4x5=20 elements).

```
char page[24][80];
```

This represents a page as a character array having 24 rows and 80 columns (80 characters in each row) i.e., (24 x 80 = 1920 characters).

```
static double records[100][66][255];
```

This represents a set of 100 static, double-precision tables, each having 66 lines and 255 elements in a line (100 x 66 x 255=1,683,000 elements).

```
static double records[x][y][z];
```

This definition is same as the previous one except instead of defining a static array size, the size is defined with symbolic constants.

If a multi-dimensional array definition includes the assignment of initial values, then care must be given to the order in which the initial values are assigned to the array elements. Only external and static arrays can be initialised. The rule of assignment is that the last/rightmost subscript increases most rapidly. Thus, the elements of a two-dimensional array will be assigned by rows, that is, the elements of the first row will be assigned, then the elements of second row, and so on.

Example:

```
int values[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

Is same as if we initialise as

```
int values[3][4]={
```

```
    {1,2,3,4},
    {5,6,7,8},
    {9,10,11,12}
};
```

1	2	3	4
5	6	7	8
9	10	11	12

If we initialise as

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

Is same as if we initialise as

```
int values[3][4]={
```

```
    {1,2,3},
    {4,5,6},
    {7,8,9}
};
```

then the rest elements will have the 0 values as

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

Manipulation of a two-dimensional array

Example:

1	2	3		4	5	6		5	7	9
5	6	7	+	7	8	9	=	12	14	16
9	11	12		11	12	14		20	23	26

```
/*Adding values of two tables*/
```

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

```
#define rmax 20
```

```
#define cmax 30
```

```
/*function prototypes*/
```

```
void readinput(int a[][cmax], int nrows, int ncols);
```

```
void computesum(int a[][cmax], int b[][cmax], int c[][cmax],int  
nrows, int ncols);
```

```
void writeoutput(int c[][cmax], int nrows, int ncols);
```

```

main()
{
    int nrows, ncols;
    /*array definition */
    int a[rmax][cmax], b[rmax][cmax], c[rmax][cmax];

    printf("\nHow many rows ?");
    scanf("%d",&nrows);
    printf("\nHow many columns ?");
    scanf("%d",&ncols);
    printf("\n\nFirst Table: \n");
    readinput(a,nrows,ncols);
    printf("\n\nSecond Table: \n");
    readinput(b,nrows,ncols);
    computesum(a,b,c,nrows,ncols);
    printf("\n\nSums of the elements: \n\n");
    writeoutput(c,nrows,ncols);
    return 0;
}

void readinput(int a[][cmax], int m, int n)
/*Read in a table of integers*/
{
    int row,col;
    for(row=0;row<m;row++)
    {
        printf("\n Enter data for row no. %2d\n", row + 1);
        for(col=0;col<n;col++)
            scanf("%d",&a[row][col]);
    }
    return;
}

void computesum(int a[][cmax], int b[][cmax], int c[][cmax], int m, int
n)
/*Add the element of two integer tables*/
{
    int row, col;
    for (row=0; row<m;row++)
        for(col=0;col<n;col++)
            c[row][col]= a[row][col]+ b[row][col];
    return;
}

void writeoutput(int a[][cmax], int m, int n)
/*write out a table of integers*/

```

```

{
    int row, col;
    for (row=0; row<m;row++)
        for(col=0;col<n;col++)
            printf("%4d", a[row][col]);
return;
}

```

Arrays and Strings

A string can be represented as a one-dimensional character array. Each character within the string will be stored within one element of the array. For smoother manipulation of a string, C compiler includes special, string oriented library functions for comparing, coping, concatenating strings.

int strcmp(char *, char *)

This function compares two character strings and returns the difference between the characters. It distinguishes between upper and lower case letters.

- It returns a negative value if the first string alphabetically precedes the second string.
- A value zero is returned if first string and second string are identical.
- A positive value is returned if the second string alphabetically precedes the first string.

Example:

```

char str[]={"Anukampa"};

char str1[]={"anukampa"};

int x=strcmp(str,str1);

```

x holds a positive value as 'a' alphabetically precedes 'A'.

int strcasecmp(char *, char *)

This function behaves same the way, as strcmp() except it does not distinguish between upper and lowercase letters.

Example:

```

char str[]={"Anukampa"};

char str1[]={"anukampa"};

int x=strcasecmp(str,str1);

```

x will hold 0, as strcasecmp() does not distinguish between uppercase 'A' and lowercase 'a'.

void strcpy(char *,char *)

This function copies the values held in the second string to the first string.

Example:

```
char str1[10];  
  
char str2[]={"GOD"};  
  
strcpy(str1,str2);
```

After the execution of the above function, str1 will contain GOD as content of str2 will be copied to str1.

void strcat(char *,char *)

This function concatenates / adds the values held in the second string to the first string.

Example:

```
char str1[]={"GOD"};  
  
char str2[]={"IS GOOD"};  
  
strcat(str1,str2);
```

After the execution of the above function, str1 will contain "GOD IS GOOD" and str2 will contain "IS GOOD" as content of str2 will be concatenated / added to str1.

MCQ

- 1) Array is used to represent
 - a) a list of data items of integer data type
 - b) a list of data items of real data type
 - c) a list of data items of different data type
 - d) a list of data items of same data type
- 2) Array name is
 - a) an array variable
 - b) a keyword
 - c) a common name shared by all elements
 - d) not used in a program
- 3) One-dimensional array is known as
 - a) Vector
 - b) Table
 - c) Matrix
 - d) an array of arrays
- 4) The array elements are represented by
 - a) index values
 - b) subscripted variables
 - c) array name
 - d) size of an array
- 5) Array elements occupy
 - a) subsequent memory locations
 - b) random location for each element
 - c) varying length of memory locations for each element
 - d) no space in memory
- 6) The address of the starting element of an array is
 - a) represented by subscripted variable of the starting element
 - b) can not be specified
 - c) represented by the array name
 - d) not used by the compiler
- 7) Identify the wrong statement.
 - a) Subscripts are also known as indices.
 - b) Array variables and subscripted variables are same.
 - c) Array name and subscripted variables are different.
 - d) Array name and subscripted variables are same.
- 8) Array subscripts in C always start at
 - a) -1
 - b) 0
 - c) 1
 - d) any value
- 9) Identify the correct declaration.
 - a) `int a[10][10];`
 - b) `int a[10,10];`
 - c) `int a(10)(10);`
 - d) `int a(10.10);`
- 10) Maximum number of elements in the array declaration `int x[10];` is
 - a) 9
 - b) 10
 - c) 11
 - d) undefined
- 11) The elements of the following array `x` are `float x[5];`
 - a) `x[0], x[1], x[2], x[3], x[4]`
 - b) `x[1], x[2], x[3], x[4], x[5]`
 - c) `x(0), x(1), x(2), x(3), x(4)`
 - d) `x(1), x(2), x(3), x(4), x(5)`
- 12) Maximum number of elements in the declaration `int y[5][8];` is
 - a) 28
 - b) 32
 - c) 35
 - d) 40
- 13) Two-dimensional array elements are stored in
 - a) column major order
 - b) row major order
 - c) both options a and b
 - d) random order
- 14) Two-dimensional array elements are stored
 - a) row by row in the subsequent memory locations
 - b) column by column in the subsequent memory locations
 - c) row by row in scattered memory locations
 - d) column by column scattered memory locations
- 15) ANSI C recommends a compiler to support at least __ dimensions of an array:
 - a) 4
 - b) 5
 - c) 6
 - d) 7
- 16) Array declaration
 - a) requires the number of elements to be specified
 - b) does not require the number of elements to be specified
 - c) assumes default size as 0
 - d) is not necessary
- 17) Identify the wrong expression given `int a[10];`
 - a) `a[-1]`
 - b) `a[10]`
 - c) `a[0]`
 - d) `++a`
- 18) What is the value of `a[0][2]` in `int a[3][4] = {{1,2}, {4,8,15}};`
 - a) 4
 - b) 2
 - c) 0
 - d) not defined

- 19) To initialize a 5 element array all having value 1 is given by
- `int num[5]={1};`
 - `int num[4]={1,1,1,1,1};`
 - `int num[5]={1,1,1,1,1};`
 - `int num[]={1};`
- 20) To initialize a 5 element array all having 0 is given by
- `int num[5]={0};`
 - `int num[5]={0,0,0,0,0};`
 - options a and b
 - `int num[5]={1};`
- 21) The declaration `float f[][3]={{1.0},{2.0},{3.}};` represents
- A one-by-three array
 - A three-by-one array
 - A three-by-three array
 - A one-by-one array
- 22) A char array with the string value "aeiou" can be initialized as
- `char s[] = {'a','e','i','o','u'};`
 - `char s[] = "aeiou";`
 - `char s[] = {'a','e','i','o','u','\0'};`
 - options b and c
- 23) If the size of an array is less than the number of initialized as
- The extra values are neglected
 - It is an error
 - The size is automatically increased
 - The size is neglected
- 24) Identify the corrected statement
- Float array can be read as a whole
 - Integer array can be read as a whole
 - Char array can be read as a whole
 - Double array read as a whole
- 25) Missing elements of partly initialized arrays, are
- Set to zero
 - Set to one
 - Not defined
 - Invalid

Practice

1. Write a program to concatenate two strings.
2. Write a program in C to delete the duplicate character from an array and print the array again?
3. Write a program to sort the elements of an array.
4. Write a program in C to add, subtract and multiply two matrixes?
5. Write a program to find the transpose of a matrix?
6. Write a program to pick up the largest number from any 5-row by 5-column matrix?
7. Write a function that accepts two arguments: an array and its size n. It performs a bubble sort on the arrays elements. Use the indirection operator “ * ” instead of array subscript operator []?
8. Write a function that accepts two string as parameters and returns 1 if the first is a substring of second , and 0 otherwise ?
9. Write a program to remove the trailing blanks in a string input by the user , and print the resulting string. Verify that the string has actually been trimmed, since spaces are invisible when printed.
10. Write a program for sorting names of persons by swapping pointers instead of data. (Use linear sort method)?
11. Write a program for finding the smallest and largest in a list of N numbers. Accept the value of N at runtime and allocate the necessary amount of storage for storing numbers.

Chapter – VI: Pointers

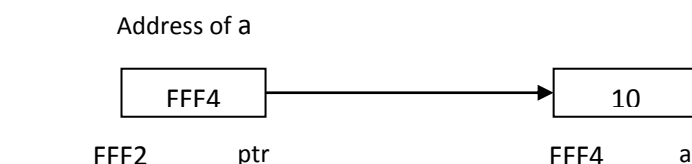
Introduction

A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in continuous groups i.e., memory cells are typically numbered serially. When we declare variables they generally occupy consecutive cells. For example, when we declare a character type of variable, it occupies one byte. For integer type of variables it occupies two bytes and for float type four consecutive memory cells are occupied. Now if we want to manipulate any variable then we need to reach the memory location where it is created. We generally access the variable by its name, thus the address corresponding the name is searched for, and then the manipulation occurs. Where as if we directly know the address it could have been much faster. For this purpose we use *pointers*. A pointer is a group of cells (often two or four) that can hold an address.

Lets consider the following declaration:

```
int num = 10;
```

We can get the address of num by using the & (address operator). Thus &num gives us the address/ memory location where variable num is created. But for further usage we need to store it another variable. So if we assign &num to any variable then the variable has to be of a special data type as memory locations are represented by hexadecimal notations.

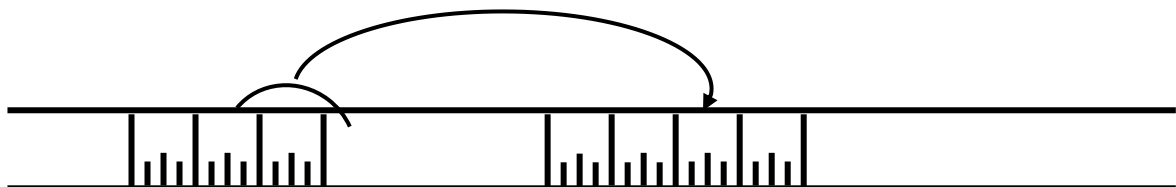


For example: if we assign `ptr=#`

then ptr is a special type of variable that can hold the address of a numeric type of variable. Here, ptr is a numeric type of pointer. The declaration of this pointer variable will be

`int *ptr;` which means ptr is a special type of variable which can store the memory location of an integer type of data. Typically a pointer occupied 2/4 bytes of memory.

If c is a char and p is a pointer that points to it, we could represent the situation this way:



The unary operator **&** gives the address of an object, so the statement

`p = &c ;` assigns the address of `c` to the variable `p` and `p` is said to “point to” `c`.

The unary operator **&** is the **address operator**, which only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants or register variables.

The unary operator ***** is the **indirection or de-referencing operator**; when applied to a pointer, it accesses the object the pointer points to.

Example 1:

```
int x=1, y=2, z[10];
int *ip;                /* ip is a pointer to integer */
ip=&x;                   /* ip now points to x */
y = *ip;                 /* y is now 1 */
*ip=0;                   /* x is now 0 */
ip = z;                  /* ip now points to z[0] */
```

Example 2:

*/*This program illustrates the relationship between two integer variables, their corresponding addresses and their associated values*/*

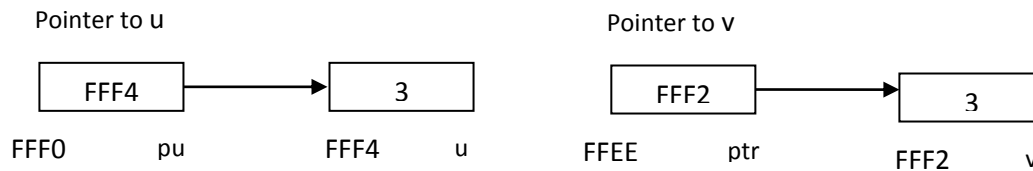
```
main()
{
    int u = 3, v;
    int *pu, *pv;        /* pointer to integer */
    pu=&u;                /* assign address of u to pu */
    v= *pu;               /* assign value of u indirectly to v */
    pv=&v;                /* assign address of v to pv */
    printf("\nu=%d  &u=%X  pu=%X  *pu=%d  &pu=%X", u, &u, pu, *pu, &pu);
    printf("\nv=%d  &v=%X  pv=%X  *pv=%d  &pv=%X", v, &v, pv, *pv, &pv);
}
```

Executing the program we will get the result as:


```
u=3    &u=FFF4    pu=FFF4    *pu=3        &pu=FFF0
v=3    &v=FFF2    pv=FFF2    *pv=3        &pv=FFEE
```


In the first line `u` represents the value stored within the variable `u`; `&u` represents the address/memory location of `u`, `pu` represents the value stored in the pointer `pu` i.e., the address of `u`; `*pu` represents the value stored in the variable whose address is stored in `pu` i.e., the value stored in `u` : 3; `&pu` represents the address of `pu`.

As we have assigned `*pu` to `v`, in this way `v` contains the value of `u` i.e., 3. In the second line `v` represents the value stored in `u`, `&v` represents the address/memory location of `v`, `pv` represents the value stored in the pointer `pv` i.e., the address of `v`; `*pv` represents the value stored in the variable whose address is stored in `pv` i.e., the value stored in `v` : 3; `&pv` represents the address of `pv`.



The unary operators `&` and `*` are members of the same precedence group as the other unary operators, i.e.; `-`, `++`, `--`, `!`, `sizeof` and `(type)`. This group of operator has a higher precedence than the groups containing the arithmetic operators and the associativity of unary operators are from right to left. (See chapter – 2).

 *The address operator (`&`) must act upon operands associated with unique addresses, such as ordinary variable or single array element. Thus the address operator can't act upon arithmetic expressions such as $2*(x+y)$.*

 *The indirection operator (`*`) can only act upon operands that are pointers (e.g., pointer variables). However, if `pv` points to `v` (`pv = &v`), then an expression such as `*pv` can be used interchangeably with its corresponding variable `v`. Thus, an indirect reference (`*pv`) can appear in place of an ordinary variable (`v`) within a more complicated expression.*

Example:

```
#include<stdio.h>
main()
{
    int u1,u2;
    int v = 3;
    int *pv;           /* pv points to v */
    u1=2*(v+5);        /* ordinary expression*/
    pv = &v;
    u2 = 2*(*pv + 5);   /* equivalent expression*/
    printf("\nu1 = %d    u2 = %d",u1,u2);
}
```

Output: u1 = 16 u2=16

Declaration of Pointers:

Like general variables pointers are also declared before they are used in a program. However, the way it is declared differs a little from that of a general variable. While declaring the pointer variable name has to proceed with an asterisk (`*`). The `*` defines the fact that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer. i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus a pointer can be declared as:

```
data-type *ptr ;
```

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

Example:

```
int *p;
float *fp;
```

Here p points to a integer quantity and fp points to a floating point quantity i.e., p can store the memory address of a integer type of variable and fp can store the memory address of a floating type variable.

Pointers and Function Arguments

Pointers can also be passed to function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion in altered form. This use of pointer for passing arguments is known as **passing reference argument**.

When C passes the arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function as the data item is copied to the function. Thus, any alteration to the data items inside the function alters only the copied data, which is not reflected on the original data items in the calling routine.

Example:

```
#include<stdio.h>
void swap(int,int);
main()
{
    int a,b;
    printf("Enter two number\n");
    printf("a = ");
    scanf("%d",&a);
    printf("b = ");
    scanf("%d",&b);
    swap(a,b);
    printf("\nInside the main().....\n");
    printf("a = %d    b = %d", a, b);
}
void swap(int a,int b)
{
    int temp;
    temp = a;
    a=b;
    b=temp;
    printf("\nInside the swap().....\n");
    printf("a = %d    b = %d", a, b);
}
```

When the above program is run, we get the following output.

Suppose, the input values as a =10 and b =20.

OUTPUT:

Inside the swap()

a = 20 b = 10

Inside main()

a = 10 b = 20

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above only swaps copies of a and b. If we need to change the original variables in the calling function, then indirectly the reference of the variables has to be passed i.e to pass pointers to the values to be changed. Thus, when a data item is passed by reference i.e., when a pointer is passed to the function, however the address of the data item is passed to the called function. By using the indirection operator the content of the address stored in the pointer can be accessed freely, either in the function or within the calling function. Thus any changes made to the data item are reflected both in the function and the calling routine. Thus, the use of pointer as a function argument enables the data item to be altered globally. For example: the above mentioned program can be rewritten where the function can be called in the following way.

swap(&a, &b); Since the operator & produces the address of a variable, *pa is a pointer to a. In swap itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

Example 1:

```
#include<stdio.h>
void swap(int *,int *);
main()
{
    int a,b;
    printf("Enter two number\n");
    printf("a = ");
    scanf("%d",&a);
    printf("b = ");
    scanf("%d",&b);
    swap(&a, &b);
    printf("\nInside the main().....\n")
    printf("a = %d      b = %d", a, b);
}
```

```

void swap(int *pa,int *pb)
{
    int temp;
    temp = *pa;
    *pa=*pb;
    *pb=temp;
}

```

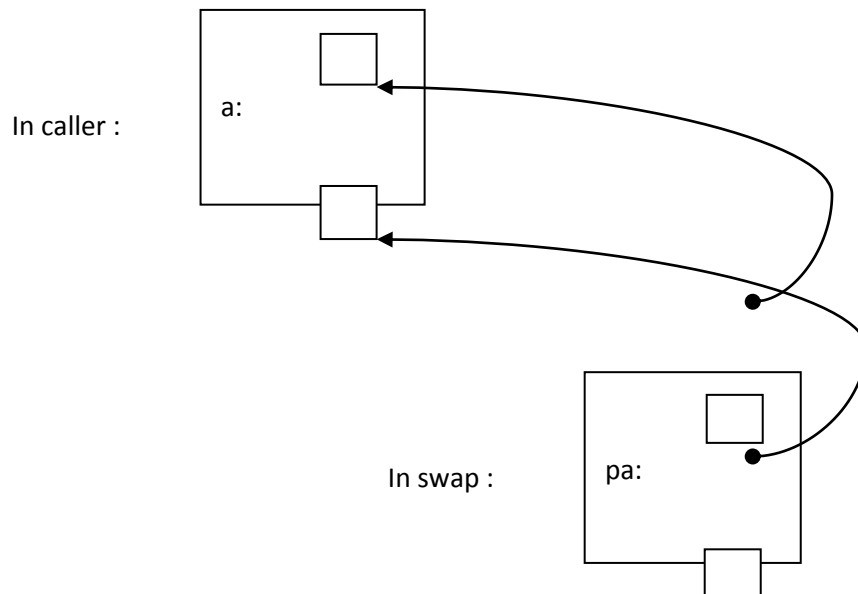
OUTPUT:

Inside the swap()

a = 20 b = 10

Inside main()

a = 10 b = 20



Example 2:

```

/*Count the number of vowels, consonants, digits, white space
characters, and other characters*/
#include<stdio.h>
void scan_line(char line[], int *pv, int *pc, int *pd, int *pw, int
*po);
main()
{
    char line[80];          /*line of text*/
    int vowels = 0;         /*vowel counter*/
    int consonant=0; /*consonant counter*/
    int digits=0;           /*digit counter*/
    int whitespace=0; /*white space counter*/
    int other=0;            /*other character counter*/
    printf("\nEnter a line of text below:\n");
    scanf("%s",line);
}

```



```

        scan_line(line,    &vowels,    &consonant,    &digits,    &whitespace,
&other);
    printf("\nNo. of vowels: %d",vowels);
    printf("\nNo. of consonants: %d", consonants);
    printf("\nNo. of digits: %d",digits);
    printf("\nNo. of whitespace: %d", whitespace);
    printf("\nNo. of other: %d",other);
}
void scan_line(char line[], int *pv, int *pc, int *pd, int *pw, int *po)
{
    char c;
    int count=0;
    while((c = toupper(line[count]))!='\0')
    {
        if( c=='A' || c=='E' || c=='I' || c=='O' || c=='U' )
            ++*pv;                                /*vowel*/
        else if(c>='A' && c<='Z')
            ++*pc;                                /*consonant*/
        else if (c>='0' && c<='9')
            ++*pd;                                /*digit*/
        else if (c == ' ' || c == '\t')
            ++*pw;                                /*white spaces*/
        else
            ++*po;                                /*other characters*/
        ++count;
    }
}

```

Sample Output:

Enter a line of text below:

C is a general-purpose structured programming language invented in 1970.

No. of vowels: 22

No. of consonants: 36

No. of digits: 4

No. of whitespace: 9

No. of other: 1

Pointers and One Dimensional Arrays

Address Arithmetic

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays and address arithmetic is one of the strength of the language. We have discussed in chapter – 5 that array names store the address of the first element in the array. So. Now after discussing the concept of pointers, we can say that array name is actually a pointer that holds the address of the first element of an array. For example, if `arr` is an array, then the address of the first element can be represented as `&arr[0]` or simply the name `arr`, similarly the second element can be represented as `&arr[1]` or `arr+1`. In general the address of $(i+1)^{th}$ element in an array can be presented as `&arr[i]` or `(arr+i)`. The expression `arr+i` is a symbolic representation of an address specification rather than an arithmetic expression. Thus, we have two different ways to write the address of any array element: we can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

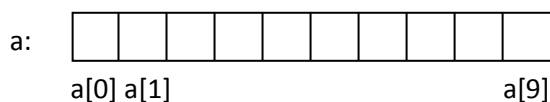
The memory cell associated with an array element will depend on the data type of the array as well as the compiler architecture. E.g., in a 16-bit compiler `int` declaration occupies 2 bytes, `float` declaration occupies 4 bytes where as in a 32-bit compiler `int` declaration occupies 4 bytes, `float` declaration occupies 8 bytes. When writing the address of an array element in the form `(arr+i)`, however, the C programmer need not be concerned with the number of memory cells associated with each type of array element as the C compiler adjust them automatically. The programmer must specify only the address of the first array element (the name of the array) and the subscript value. The value of subscript is also referred to as the offset.

Any operation that can be achieved by array subscripting can also be done with pointers.

Example:

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

Defines an array `a` of size 10, i.e, a block of 10 consecutive objects named `a[0],... a[9]`.



The notation `a[i]` refers to the i -th element of the array.

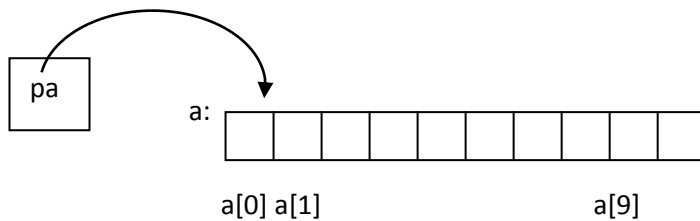
If `pa` is a pointer to an integer declared as,

```
int *pa;
```

Then the assignment

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

Sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`.



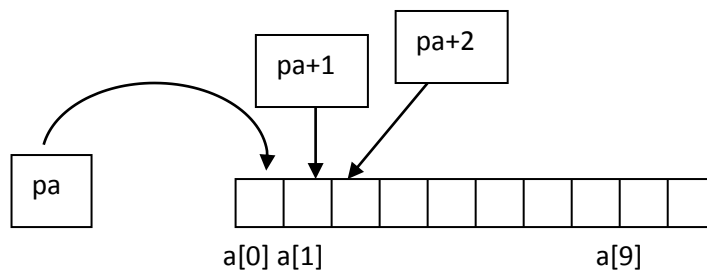
Now the assignment

`x=pa`

Will copy the contents of `a[0]` into `x`.

If `pa` points to a particular element of an array, then by definition `pa + 1` points to the next element, `pa + i` points to `i` elements after `pa` and `pa - i` points `i` elements before. This if `pa` points to `a[0]`,

`*(pa+1)` refers to the contents of `a[1]`, `pa+i` is the address of `a[i]`, and `*(pa + i)` is the contents of `a[i]`.



Pointers and Arrays – A Difference

- ❑ A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; construction like `a=pa` and `a++` are illegal. Thus, it is not possible to assign an arbitrary address to an array name or to an array element.
- ❑ When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable so array name is a pointer, i.e., variable containing the address.

As we have discussed already that array name is actually a pointer to the first element within the array. Therefore, it should be possible to define the array as the pointer variable rather than a conventional array. Syntactically both are equivalent; however there is some difference lies in them. The conventional array results in a fixed block of memory being reserved at the beginning of the program execution, whereas this does not occur if the array is represented in terms of a pointer variable. As a result the use of a pointer variable to represent an array requires some sort of initial memory assignment before the array elements are processed which can be accomplished by using the `malloc()` library function.

Dynamic Memory Allocation

Memory Allocation Functions

alloc(n) : Returns a pointer p to n consecutive character positions, which can be used by the caller of alloc for storing characters. The storage managed by alloc is a stack.

calloc: Allocates a block of memory

farmalloc: Allocates memory from far heap

malloc: Allocates a block of memory

realloc: Reallocates a block of memory initially allocated using calloc()/malloc()

afree(p) : It releases the storage thus acquired so that it can be reused later. The storage managed by afree is a stack.

Free: Frees a block allocated with calloc()/malloc().

Character Pointers and Functions

- **strlen()**: This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main()
{
    char arr[] = "beautiful";
    int length;
    length = strlen(arr);
    printf("The length of the string represented by arr is
%d", length);
    return 0;
}
```

The output is:

The length of the string represented by arr is 9

- **strcpy()**: This function copies the content of one string into another. The base address of the source and target string should be supplied to this function. Its usage is illustrated in the following program.

```
main()
{
    char source[] = "beautiful", destination[];
    strcpy ( target, source);
    printf("The source string is %s", source);
    printf("The target string is %s", destination);
}
```

The output is:

The source string is beautiful

The target string is beautiful

- **strncpy():** This function copies the first n characters of the content of one string into another. The base address of the source and target string and no of characters should to be supplied to this function. Its usage is illustrated in the following program.

```
main()
{
    char source[] = "beautiful", destination[];
    strncpy ( target, source, 5);
    printf("The source string is %s", source);
    printf("The target string is %s", destination);
    return 0;
}
```

The output is:

The source string is beautiful

The target string is beaut

- **strcat():** This function concatenates the source string at the end of the target string. Its usage is illustrated in the following program.

```
main()
{
    char source[] = "Beautiful", destination[] = "World";
    strcat ( target, source);
    printf("The source string is %s", source);
    printf("The target string is %s", destination);
    return 0;
}
```

The output is:

The source string is Beautiful

The target string is BeautifulWorld

- **strcmp():** This is a function which compares two strings to find out whether they are same or different. If the two strings are equal strcmp() returns 0, if the first string is greater than the second string it returns 1 and if the first string is less than the second string it returns -1. Its usage is illustrated in the following program.

```

main()
{
    char    str1[]="Beautiful",    str2[]="beautiful",
    str3[]="Beautiful";
    printf("%d %d %d", strcmp( str1, str2), strcmp( str1,
    str3), strcmp( str2, str3));
    return 0;
}

```

The output is:

-1 0 1

Two Dimensional Arrays

A two dimensional array is a grid containing rows and columns in which each element is uniquely specified by means of its row and column coordinates. Character two-dimensional arrays hold an array of strings wherein a row represents a string and a column represents a single character in the string. In a two-dimensional array elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializer in braces; each row of a two-dimensional array is initialized by a corresponding sub list.

Two Dimensional Arrays - Declaration

The general form of declaration of a character two-dimensional array is

`char arrayname[x][y];` Where

x – number of rows

y – number of columns

The following guidelines need to be followed while declaring character two-dimensional arrays. The number of rows is the number of strings that are going to be part of the array.

The length of the longest string plus one would be the column specification.

Initialising Two-dimensional Arrays

The rules for initialising a two-dimensional array are the same as for a one-dimensional array.

```

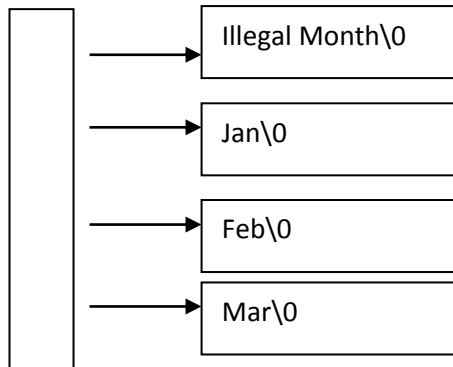
/* month name : return name of n-th month*/
char *month_name(int n)
{
    static char *name[ ] = { "Illegal month", "January", "February",
    "March", "April", "May", "June", "July", "August", "September",
    "October", "November", "December" };
    return (n<1 || n>12) ? Name[0] : name[n];
}

```

}

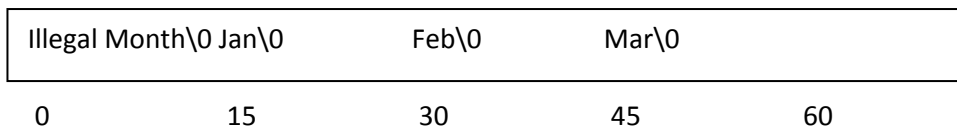
For an array of pointers

```
char *name[] = {"Illegal month", "jan", "feb", "mar"};
```



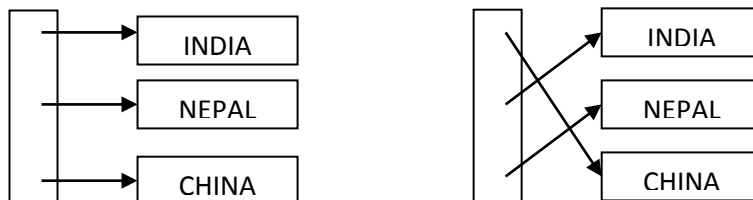
For a two-dimensional array:

```
char name[ ][15] = {"Illegal month", "jan", "feb", "mar"};
```



Array of Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can.



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

In general terms , a two dimensional array can be defined as a one-dimensional array of pointers by writing:

```
data-type *array[expression - 1];
```

rather than the conventional array definition.

Similarly a n-dimensional array can be defined as (n-1) dimensional array of pointers by writing

```
data-type *array[expression - 1 ][expression - 2 ] .... [expression - n-1 ];
```

In the above declarations **data-type** refers to the data-type of the original n-dimensional array, **array** is a array name and **expression – 1, expression – 2.....expression– n** are positive valued integer expressions that indicate the maximum number of elements associated with each subscript.

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

Please note that the last (rightmost) expression is omitted when defining an array of pointers, where as the first (the left most) expression is omitted when defining a pointer to a group of arrays.

When an n- dimensional array is expressed in this manner, an individual array element within n-dimensional array can be accessed by a single use of indirection operator.

For example:

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

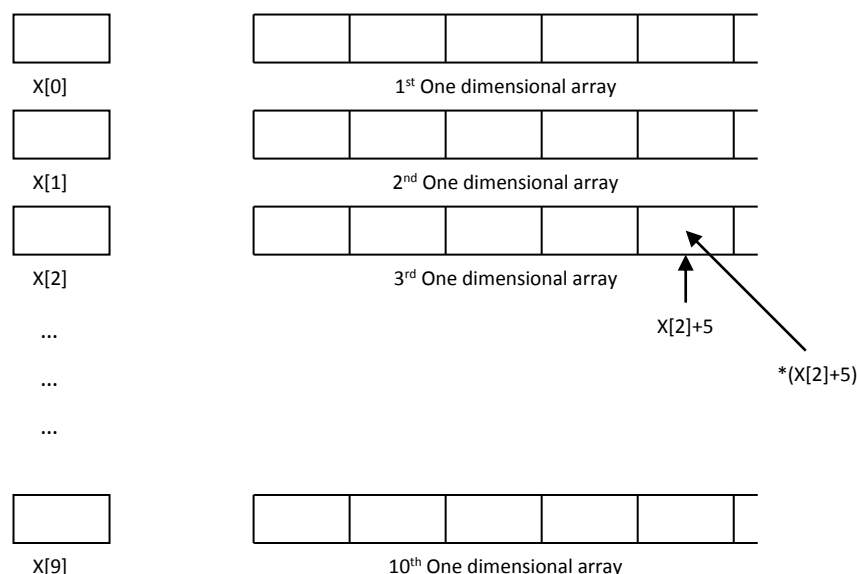
```
int*x[10];
```

Hence, x[0] points to the beginning of the first row, x[1] points to beginning of second row, and so on. Here, the number of arrays in each row is not explicitly specified.

An individual array elements, say x[2][5] can be accessed by writing

```
*(x[2] + 5)
```

The expression, x[2] is a pointer to the first element in the row 2, so that (x[2]+5) points to element 5 i.e, the sixth element, within row-2

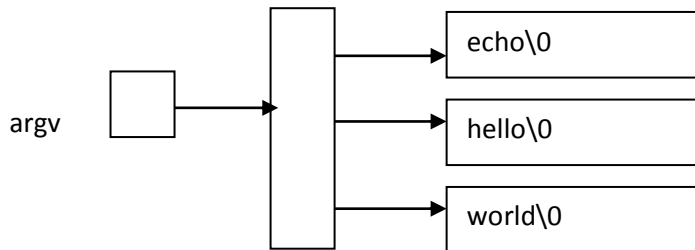


Command Line Argument

In environments that support C, there is a way to pass command line arguments or parameters to a program when it begins executing. When main is called it is called with two arguments – the first (argc, for argument count) in the no. of command line arguments the program was invoked with; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.

Example: echo hello, world

```
main(int argc, char *argv[ ])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) > " " : "");
    printf("\n");
return 0;
}
```



Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

char **argv

argv: pointer to pointer to char

int (*daytab) [13]

daytab: pointer to array[13] of int

int *daytab[13]

daytab: array[13] of pointer to int

void *comp()

comp: function returning pointer to void

void (*comp) ()

comp: pointer to function returning void

char (*(*x()) []) ()

x : function returning pointer to array[] of pointer to function returning char

char (*(*x[3]) ()) [5]

x : array[3] of pointer to function returning pointer to array[5] of char

MCQs

- 1) Pointers are supported in
 - a) FORTRAN
 - b) PASCAL
 - c) C
 - d) both options b and c
- 2) Pointer variable may be assigned
 - a) an address value represented in hexadecimal
 - b) an address value represented in octal
 - c) the address of another variable
 - d) An address value represented in binary
- 3) A pointer value refers to
 - a) an integer constant
 - b) a float value
 - c) any valid address in memory
 - d) any ordinary variable
- 4) Identify the correct declaration of pointer variables p1, p2.
 - a) int pi, p2;
 - b) int *pl, p2;
 - c) int pi, *p2;
 - d) int *pl, * p2;
- 5) The operators exclusively used in connection with pointers are
 - a) * and /
 - b) & and *
 - c) & and |
 - d) - and >
- 6) Identify the invalid expression for given register int r = 10;
 - a) r = 20
 - b) &r
 - c) r+15
 - d) r/10
- 7) Identify the invalid expression.
 - a) &274
 - b) &(a+b)
 - c) &(a*b)
 - d) all the above
- 8) Identify the wrong declaration statement
 - a) int*p,a=10;
 - b) int a = 10, *p = &a.
 - c) int*p=&a,a=10;
 - d) options a and b
- 9) Identify the invalid expression given
int num = 15, *p = #
 - a) *num
 - b) *(&num)
 - c) *&*num
 - d) **&p
- 10) Identify the invalid expression for given
float x = 2.14, -y = &x;
 - a) &y
 - b) *(&num)
 - c) **&y
 - d) (*&)x
- 11) The operand of the address of operator is
 - a) a constant
 - b) an expression
 - c) a named region of storage
 - d) a register variable
- 12) How does compiler differentiate address of operator from bitwise AND operator ?
 - a) By using the number of operands and position of operands
 - b) By seeing the declarations
 - c) Both options a and b
 - d) By using the value of the operand
- 13) How does compiler differentiate address of operator from multiplication operator ?
 - a) By using the number of operands
 - b) by seeing the declarations
 - c) both options a and b
 - d) by using the value of the operand
- 14) The address of operator returns
 - a) The address of its operand
 - b) lvalue
 - c) both options a and b
 - d) rvalue
- 15) The indirection operator returns
 - a) The data object stored in the address represented by its operand
 - b) Lvalue
 - c) both options a and b
 - d) rvalue
- 16) The operand of indirection operator is
 - a) pointer variable
 - b) pointer expression
 - c) both options a and b
 - d) ordinary variable
- 17) The operand of address of operator may be
 - a) an ordinary variable
 - b) an array variable
 - c) a pointer variable
 - d) Any one of the above
- 18) Identify the invalid lvalue given
int x, *p = &x;
 - a) *(p+1)
 - b) *(p-3)
 - c) both options a and b
 - d) &x
- 19) After the execution of the statement int x;
the value of x is
 - a) 0
 - b) undefined
 - c) 1
 - d) -1
- 20) Pointer variable may be initialized using
 - a) Static memory allocation
 - b) Dynamic memory allocation

- c) Both options a and b
 - d) A positive integer
- 21) Identify the invalid pointer operator.
- a) &
 - b) *
 - c) >>
 - d) none of the above
- 22) Assume 2 bytes for int, 4 bytes for float and 8 bytes for double data types respectively, How many bytes are assigned to the following pointer variables?
- int *ip; float *fp; double *dp;**
- a) 2 bytes for ip, 4 bytes for fp and 8 bytes for dp
 - b) 2 bytes for all pointer variables ip, fp and dp
 - c) one byte for ip, 2 bytes for fp and 4 bytes for dp
 - d) 2 bytes for ip and 8 bytes for each fp and dp
- 23) The number of arguments used in malloc() is
- a) 0
 - b) 1
 - c) 2
 - d) 3
- 24) The number of arguments used in calloc() is
- a) 0
 - b) 1
 - c) 2
 - d) 3
- 25) The number of arguments used in realloc() is
- a) 0
 - b) 1
 - c) 2
 - d) 3
- 26) The function used for dynamic deallocation of memory is
- a) Destroy()
 - b) Delete()
 - c) Free()
 - d) Remove()
- 27) The function call realloc (ptr, 0) is
- a) same as free(ptr)
 - b) used to clear the values in the address represented by ptr
 - c) used to set the value of ptr to be 0
 - d) invalid
- 28) In the expression *cp++
- a) *cp is evaluated first and *cp is incremented by 1.
 - b) *cp is evaluated first and cp is incremented by 1.
 - c) cp is incremented by 1 first and * is applied.
 - d) cp is incremented by 1 first and -is applied to the previous value of cp.
- 29) The pointers can be used to achieve
- a) call by function
 - b) call by reference
 - c) call by name
 - d) call by procedure
- 30) The operators &, *, ++ and - have
- a) same precedence level and same associativity
 - b) same associativity and different precedence level
 - c) different precedence level and different associativity
 - d) different precedence level and same associativity

Practice

1. Write an appropriate declaration for each of the following situations:
 - a. Declare two pointers whose objects are integer variables x and y.
 - b. Declare a pointer to a floating-point quantity and a pointer to a double precision quantity.
 - c. Declare a function that accepts two integer arguments and returns a pointer to a long integer.
 - d. Declare a function that accepts two arguments and returns a long integer. Each argument will be a pointer to an integer quantity.
 - e. Declare a one-dimensional floating-point array using pointer notation.
 - f. Declare a two-dimensional floating-point array with 30 rows and 50 columns, using pointer notation.
 - g. Declare an array of strings whose initial values are "red", "green" and "blue".
 - h. Declare a function that accepts another function as an argument and returns an integer quantity. The function passed, as an argument will accept an integer argument and returns an integer quantity.
 - i. Declare a pointer to a function that accepts three integer arguments and return an floating-point quantity.
 - j. Declare a pointer to a function that accepts three pointers to integer quantities as arguments and returns a pointer to a floating-point quantity.
2. A program contains the following statements:

```
char u, v, = 'A';  
char *pu, *pv=&v;  
...  
*pv=v+1;  
u=*pv+1;  
pu=&u;
```

Suppose each character occupies one byte of memory. If the value assigned to 'u' is stored in (hexadecimal) address F8C and the value assigned to 'v' is stored in address F8D, then

- a. What value is represented by &v?
 - b. What value is assigned to pv?
 - c. What value is represented by &v?
 - d. What value is assigned to u?
 - e. What value is represented by &u?
 - f. What value is assigned to pu?
 - g. What value is represented by *pu?
3. A C program has the following statements:

```
int x, y=25;
```

```
int *px, *py=&y;
```

```
...
```

Suppose each integer quantity occupies 2 bytes of memory. If the value assigned to 'x' begins at (hexadecimal) address F9C and the value assigned to 'y' begins at address F9E. Then,

- a. What value is represented by &x?
 - b. What value is represented by &y?
 - c. What value is assigned to py?
 - d. What value is assigned to *py?
 - e. What value is assigned to x?
 - f. What value is represented by px?
 - g. What final value is assigned to *px?
 - h. What value is represented by (px+2)?
 - i. What value is represented by the expression (*px+2)?
 - j. What value is represented by the expression (*py+2)?
4. A C program has the following statements:

```
float a = 0.001, b = 0.003;
```

```
float c, *pa, *pb;
```

```
pa=&a;
```

```
*pa = 2 * a;
```

```
pb=&b;
```

```
c = 3 * ( *pb - *pc );
```

Suppose each floating-point number occupies 4 bytes of memory. If the value assigned to 'a' begins at (hexadecimal) address 1130 and the value assigned to 'b' begins at address 1134 and the value 'c' begins at 1138. Then,

- a. What value is assigned to &a?
- b. What value is assigned to &b?
- c. What value is assigned to &c?
- d. What value is assigned to pa?
- e. What value is represented by *pa?
- f. What value is represented by &(*pa)?
- g. What value is assigned to pb?
- h. What value is represented by *pb?

- i. What final value is assigned to c?
5. A C program has the following declaration:


```
static int x[8] = {10, 20, 30, 40, 50, 60, 70, 80};
```

 - a. What is the meaning of x?
 - b. What is the meaning of x+2?
 - c. What is the value of *x?
 - d. What is the value of *(x+2)?
 - e. What is the value of (*x+2)?
6. A C Program contains the following declaration:


```
static float table[2][3]= {
                                {1.1, 1.2, 1.3},
                                {2.1, 2.2, 2.3}
                                };
```

 - a. What is the meaning of **table**?
 - b. What is the meaning of (table+1)?
 - c. What is the meaning of *(table+1)?
 - d. What is the meaning of (*(table+1)+1)?
 - e. What is the meaning of *(table)+1)?
 - f. What is the value of (*(table+1)+1)?
 - g. What is the value of *(table)+1)?
 - h. What is the value of (*(table+1))?
 - i. What is the value of (*(table+1)+1)+1?
7. Explain the meaning of each of the following declarations.
 - a. float (*x)(int *a);
 - b. float (*x(int *a))[20];
 - c. float x(int (*a));
 - d. float x(int *a);
 - e. float *x(int a[]);
 - f. float *x(int (*a));
 - g. float *x(int *a[]);
 - h. float (*x)(int (*a));
 - i. float (*x)(int *a[]);
 - j. float (*x[20])(int a);
 - k. float (*x[20])(int *a);

Chapter VII: User Defined Data Types

Introduction

Till now we have dealt only with fundamental datatypes, which we used, in the form of ordinary variables and arrays. Now the question arises whether it is sufficient to work or not! Lets discuss the limitations of using only fundamental data types.

Ordinary variables can hold only one piece of information at a time, arrays can represent more information but all of the same data type. But in so many cases we come across a situation where we deal with entities having a group of dissimilar data types called attributes. For example, say we want to store the information for some students. Now a student has a roll number, he has a name, sex, is enrolled for a course, has paid some fees and also has secured some marks. These all are different attributes or properties of a single student. So, multiple students, each of them has all the attributes but values in the attributes are different. So, the solution can be array. But there is a problem; all the attributes are of dissimilar data types. Roll number is integer, name and course are strings, sex is represented by a single character (M/F), fees are represented by float and marks can be represented by integer.

We can construct individual arrays, one for storing roll number, one for name, one for sex, one for course, one for fees and one more for marks and then accept all roll numbers, all names and so on. In this way, person 1's roll number may contain person 4's name, person 3's marks etc. This happens because the arrays exist separately having no relationship with each other. To solve this problem, the C language allows the user to create and use data type other than the fundamental data type known as **user defined data types**. Such a user-defined data type is structure, which can be declared by using the keyword structure or struct.

A structure is a collection of one or more variable possibly of different types grouped under a single name for convenient handling.

Structures represent a group of data those are logically related to each other in some languages known as records. Structures help in organizing complicated data as they permit a group of related variables to be treated as a single unit.

These data types are called user-defined data types. Data types are defined using the keyword structures or struct.

- Structure is a collection of variables that are preferences by a single name.
- The variables can be of different types.
- They provide a convenient means of keeping related information together.
- Structure definition forms a template that is used to create structure variable.
- The variables that make up the structure are called structure elements.

Declaring a Structure

The following code segment shows how a structure template that defines an invoice number and invoice amount be created. The keyword `struct` tells the compiler that a structure template is being declared.

```
struct student
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
};
```

Notice that a semi-colon terminates the declaration. This is because the structure declaration is a statement. Also, the structure tag `student` identifies this particular data structure. At this point in the code, no variable has been declared; only the form of the data has been defined. To declare a variable, you would write

```
struct student s;
```

This defines a structure variable of type 'student', called `s`. When a structure is declared, in essence, a data type is getting defined. The compiler automatically allocates sufficient memory to accommodate all the elements that make up the structure. In the structure `student`, when we declare a structure variable `s`, 24 bytes of memory is reserved. (2 byte for roll no, 10 bytes for name, 1 byte for sex, 5 bytes for course, 4 bytes for fees and 2 bytes for marks.) More than one variables can also be declared at that time of declaring a structure.

For example:

```
struct student
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
} s1,s2;
```


or

```
struct student s1,s2;
```

will declare a structure type called student and declare the variables s1 and s2.

The general form of declaration is:

```
struct <tag>
{
    <type><variable1>;
    <type><variable1>;
}<struct vars>;
```

where

<tag> - is the name of the structure declaration, effectively the name of the new data type. The tag is optional.

<struct vars>-names of structure variables.

Example:

```
struct
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
} s1,s2;
```

In this case the structure variables should be defined along with the definition, as we do not have a tag we won't be able to refer to it afterwards.

So, we can summarize the points to remember while declaring a structure variable as follows:

- A semi-colon must follow the closing brace in the structure type declaration.
- It is important to understand that a structure type declaration does not tell the compiler to reserve any space in the memory. All a structure declaration does is, it defines the 'form' of the structure.
- Usually structure type declaration appears at the top of the source code file, before any variable or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using # include) in the program that wants to use the struct.

Accessing Structure Elements

The dot (.) Operator

Individual structure elements can be referenced by combining the .(dot) operator and the name of the structure variable. For example, the following statement assigns 75 marks to the element marks of the structure variable s1 declared earlier.

```
s1.marks=75;
```

The dot (.) Operator (example)

```
#include<stdio.h>
struct student
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
};
main()
{
    struct student s1;
    char creply;
    do
    {
        printf("Enter roll No.\t");
        scanf("%d",&s1.roll_no);
        printf("Enter Name\t");
        scanf("%s",s1.name);
        printf("Enter Sex.\t");
        scanf("%c",&s1.sex);
        printf("Enter Course\t");
        scanf("%s",s1.course);
        printf(" Enter Fees\t");
        scanf("%f",& s1.fees);
        printf("Enter Marks\t");
        scanf("%d",&s1.marks);
```

```

        printf(" \n\nRoll No\t:%d \n Name\t: %s \n Sex\t:%c \n
        Course\t:%s \n Fees\t:%f \n Marks\t:%d\n",
        s1.rno,s1.name,s1.sex,s1.course,s1.fees,s1.marks);
        printf(" Press y to continue : ");
        creply=getche();
    }while(creply=='y');
    return 0;
}

```

Output:

```

Enter roll No.    1
Enter Name       Sumit
Enter Sex.       M
Enter Course     JBC
Enter Fees       4500.00
Enter Marks      89

```

```

Roll No          :1
Name             :Sumit
Sex              :M
Course           :JBC
Fees             :4500.00
Marks            :89
Press y to continue : n

```

Pointer to Structure (The -> Operator):

The -> operator is used when a structure element has to be accessed through a pointer. The pointer should be initialized before it can access any structure element. Consider the following code fragment, which is used to initialize pointers with regards to the user-defined data type invoice created earlier.

```

student *sptr;

sptr=(struct student *)malloc (sizeof(struct student));

```

Another way of initializing pointers would be

```

student s1, *sptr;

sptr=&b1;

```

The -> Operator (Example)

```
#include<stdio.h>

char creply;
struct student
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
}

main()
{
    struct student*sptr;
    sptr=(struct student *)malloc (sizeof(struct student));

    do
    {
        printf("Enter roll No.\t");
        scanf("%d",&sptr->roll_no);
        printf("Enter Name\t");
        scanf("%s",sptr->name);
        printf("Enter Sex.\t");
        scanf("%c",s1->sex);
        printf("Enter Course\t");
        scanf("%s",s1->course);
        printf(" Enter Fees\t");
        scanf("%f",& s1->fees);
        printf("Enter Marks\t");
        scanf("%d",&s1->marks);
        printf(" \n\nRoll No\t:%d \n Name\t: %s \n Sex\t:%c \n
        Course\t:%s \n Fees\t:%f \n Marks\t:%d\n", s1->rno,s1-
        >name,s1->sex,s1->course,s1->fees,s1->marks);
        printf(" Press y to continue : ");
        creply=getche();
    }while(creply=='y');
    return 0;
}
```

Output:

Enter roll No. 1
Enter Name Sumit
Enter Sex. M
Enter Course JBC
Enter Fees 4500.00
Enter Marks 89

Roll No :1
Name :Sumit
Sex :M
Course :JBC
Fees :4500.00
Marks :89
Press y to continue : n

How structure elements are stored?

What ever may be the data type of elements of the structure, they always are stored in contiguous memory locations.

Example:

```
struct student
{
    int rollno;
    char name[10];
    char sex;
    char course[5];
    float fees;
    int marks;
}

struct student s1={1,"SUMIT",'M',"JBC",4500.00,89};
```

When we declare a structure variable it occupies 24 contiguous bytes in memory.

s1.rollno s1.name s1.sex s1.course s1.fees s1.marks

1	SUMIT	M	JBC	4500.00	89
---	-------	---	-----	---------	----

Array of Structure

Like fundamental data types, array of structure can also be declared to represent continuous information. For example, in the following structure week which contains the weekday and the day name. Weekday is an array of week type that contains complete information regarding a weekday.

```
struct week
{
    int weekno;
    char day[8];
};

struct week weekday[]={
    {1,"Monday"},
    {2,"Tuesday"},
    {3,"Wednesday"},
    {4,"Thursday"},
    {5,"Friday"},
    {6,"Saturday"},
    {7,"Sunday"}
};
```

sizeof()

C provides a compiled-time unary operator called sizeof that can be used to compute the size of an object.

The expressions

sizeof object

And

sizeof (type name)

Yield an integer equal to the size of the specified object or type in bytes.

Structures within Structures – nested structure

A structure can have other structures as members. References to the members of the inner structure include the names of both structure variables, as the example given below.

```
#include<stdio.h>
/*Date structure.*/
struct date
{
    int month;
    int day;
    int year;
};

// Employee structure
```

```

struct employee
{
    int empno;
    float salary;
    date datehired;
};
//The main()
int main()
{
    /*An initialised Employee structure*/
    employee ravi = {123, 35500, {5, 17, 82}};
    /*Display the employee information.*/
    printf( " Emp # : %d\n Salary:%d\n Date hired   :%d",    ravi.empno,
    ravi.salary,          ravi.datehired.Month,          ravi.datehired.day,
    ravi.datehired.year);
    return 0;
}

```

Self-referential structures

```

struct tnode                /*the tree node:*/
{
    char *word;              /* points to the text*/
    int count;               /*number of occurrences*/
    struct tnode *left;      /*left child*/
    struct tnode *right;     /*right child*/
};

```

A Complete Linked list program using dynamic allocation of memory

```

#include<stdio.h>

#include<malloc.h>
#define NULL '\0'

struct node
{
    int info;
    struct node *next;
};

struct node *start=NULL;
/*NODE CREATION*/
/*function to allocate memory for a new node*/
struct node*create_node(int data)
{
    struct node *nw;
    nw = (struct node *)malloc(sizeof(struct node));
    nw -> info = data;
}

```

```

        nw -> next = NULL;
        return nw;
    }
    /* INSERTION */
    /*function to add node in the beginning of the linked list*/
    void add_beg(struct node *nw)
    {
        if (start == NULL)
            start=nw;
        else
        {
            nw->next=start;
            start=nw;
        }
    }
    /*function to add node in the end of the linked list*/
    void add_end(struct node *nw)
    {
        struct node *ptr;
        if (start == NULL)
            start=nw;
        else
        {
            for( ptr=start; ptr->next != NULL; ptr=ptr->next);
            ptr->next=nw;
        }
    }
    /*function to add node at a particular position of the linked list*/
    void add_pos(struct node *nw, int pos)
    {
        int choice,position;
        struct node *ptr;
        if (start == NULL)
        {
            printf("Linked list is empty, would you like to add the node
            ?(1/0)");
            scanf("%d",&choice);
            if (choice==1)
                start=nw;
        }
        else
        {
            for(ptr=start,position=1;position<pos&&ptr;position++,ptr=ptr
            ->next);
            if (ptr != NULL)
            {
                nw -> next = ptr -> next;
                ptr -> next = nw;
            }
            else
            {
                printf("\nThe position entered is outside the list");
                printf("\nWould you like to add at the end ?(1/0)");
                scanf("%d",&choice);
                if (choice==1)

```



```

        add_end(nw);
    }
}
/* DELETION*/
/*function to delete node from the beginning of the linked list*/
void del_beg()
{
    if (start == NULL)
        printf("Sorry, can't delete. The list is empty");
    else
        start=start->next;
}
/*function to delete node from the end of the linked list*/
void del_end()
{
    struct node *ptr;
    if (start == NULL)
        printf("Sorry, can't delete. The list is empty");
    else
    {
        for( ptr=start; (ptr->next)->next != NULL; ptr=ptr->next);
        ptr->next= NULL;
    }
}
/*function to delete node from a particular position of the linked list*/
void add_pos(int pos)
{
    int choice,position;
    struct node *ptr;
    if (start == NULL)
        printf("Sorry, can't delete. The list is empty");
    else
    {
        for( ptr=start, position=1; position < pos && ptr ; position
        ++,ptr=ptr->next);
        if (ptr != NULL)
            ptr -> next = (ptr -> next) -> next;
        else
        {
            printf("\nThe position entered is outside the list");
            printf("\nWould you like to delete the last node
            ? (1/0)");
            scanf("%d",&choice);
            if (choice==1)
                delete_end();
        }
    }
}
/*TRAVERSAL */
void traverse( )
{
    struct node *ptr;
    if (start == NULL)

```

```

        printf("EMPTY LIST");
    else
    {

        for( ptr=start; ptr; ptr=ptr->next)
            printf("%d \t",ptr->info);

    }
}

/*MODIFY*/
/*function to modify the information in a node at a particular position
of the linked list*/
void modify(int pos, int nwinfo)
{
    int position;
    struct node *ptr;
    if (start == NULL)
        printf("Linked list is empty);
    else
    {
        for( ptr=start, position=0; position < pos && ptr ; position
        ++,ptr=ptr->next);
        if (ptr != NULL)
            ptr -> info = nwinfo;
        else
            printf("\nThe position entered is outside the list");
    }
}

/*MAIN()*/
main()
{
    struct node *nw;
    int choice,choice1,data,position,nwdata;
    do
    {
        printf("\nLINKED LIST OPERATIONS");
        printf("\n\n1. INSERT");
        printf("\n2. MODIFY");
        printf("\n3. DELETE");
        printf("\n4. TRAVERSE");
        printf("\n5. EXIT");
        printf("\n\n Enter your choice");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("\nEnter information for the new node");
                scanf("%d",&data);
                nw=(struct node *)malloc(sizeof(struct node));
                nw=create_node(nw);
                printf("\n1. INSERT NODE IN THE BEGINNING");

                printf("\n2. INSERT NODE AT THE END");

```

```

printf("\n3.  INSERT  NODE  AT  A  PARTICULAR
POSITION");
printf("\n4. RETURN");
printf("\n\n Enter your choice");
scanf("%d",&choice1);
switch(choice)
{
    case 1:
        add_beg(nw);
        break;
    case 2:
        add_end(nw);
        break;
    case 3:
        printf("\nEnter the position");
        scanf("%d",&position);
        add_pos(nw,position);
        break;
    case 4:
        break;
}
}while(choice1!=4);
case 2:
    printf("\nEnter the position of the node you want
to change");
    scanf("%d",&position);
    printf("\nEnter the new information");
    scanf("%d",&nwdata);
    modify(position,nwdata);
    break;
case 3:
    printf("\n1. DELETE NODE FROM THE  BEGINNING");
    printf("\n2. DELETE NODE FROM THE  END");
    printf("\n3.  DELETE  NODE  AT  A  PARTICULAR
POSITION");
    printf("\n4. RETURN");
    printf("\n\n Enter your choice");
    scanf("%d",&choice1);
    switch(choice)
    {
        case 1:
            del_beg();
            break;
        case 2:
            del_end();
            break;
        case 3:
            printf("\nEnter the position");
            scanf("%d",&position);
            del_pos(position);
            break;
    }
}

```

```

                                case 4:
                                    break;
                                }
                            }while(ch!=4);
                        case 4:
                            traverse();
                        case 5:
                            exit(0);
                    }
                }while(choice!=5);
                return 0;
            }

```

Passing and returning structures to and from functions

A function can accept a structure as a parameter, and a function can return a structure. For large structures, programmers usually pass structure pointers or reference variables and let the calling and called functions share copies of the structure. This practice is more efficient as it reduces the overhead of copying large memory segments. It is also somewhat safer. Arguments are passed on the stack, which can become exhausted if a program passes many large objects, particularly to recursive functions.

```

#include<stdio.h>
typedef struct
{
    int month, day, year;
}Date;
//function prototypes
Date GetToday(void);
void PrintDate(Date);
//The main function
int main()
{
    Date dt = GetToday();
    PrintDate(dt);
    return 0;
}
//Function that returns a structure.
Date GetToday(void)
{
    Date dt;
    printf(" Enter date (mm dd yy) : ");
    scanf("%d, %d, %d", dt.month, dt.day, dt.year);
    return dt;
}

```

```

    }

//Function that returns a structure parameter
void PrintDate(Date dt)
{
    printf("%d / %d / %d" dt.month, dt.day, dt.year);
}

```

Benefits of Structure – At a glance

- The value of a structure variable can be assigned to another structure variable of the same type using the assignment operator, which avoids piecemeal copying.

Example:

```

struct course
{
    char name;
    int duration;
    float fees;
};

struct course t1={"JBC",144,4500.00};
struct course t2,t3;

/*Piece-meal copying*/

strcpy(e2.name,e1.name);
e2.age=e1.age;
e2.salary=e1.salary;

/*copying all elements at one go*/
e3=e2;
printf("\n%s %d %f",e1.name,e1.age,e1.salary);
printf("\n%s %d %f",e2.name,e2.age,e2.salary);
printf("\n%s %d %f",e3.name,e3.age,e3.salary);
}

```

SAMPLE OUTPUT

JBC144 4500.00

JBC144 4500.00

JBC144 4500.00

- One structure can be nested within another structure. Using this facility complex data types can be created.

```

struct date
{
    int dd;
    int mm;
    int yy;
}

```

```

};
struct course
{
    char name;
    struct date start_date;
    int duration;
    float fees;
};

```

- Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure at one go.

```

main()
{
    struct course
    {
        char name;
        int duration;
        float fees;
    };
    struct course t1={"JBC",144,4500.00};
    /*passing individual elements*/
    display(t1.name,t1.duration,t1.fees);
    /*passing the structure variable at once*/
    display(t1);
}

display(char *n, int d, float f)
{
    printf("%s %d %f",n,d,f);
}

display(struct course t)
{
    printf("%s %d %f",t.name, t.duration, t.fees);
}

```

- We can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct – known as structure pointers.

```

struct course
{
    char name;
    int duration;
    float fees;
};

struct course t1={"JBC",144,4500.00};
struct course *t2;

```

```

t2=&t1;
printf("%s %d %f",t2->name, t2->duration, t2->fees);

```

Some of the critical uses of structures are listed below.

- Changing the size of the cursor
- Clearing the contents of the screen
- Placing the cursor at appropriate position on the screen
- Checking memory size of a computer
- Receiving a key from the key board
- Formatting a floppy
- Sending output to the printer
- Drawing any graphics shape on the screen

Examples of the above applications are beyond the scope of the book.

The Union Data Type

A union looks just like a structure except that it has the union keyword instead of struct. The difference between union and structure is that a structure defines an aggregate of adjacent data members, and a union defines a memory address, shared by all its data members. A union can contain only one value at a time, the value of the type of one of its members. All its members occupy the same memory location. The size of union is the size of the widest member.

```

#include<stdio.h>
union holder
{
    char holdchar;
    short int holdint;
    long int holdlong;
    float holdfloat;
};
void DisplayHolder(holder, char *);
int main()
{
    holder hld;
    //Assign to the first member
    hld.holdchar='x';
    DisplayHolder(hld, "char");
    //Assign to the second member
    hld.holdint=12345;
    DisplayHolder(hld, "int");
    //Assign to the third member

```

```

        hld.holdlong=7654321;
        DisplayHolder(hld, "long");
        //Assign to the fourth member;
        hld.holdfloat=1.23;
        DisplayHolder(hld, "float");
        return 0;
    }
void DisplayHolder(Holder hld, char* tag)
{
    printf("---Initialized %s ----", tag);
    printf("holdchar   %c",hld.holdchar);
    printf("holdint    %d",hld.holdint);
    printf("holdlong   %ld",hld.holdlong);
    printf("holdfloat  %f",hld.holdfloat);
}

```

Initializing a Union

Only the first of a union's variables can be initialized. Braces enclose the initialize, and there is only one data value, whose type must be compatible with the first member of the union.

```

//Initialize the union variable
Holder hld = {'x'};

```

Typedef

The typedef technique helps to clarify the source code of the C program in some situations. Its purpose is to redefine the name of an existing variable type.

Example 1: Suppose we are using the employee structure where basic salary, da, ta, tax, pf, bonus, gross, net etc. Now they all are related to salary.

Thus to clarify the codes we can typedef in the flowing way.

```

typedef float salary
salary basic, salary da, salary tax, salary gross etc.

```

The datatype with many qualifiers are also sometimes cumbersome to write. In this case, we can use typedef.

Example 2:

The variable of unsigned long int type can be typedef as

```

typedef unsigned long int TWOWORDS;
TWOWORDS var1,var2;

```


Example 3:

The structure declaration can be made more handy to use with typedef

```
struct course
{
    char name;
    int duration;
    float fees;
};
typedef struct course CRS;
CRS c1,c2;
```

By reducing the length and complexity of data types, typedef can help to clarify source listing, save time and energy spent in understanding a program.

Enumerated Data type

The enumerated data type gives the programmer an opportunity to define his own data type and also the values that a data type can take. This helps in making a program listing more readable, which can be an advantage when a program gets complicated or when more than one programmer would be working on it.

Example:

```
enum course
{
    JBC,
    ORACLE,
    ANDROID,
    PHP,
    PERL
};
enum course c;
```

Use

```
enum course
{
    JBC,
    ORACLE,
    ANDROID,
    PHP,
    PERL
};

struct student
```

```

{
    int rollno;
    char name[10];
    char sex;
    enum course c;
    float fees;
    int marks;
};

struct student s;
s.rollno=1;
strcpy(s.name,"SUMIT");
s.sex='M';
s.c=JBC;
s.fees=4500.00;
s.marks=89;

```

In the above program we can't assign a course name to the course variable that is not in the list, i.e., we can't take any course other than JBC,ORACLE,ANDROID,PHP,PERL.

Internally compiler treats the enumerators as integers. Each value on the list of permission values corresponds to an integer, starting with 0. Thus JBC is stored as 0, ORACLE as 1, ANDROID as 2, PHP as 3 and PERL as 4.

MCQ

- 1) Structure is a
 - a) scalar data type
 - b) derived data type
 - c) both options a and b
 - d) primitive data type
- 2) Structure is a data type in which
 - a) each element must have the same data type.
 - b) each element must have pointer type only
 - c) each element may have different data type.
 - d) no element is defined.
- 3) provides a facility for user defined data type using
 - a) pointer
 - b) function
 - c) structure
 - d) array
- 4) The keyword used to represent a structure data type is
 - a) structure
 - b) struct
 - c) struc
 - d) structr
- 5) Structure declaration
 - a) describes the prototype
 - b) creates structure variable
 - c) defines the structure function
 - d) is not necessary
- 6) In a structure definition
 - a) initialization of structure members are possible
 - b) initialization of array of structures are possible
 - c) both options a and b
 - d) initialization of array of structures are not possible
- 7) The operator exclusively used with pointer to structure is
 - a) .
 - b) ->
 - c) []
 - d) *
- 8) If one or more members of a structure are pointer to the same structure, the structure is known as
 - a) nested structure
 - b) invalid structure
 - c) self-referential structure
 - d) structured structure
- 9) If one or more members of a structure are other structures, the structure is known as
 - a) nested structure
 - b) invalid structure
 - c) self-referential structure
 - d) unstructured structure
- 10) What type of structure is created by the following definition?

```
struct first {... ; struct second *s;};  
struct second {...; struct first *f ;};
```

 - a) Nested structure
 - b) Self-referential structure
 - c) Invalid structure
 - d) Structured structure
- 11) The changes made in the members of a structure are not available in the calling function if
 - a) pointer to structure is passed as argument
 - b) the members other than pointer type are passed as arguments
 - c) structure variable is passed as argument
 - d) both options b and c
- 12) The changes made in the members of a structure are available in the calling function if
 - a) pointer to structure is passed as argument
 - b) structure variable is passed
 - c) the members other than pointer type are passed as arguments
 - d) both options a and c
- 13) Identify the wrong statement.
 - a) Structure variable can be passed as argument.
 - b) Pointer to structure can be passed as argument.
 - c) Member variable of a structure can be passed as argument
 - d) None of the above.
- 14) Union is
 - a) A special type of structure
 - b) a pointer data type
 - c) a function data type
 - d) not a data type
- 15) The restriction with union is
 - a) The last member can only be initialized.
 - b) The first member can only be initialized.
 - c) Any member can be initialized .
 - d) Union cannot be initialized.
- 16) Union differs from structure in the following way.
 - a) All members are used at a time.
 - b) Only one member can be used at a time.
 - c) Union cannot have more members.

- d) Union initializes all members as structure.
- 17) Identify the correct statement.
- Unions can be members of structures.
 - Structures can be members of unions.
 - Both options a and b.
 - Union can contain bit field.
- 18))What is not possible with union?
- Array of union
 - Pointer to union
 - Self-referential union
 - None of the above
- 19) Structure is used to implement the data structure.
- stack
 - queue
 - tree
 - all the above
- 20) The nodes in a linked list are implemented using
- self-referential structure
 - nested structure
 - array of structure
 - ordinary structure
- 21) Identify the wrong statement.
- An array is a collection of data items of same data type.
 - An array declaration reserves memory space and structure declaration does not reserve memory space.
 - Array uses the key word array in its declaration.
 - A structure is a collection of data items of different data types.
- 22) Identify the wrong statement.
- An array can have bit fields.
 - A structure may contain bit fields.
 - A structure has declaration and definition.
 - A structure variable can be initialized.
- 23) A bit field is
- a pointer variable in a structure.
 - one bit or a set of adjacent bits within a word.
 - a pointer variable in a union.
 - not used in C.
- 24) A bit field can be of
- int
 - float
 - double
 - all the above
- 25) Identify the wrong statement(s).
- Bit fields have addresses
 - informati
- Bit fields can be read using scanf()
 - Bit fields can be accessed using pointer
 - all the above
- 26) Identify the correct statement(s).
- Bit fields are not arrays.
 - Bit fields can not hold the values beyond their limits.
 - Bit fields may be unnamed also.
 - All the above.
- 27) About structures which of the following is true.
- Structure members are aligned in memory depending on their data type.
 - The size of a structure may not be equal to the sum of the sizes of its members.
- only option 1
 - only option 2
 - both options 1 and 2
 - neither option 1 nor 2
- 28) struct date
- ```

{
 int day;
 int month;
 int year;
};
main()
{
 struct date *d;

 ...
 ++d->day;
 /* statementN */
}

```
- Then the statement statementN
- increments the pointer to point the month
  - increment the value of day
  - increment d by sizeof(struct date)
  - none
- 29) Which is the valid declaration?
- #typedef struct { int i;}in;
  - typedef struct in {int i};
  - #typedef struct int {int i};
  - typedef struct {int i;} in;
- 30) The following statement is  
 "The size of a struct is always equal to the sum of the sizes of its members."
- Valid
  - Invalid
  - Can't say
  - Insufficient

## Practice

1. Define a structure as student. The member of this structure are roll\_no, name[20],branch[15], marks. Take the data of 10 students and find out the average marks scored by the whole batch.
2. Write a program, which processes date structure. For example, in the above mentioned student structure add two elements – exam\_start\_dt and exam\_end\_dt. Accept the start date for the exam in exam\_st\_dt and calculate the end day for the examination if the examination is conducted for 20 days.
3. Write a program to process complex numbers. The program should be able to perform addition, subtraction, multiplication, and division of complex numbers. Print the results.
4. Write a program in structure to input the details of 10 employees .The data members are name, age, and salary. Sort the records according to name, then by age and then by salary.

-----

# Chapter VIII : Input and Output in C

---

When Denis Ritchie developed C, he wanted to keep C compact. So, C doesn't have any provision for receiving data from any of the input devices like keyboard, nor sending the data to output device say VDU. Thus automatically the question arises is how we are able to accept and display data using `printf()` and `scanf()` then !!! We will be discussing how C offers I/O operations in this chapter.

## Types of I/O

Each operating system has its own facility for inputting and outputting data from and to the files or devices. So, the only thing the developer of the C compiler has to do is to write a few small programs that would link the C compiler for particular Operating System's I/O facilities. They write certain standard I/O functions and put them in the libraries. Though these functions are not part of C's formal definition; they have become a standard feature of C language.

There are numerous library functions available for I/O. They can be categorized into three broad categories.

- Console I/O functions – functions to receive input from keyboard and write output to VDU.
- Disk I/O functions – functions to perform I/O operations on a floppy disk or hard disk.
- Port I/O functions – functions to perform I/O operation on various ports.

## Console I/O Functions

Console I/O functions can be further sub-divided into two categories –

### ➤ *Formatted console I/O function* –

The formatted function allows the input read from keyboard and or the output displayed in VDU to be formatted as per the requirement. For example suppose the bill number and bill amount has to be displayed on the screen, then the details like where this output will appear on the screen, how many spaces would be present between two values, the number of places after the decimal point etc. can be controlled using formatted functions.

#### **Example:**

```
main()
{
 int billno=18;
 float billamt=2118.75;
 printf("Bill number\t=\t%d \n Bill Amount\t=\t%f", billno,
billamt);
}
```

The output will be

Bill Number : 18

Bill Amount : 2118.750000

Now the question arises how does printf() interpret the contents of the format string. For this it examines the string from left to right. So long as it doesn't come across either a % or a \ it continues to dump the characters that it encounters, on to the string. In the above example, Bill number is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up first variable in the list of variables and prints its value in the specified format. The example mentioned above the moment %d is met, the variable billno is picked up and its value is printed. Similarly the moment as escape sequence is met it takes its appropriate action. In the example when it comes across \n the cursor is placed at the beginning of the next line.

The lists of conversion characters are as follows.

| Data Type |                      | Conversion Character |
|-----------|----------------------|----------------------|
| Integer   | short signed         | %d or %i             |
|           | short unsigned       | %u                   |
|           | long signed          | %ld                  |
|           | long unsigned        | %lu                  |
|           | unsigned hexadecimal | %x                   |
|           | unsigned octal       | %o                   |
| Real      | float                | %f                   |
|           | double               | %lf                  |
| Character | signed character     | %c                   |
|           | unsigned character   | %c                   |
| String    |                      | %s                   |

The following specifiers can be used in conversion specification too.

| Specifier | Descriptor                                                                                                           |
|-----------|----------------------------------------------------------------------------------------------------------------------|
| dd        | Digits specifying field width                                                                                        |
| .         | Decimal points separating field width from precision (precision stands for number of places after the decimal point) |
| dd        | Digits specifying precision                                                                                          |
| -         | Minus sign for left justifying the output in the prescribed field width.                                             |

Though we have already discussed the use of conversion characters, we are discussing it again through an example for you better understanding.

```
main()
{
 int marks=89;
 float percentage=85.6785;
 char string[]="microsoft";
 printf("\n marks : %d:",marks);
 printf("\n marks : %2d:",marks);
 printf("\n marks : %4d:",marks);
 printf("\n marks : %6d:",marks);
 printf("\n marks : %-6d:",marks);
 printf("\n precentage : %8.4f",marks);
 printf("\n precentage : %8.3f",marks);
 printf("\n precentage : %8.2f",marks);
 printf("\n precentage : %8.1f",marks);
 printf("\n%s",string);
 printf("\n%15.5s",string);
 printf("\n%.5s",string);
}
```

#### Output:

|        |                                          |
|--------|------------------------------------------|
| Column | 0123456789012345678901234567890123456789 |
|        | marks : 89:                              |
|        | marks : 89:                              |
|        | marks : 89:                              |
|        | marks : 89:                              |
|        | marks : 89 :                             |
|        | percentage : 85.6785                     |
|        | percentage : 85.679                      |
|        | percentage : 85.68                       |
|        | percentage : 85.7                        |
|        | microsoft                                |
|        | micro                                    |
|        | micro                                    |



### sprintf() and sscanf() functions:

The sprintf() function works similar to printf() function except for one small difference. Instead of sending the output to screen as printf() function does, the function writes the output to an array of characters.

#### **Example:**

```
main()
{
 int i=15;
 char ch='S';
 float f=21.18
 char str[10];
 printf("\n%d %c %f",i,ch,a);
 sprintf(str,"%d %c %f",i,ch,a);
 printf("\n%s",str);
}
```

#### **output:**

15 S 21.180000

15 S 21.180000

### **➤ Unformatted console I/O function**

There are several standard library functions available under this category – those, which can deal with a single character and those, which can deal with a string of characters.

Using scanf() function, the user needs to press the enter key after submitting the value. Now in certain circumstances, it is required that the function should read a single character the instance it is typed without waiting for the enter key to be hit. getch() and getche() are two functions, which serve this purpose. These functions return character that has been most recently typed. The 'e' in getche() function means it echoes the character you have typed on the screen, but unfortunately requires enter key to be pressed following the character that is been typed by the user. The difference between getchar() and fgetchar() is that the former is a macro whereas the latter is a function.

```
main()
{
 char ch;
 printf("\nPress any key to continue");
 getch(); /*will not echo the character */
 printf("\nType any character");
 ch=getche(); /*will echo the character typed */
 printf("\nType any character");
}
```

```

 getchar(); /*will echo character, must be follow by the enter key*/
 printf("\nContinue Y/N");
 fgetchar(); /*will wcho character, must be followed by enter key*/
 }

```

**When we run the program:**

Press any key to continue

Type any character A

Type any character S

Continue Y/N Y

putch() and putchar() works similar to getch() and getchar() for writing a character. As far as the working of **putch()**, **putchar()** and **fputchar()** is concerned it's exactly same.

**Example:**

```

main()
{
 char ch='A';
 putch(ch);
 putchar(ch);
 fputchar(ch);
 putch('A');
 putchar('A');
 fputchar('A');
}

```

**When we run the program:**

AAAZZZ

**Limitation:**

putch(), putchar() and fputchar() can output only one character at a time.

gets() and puts() functions:

Till now, when you are working with strings you must have faced a common problem as stated below.

```

main()
{
 char name[20];
 printf("Enter your name ");
 scanf("%s",name);
 printf("\nYour name is %s",name);}

```

**output:**

Enter your name Anukampa Mitra

Your name is Anukampa

When we are trying to input a multi-word name, it is only able to store the first word. The nest words are never getting stored in the array name[], because the moment the blank was typed after the first word scanf() assumes that the name that is been entered is finished. Thus, we were not being able to enter multiple words.

As a solution to this problem, gets() function can be used. gets() receives a string from the keyboard, spaces and tabs are perfectly accepted as part of input string. gets() gets a newline (\n) terminated string of characters from the keyboard and replaces the \n with a \0.

Puts() function works just opposite as gets() function works.

```
main()
{
 char name[20];
 puts("Enter Name");
 gets(name);
 puts("Wish you a very good morning");
 puts(name);
}
```

**output:**

Enter name

Anukampa Mitra

Wish you a very good morning

Anukampa Mitra

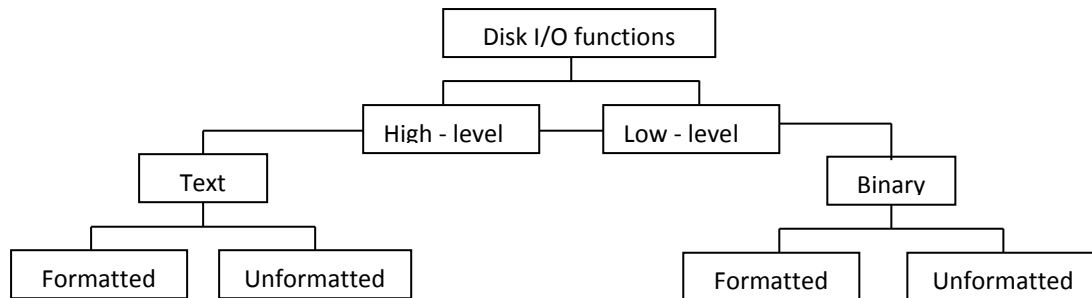
The reason why we put "Wish you a very good morning" and name in two different puts() functions is that, unlike printf() function, puts() function can print only one string at a time. Similarly unlike scanf() function, gets() function can accept only one string at a time.

## Disk I/O Functions

Disk I/O functions basically deal with the input and output from files i.e., reading from and writing to files. These functions can be broadly divided into two categories:

- High Level I/O functions (also called standard or stream I/O functions).
- Low-level file I/O functions (also called system I/O functions).

High-level I/O functions are easier to use than low-level disk I/O functions. Higher-level disk I/O functions do their own buffer management, whereas in lower-level disk functions, buffer management has to be done explicitly by the user. Though it is difficult to program low-level I/O functions, they are more efficient both in terms of operation and the amount of memory used by the program.



Higher-level functions are further categorized into two types – Text and Binary. This classification depends upon which mode the file is opened for input or output. The following things are determined depending upon the mode in which the file is opened:

- How newline (\n) characters are stored.
- How end of file is indicated.
- How numbers are stored in the file.

Before going to these details, first let's write a simple program which performs file I/O in high-level, unformatted, text mode.

The following program opens a file in read mode, and counts how many characters, spaces, tabs and newline are present in the file.

#### Example – 1:

```

/*Count characters, spaces, tabs, and newlines in a file*/
#include<stdio.h>
main()
{
 FILE *fp;
 char ch;
 int nol=0, notb=0, nos=0, noc=0;

 fp=fopen("application.txt","r");
 while(1)
 {
 ch=fgetc(fp);
 if(ch==EOF)

```

```

 break;
 noc++;
 if(ch==' ')
 nob++;
 if(ch=='\n')
 nol++;
 if(ch=='\t')
 notb++;
}

fclose(fp);
printf("\nNumber of characters = %d",noc);
printf("\nNumber of blanks = %d",nob);
printf("\nNumber of tabs = %d",notb);
printf("\nNumber of lines = %d",nol);
}

```

In the above program, we have opened a file, read it and closed it. Lets discuss each of these activities.

### Opening a file

Before doing any kind of manipulations, the file has to be opened. Opening a file establishes the link between the program and operating system. The link between the OS and program, is a structure called FILE which has been defined in the header file "stdio.h". Thus, this header file needs to be included whenever any high-level I/O operation is done. When we request the S to open a file, it returns a pointer to the structure FILE.

Each file we open has its own file structure, which contains information about the file being used, such as its current size, location in memory etc. It contains a character pointer, which points to the character that is about to get read.

FILE \*fp;

fp is a pointer variable, which contains address of the structure FILE which has been defined in the header file "stdio.h".

For opening a file we use fopen(), which opens the file and the file's contents are brought into memory (partly or wholly depending upon the file size).

fp=fopen("application.txt","r");

In the above statement, the file application.txt is opened in '**read**' mode, which tells the C compiler that the file has been opened to read the contents of the file. As we are putting **r** within double quotes, it signifies that r is a string not a character. When we open a file in "r" mode, fopen dos the following tasks.

- Firstly it searches on the disk file to be opened.

- If the file is present, it loads the file from the disk into the memory. If the file is very big, then file is loaded part by part.

If the file is not present, then `fopen()` function returns a NULL value.

- It sets up a character pointer (which is part of the FILE structure) which points to the first character of the chunk of memory where the file has been loaded.

## Reading From a File

After the file is loaded to the memory, the file pointer points to the first character of the file. To read the file's contents from memory `fgetc()` function is used.

```
ch= fgetc (fp) ;
```

`fgetc()` reads the character from current pointer position, advances the pointer position so that it read, which gets collected in the variable `ch`. Once the file is opened, its name is no more used, the file is accessed through file pointer.

Thus for reading the file we use the following code:

```
while (1)
{
 ch=fgetc (fp) ;
 if (ch==EOF)
 break;
}
```

In the above program we are using an infinite while loop. The moment we reach the end of file, the loop is broken. End-of-file is represented by a special character whose ASCII value is 26. This character is inserted beyond the last character in the file, when it is created. This character can also be generated from the keyboard by typing `Ctrl+z`.

While reading the file, when `fgetc()` encounters this special character, instead of returning the character that is has read, it returns the macro `EOF`, which as been defined in the file `stdio.h`.

## Troubles In Opening a file

When we are trying to open a file, it may not open due to the following reasons:

- The file may not be present in the disk at all. Thus obviously we can't open a file, which is not present in the disk.
- The disk may be write-protected.
- Space may be insufficient to open a file.

In any of the above mentioned cases `fopen()` returns a NULL value.

```
#include<stdio.h>
main()
{
 FILE *fp;
 char ch;
 int nol=0, notb=0, nos=0, noc=0;
 fp=fopen("application.txt","r");
 if(fp==NULL)
 {
 printf("Unable to open the file");
 exit();
 }
}
```

## Closing The File

When the job is finished, the file is needed to be closed. This is done using the function `fclose()` through the statement.

```
fclose (fp);
```

This deactivates the file and hence it can no longer be accessed using `fgetc()`.

## File Opening Modes

There are different modes in which a file can be opened. Here we are listing all possible modes in which a file can be opened.

**“r”** Searches file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist it returns NULL.

**Operations Possible:** Reading from file

**“w”** Searches File. If the file exists its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open a file.

**Operations Possible:** Writing to file

**“a”** Searches file. If the file exists, loads it into memory and sets up a pointer, which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open a file.

**Operations Possible:** Appending new contents at the end of the file.

**“r+”** Searches file. If the file exists, loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist it returns NULL.

**Operations Possible:** Reading existing contents, writing new contents, modifying existing contents of the file.

**“w+”** Searches file. If the file exists, its contents are destroyed. If the file doesn't exist a new file is created. Returns NULL if unable to open a file.

**Operations Possible:** Writing new contents, reading it back and modifying existing contents of a file.

**“a+”** Searches file. If the file exists, loads it into memory and sets up a pointer, which points to the last character in it. If the file doesn't exist a new file is created. Returns NULL if unable to open a file.

**Operations Possible:** Reading existing contents, appending new contents at the end-of-the file. Can't modify existing contents.

### **A sample program that will copy a file**

```
#include<stdio.h>
main()
{
 FILE *src, *dst;
 /* Two pointer to the source and destination file
 respectively*/
 char ch;
 /* Character variable that will hold each character that is read
 from the file*/
 src=fopen("anu.dat","r");
 /*opening anu.dat - the source file in the read mode */
 if(src==NULL)
 {
 printf("Can't open the target file");
 exit();
 }
 dst=fopen("anu_dst.dat","w");
 /*opening anu.dat - the source file in the write mode */
 if(dst==NULL)
 {
 printf("Can't open the destination file");
 fclose(src);
 exit();
 }
 while(1)
 {
 ch=fgetc(src);
 if(ch==EOF) /*checking for end-of-file*/
 {
 break;
 }
 }
```



```

 else
 {
 fputc(ch,dst);
 }
 }
 fclose(src);
 fclose(dst);
}

```

### Writing to a file

For writing to a file, we can use the function `fputc()` function. This is similar to `putc()` function. `putc()` function always writes to the VDU, where as `fputc()` function writes to the file that is opened with the help of `fopen()` function for writing purpose.

### Buffer

Buffer is an area in the memory, which is used to temporarily store the data. While reading a file, if each time the character would have to be read from the disk then it will be highly ineffective, as each time the disk has to rotate and the disk drive position the read/write head correctly in the place where the data will be read from. Thus, for each character read/ write it has to position the head every time, which will take a lot of time. Therefore, a temporary memory area is used where the characters are actually stored, and from there the read/write operation takes place. When the buffer is full, the content are written to the disk at once.

Now if we close a file, then no matter whether the buffer is full or not, the content of the buffer is forcefully written onto the file. And `fclose()` function also frees the link used by the particular file, and the associated buffers, so that these are available for other files.

## Using Command line Arguments for read/write to files

For simulation of DOS commands like copy, we can take the help of command line arguments. In this way there will not be any need for recompiling the program each time we need to copy the files. Just running the exe files with source and destination files will serve the purpose.

We can write it in the following form:

```
C:\>mycopy anu.txt anu1.txt
```

Where `anu.txt` is the source file name and `anu1.txt` is the destination file name.

### Example:

```

#include<stdio.h>
main(int argc, char *argv[])
{
 FILE *src, *dst;

```

```

 /* Two pointer to the source and destination file
 respectively*/
 char ch;
 /* Character variable that will hold each character that is read
 from the file*/
 if(argc!=3)
 {
 printf("\nInvalid number of arguments");
 exit();
 }
 src=fopen(argv[1],"r");
 /*opening source file, the source file in the read mode */
 if(src==NULL)
 {
 printf("Can't open the target file");
 exit();
 }
 dst=fopen(argv[2],"w");
 /*opening destination file - the source file in the write mode */
 if(dst==NULL)
 {
 printf("Can't open the destination file");
 fclose(src);
 exit();
 }
 while((ch=fgetc(src))!=EOF)
 {
 fputc(ch,dst);
 }
 fclose(src);
 fclose(dst);
 }

```

In the above program, argc contains 3 as exe file name, source and destination file name.

argv[0] contains the base address of mycopy

argv[1] contains the base address of "anu.txt"

argv[2] contains the base address of "anu1.txt"

The above program is better than the former program in three areas.

- There is no need to recompile the program every time we want to use this utility. It can be executed at command prompt.

- Once the executable file is created nobody with malicious intention can tamper with the source code.
- We are able to pass source file name and target file name to main(), and utilise them in main().

## String (line) I/O in files

Reading or writing strings of characters from and to files is as easy as reading and writing individual characters. We can use fputs() function for writing into a file.

### Sample Program

```
/*Program that receives a string from keyboard and writes it to the
file*/
#include<stdio.h>
main()
{
 FILE *fp;
 char str[80];
 fp=fopen("APPLICATION.TXT", "w");
 if (fp==NULL)
 {
 puts("Cannot open file");
 exit();
 }
 printf("\n Enter a few lines of text: \n");
 while(strlen(gets(str))>0)
 {
 fputs(s,fp);
 fputs("\n");
 }
 fclose(fp);
}
```

While entering each string each line should be terminated by hitting enter. To terminate the execution of the program, hit enter at the beginning of a line. This creates a string of zero length, which the program recognizes as the signal to close the file and exit.

Similarly for reading from a file we can use fgets() function which takes three arguments. The first is the address where the string is stored, second is the maximum length of the string which prevents from reading in too long a string, and the third argument is pointer to the structure FILE.

### Sample Program:

```
/*Reading strings from a file and display it on the screen*/
```

```

#include<stdio.h>
void main()
{
 FILE *fp;
 char str[80];

 fp=fopen("APPLICATION.TXT");
 if (fp==NULL)
 {
 printf("Can't open file");
 exit();
 }

 while(fgets(str,79,fp)!=NULL)
 {
 printf("%s",str);
 }

 fclose(fp);
}

```

### **All functions together**

| Category                                       | Functions                                                                                                         |
|------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| Higher Level, Text, Unformatted, Character I/O | getc(), putc(), fgetc(), fputc()<br><br>getc(), putc() are macros, gets() and puts() are their function versions. |
| Higher level, text, unformatted, string I/O    | fgets(), fputs()                                                                                                  |
| Higher level, text, unformatted, int I/O       | No standard library function                                                                                      |
| Higher level, text, unformatted, float I/O     | No standard library function                                                                                      |

## **Formatted Disk I/O functions**

So far we have discussed only about reading and writing character string. For formatted reading/writing characters, strings, integers, floats, there exists two functions, **fscanf()** and **fprintf()**. For example:

```

#include<stdio.h>
main()
{
 FILE *fp;
 char another='y';
}

```

```

char item_name[20];
int code;
float price;
fp=fopen("ITEM.DAT","w");
if(fp==NULL)
{
 printf("Can't open file");
 exit();
}

while(another=='y')
{
 printf("Enter Item name, code and price \n");
 scanf("%s %d %f", item_name, &code, &price);
 fprintf(fp, "%s %d %f", item_name, code, price);
 printf("Another item ???");
 fflush(stdin);
 another=getche();
}

fclose(fp);
}

```

**Now for reading the file:**

```

#include<stdio.h>
main()
{
 FILE *fp;
 char another='y';
 char item_name[20];
 int code;
 float price;
 fp=fopen("ITEM.DAT", "r");
 if(fp==NULL)
 {
 printf("Can't open file");
 exit();
 }
 while(fscanf(fp,"%s %d %f", item_name, &code, &price) != EOF)
 printf("\n%s %d %f", item_name, code, price)
 fclose(fp);
}

```

## Text mode versus Binary mode

The high-level disk I/O functions can be categorized as text and binary. This classification is dependent upon in which mode the file is opened. In the following three areas text and binary modes behave differently.

- Handling of newlines
- Representation of end-of-file
- Storage of numbers

### Text versus Binary Mode: Newline

As we have discussed earlier, in text mode, a newline character is converted into carriage return – linefeed combination before being written to the disk. Similarly, the carriage return – linefeed combination on the disk is converted back into newline when C program reads the file. However if a file is opened in binary mode, these conversions will not take place.

In text mode the end of line is counted as two characters whereas in binary mode it is counted as one.

To open a file in binary read mode we can give the mode as “**rb**”.

### Text versus Binary Mode: End of file

In text mode for representing end-of-file a special character, whose ASCII value is 26 is used which is inserted after the last character in the file. If this special character is detected at any point of time in the file, the read function would return the EOF signal to the program.

But, in binary mode there is no such character like this is used. The binary mode files keep track of the end-of-file by the number of characters present in the directory entry of the file.

As these two ways are not compatible with each other, a file that is written in text mode has to be read in text mode and a file that is written in binary mode has to be written in binary mode.

### Text versus Binary Mode: Storage of Numbers

In text mode the numbers are stored in the form of string of characters. For example: 1234 is stored as the string “1234” i.e., it takes 4 bytes of storage space whereas in the binary mode it is represented as a number and it occupies 2 bytes of space. 3214.55398 occupies 10 bytes in text mode whereas 4 bytes in binary mode.

## Record I/O in Files

Suppose we want to write records to a file, i.e., a combination of dissimilar data types then we can do it in the following way:

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

```

FILE *fp;
char another='y';
struct item
{
 char item_name[20];
 int code;
 float price;
};
struct item t;
fp=fopen("ITEM.DAT","w");
if(fp==NULL)
{
 printf("Can't open file");
 exit();
}
while(another=='y')
{
 printf("Enter Item name, code and price \n");
 scanf("%s %d %f", t.item_name, &t.code, &t.price);
 fprintf(fp, "%s %d %f", t.item_name, t.code, t.price);
 printf("Another item ???");
 fflush(stdin);
 another=getche();
}
fclose(fp);
}

```

Now the above program has two disadvantages:

- The numbers (code and price) will occupy more number of bytes, since the file has been opened in text mode as in text mode the numbers are stored in character string.
- If the number of fields in the structure increase, then reading/writing using fscanf()/fprintf() becomes clumsy.

Thus, for efficiently handling the record we can use fread() and fwrite() functions.

The same program we can rewrite in the following way:

```

#include<stdio.h>
main()
{
 FILE *fp;

```

```

char another='y';
struct item
{
 char item_name[20];
 int code;
 float price;
};
struct item t;
fp=fopen("ITEM.DAT","wb");
if(fp==NULL)
{
 printf("Can't open file");
 exit();
}
while(another=='y')
{
 printf("Enter Item name, code and price \n");
 scanf("%s %d %f", t.item_name, &t.code, &t.price);
 fwrite(&t,sizeof(t),1,fp);
 printf("Another item ???");
 fflush(stdin);
 another=getche();
}
fclose(fp);
}

```

The change you can notice in the program is

```
fwrite(&t, sizeof(t), 1, fp);
```

Here the first argument is the address of the structure to be written to the disk. Second argument is the size of structure in bytes. Instead of counting the size, we can use the sizeof() operator. The third argument is the number of such structures we want to write at one time and the last argument is the pointer to the file.

Similarly we can read the data in binary mode too.

```

#include<stdio.h>
main()
{
 FILE *fp;
 char another='y';
 struct item
 {

```



```

 char item_name[20];
 int code;
 float price;
 };
 struct item t;
 fp=fopen("ITEM.DAT","rb");

if (fp==NULL)
 {
 printf("Can't open file");
 exit();
 }
 while(fread(&e,sizeof(e),1,fp)==1)
 {
 printf("\n%s %d %f", t.item_name, t.code, t.price);
 }
 fclose(fp);
}

```

Structure of fread() is similar to fwrite(). The fread() function returns the number of records read. In the above case as we are reading one record at a time it will return 1. When we reach the end of file, since fread() cannot read anything, it returns a 0.

### **A sample program on file handling**

```

#include<stdio.h>
#include<stdlib.h>
int rn=0;
main()
{
 FILE *fp,*ft;
 int wish,rec;
 char another,choice;
 struct item
 {
 int rec_no;
 char tool_name[20];
 int quantity;
 int cost;
 };
 struct item e;
 long int recsize;
 fp=fopen("HARDWARE.DAT","rb+");

```

```

if (fp==NULL)
{
 fp=fopen("HARDWARE.DAT","wb+");
 if (fp==NULL)
 {
 puts("\nCan't open file");
 exit(0);
 }
}
recsize=sizeof(e);
while(1)
{
 clrscr();
 gotoxy(30,10);
 printf("1. Add New Record");
 gotoxy(30,12);
 printf("2. List Records");
 gotoxy(30,14);
 printf("3. Update Records");
 gotoxy(30,16);
 printf("4. Delete Records");
 gotoxy(30,18);
 printf("0. Exit");
 gotoxy(30,22);
 printf("Enter your choice");
 fflush(stdin);
 choice=getche();

 switch(choice)
 {
 case '1':
 fseek(fp,0,SEEK_END);
 another='y';
 while(another=='y')
 {
 clrscr();
 rn++;
 e.rec_no=rn;
 printf("\nEnter the tool name, quantity, cost");
 scanf("%s %d %d",e.tool_name,e.quantity,e.cost);

```



```

scanf("%s",e.tool_name);
printf("\nQuantity\t:\t");
scanf("%d",e.quantity);
printf("\nCost\t:\t");
scanf("%d",e.cost);
fseek(fp,-recsize,SEEK_CUR);
fwrite(&e,recsize,1,fp);
 }
}
}
printf("\n Want to modify another record ?");
fflush(stdin);
another=getche();
}
break;
case '4':
 another='y';
 while(another=='y')
 {
 printf("\nEnter the record number you want to
delete");

 scanf("%d",&rec);
 ft=fopen("TEMP.DAT","wb");
 rewind(fp);
 while(fread(&e,recsize,1,fp))
 {
 if(e.rec_no!=rec)
 fwrite(&e,recsize,1,ft);
 }
 fclose(fp);
 fclose(ft);
 remove("HARDWARE.DAT ");
 rename("TEMP.DAT"," HARDWARE.DAT ");
 fp=fopen("HARDWARE.DAT ","rb+");
 printf("Delete another Record (y/n)");
 fflush(stdin);
 another=getche();
 }
 break;
case '0':
 fclose(fp);

```

```

 exit(0);
 }
}

```

Lets discuss briefly how the above program works!

A pointer **fp** is initiated whenever we open a file. On opening a file a pointer is set up which points to the first record in the file. On using the functions **fread()** and **fwrite()** the pointer moves to the beginning of the next record. On closing a file, the pointer is deactivated. **fread()** function always reads the that record where the pointer is currently placed. Similarly, **fwrite()** function always writes the record where the pointer is currently placed.

The **rewind()** function works just like rewind operation ion tape recorders, it places the pointer to the beginning of the file, irrespective of where it is present currently.

The **fseek()** function moves the pointer from one record to another.

```
fseek (fp,-reccsize,SEEK_CUR)
```

takes the cursor one record before the current record.

The second argument is the offset, which tells the compiler by how many bytes should a pointer, be moved from a particular position. The third argument can be **SEEK\_SET**, **SEEK\_CUR** or **SEEK\_END**, **SEEK\_SET** means move the pointer with reference to the beginning of the file. **SEEK\_CUR** means move the pointer with reference to its current position and **SEEK\_END** means move the pointer from the end of the file.

If we want to know the current position of pointer, then **ftell()** function can be used. It returns the position as a **long int**, which is an offset from the beginning of the file.

## MCQ

- 1) File is a
  - a) a data type
  - b) a region of storage
  - c) both options a and b
  - d) a variable
- 2) The way to access file contents from a program is
  - a) by using library functions
  - b) by using system calls
  - c) both options a and b
  - d) to use a linker
- 3) Stream oriented files are accessed through
  - a) system calls
  - b) library functions
  - c) linker
  - d) loader
- 4) Low level files are accessed through
  - a) system calls
  - b) library functions
  - c) linker
  - d) loader
- 5) C supports
  - a) high level files
  - b) low level files
  - c) both options a and b
  - d) executable files only
- 6) A stream is
  - a) a library function
  - b) a system call
  - c) a source or destination of data that may be associated with a disk or other I/O devices
  - d) a file
- 7) I/O stream can be
  - a) a text stream
  - b) a binary stream
  - c) both options a and b
  - d) an I/O operation
- 8) File opening is
  - a) a default action in file processing
  - b) an action of connecting a program to a file
  - c) not necessary
  - d) not using any library function
- 9) FILE defined in stdio.h is
  - a) a region of storage
  - b) a data type
  - c) not a data type
  - d) a variable
- 10) A file pointer is
  - a) a stream pointer
  - b) a buffer pointer
  - c) a pointer to FILE data type
  - d) all the above
- 11) The default stream pointers available during execution of a program is
  - a) stdin
  - b) stdout
  - c) stderr
  - d) all the above
- 12) The function call fopen(" data", "w+b")
  - a) is invalid.
  - b) returns the file pointer pointing to file named data and opens the file for read and writing using binary stream.
  - c) returns the file pointer pointing to file named data and opens the file for read and writing using text stream
  - d) does not return file pointer
- 13) If fopenQ fails, it returns
  - a) -1
  - b) NULL
  - c) 1
  - d) the file pointer
- 14) The function fclose() is
  - a) used to disconnect a program from file
  - b) used to close a file logically
  - c) both options a and b
  - d) a mandatory function call in file handling
- 15) The action of connecting a program to a file is obtained by using
  - a) connect()
  - b) fopen()
  - c) OPEN()
  - d) file()
- 16) The action of disconnecting a program from a file is obtained by the function
  - a) fclose()
  - b) delete()
  - c) fdconnect()
  - d) clear()
- 17) The value returned by fcloseQ, if an error occurs is
  - a) 0
  - b) 1
  - c) EOF
  - d) -1
- 18) The value returned by fclose() for successful closing of a file is
  - a) 0
  - b) 1
  - c) EOF
  - d) OK
- 19) The function(s) used for reading a character from a file is (are)
  - a) getc()

- b) fgetc()
  - c) both options a and b
  - d) fgetcharQ
- 20) The function(s) used for writing a character to a file is (are)
- a) putc()
  - b) fputc()
  - c) both options a and b
  - d) fputcharQ
- 21) The function(s) used for reading formatted input data from a file is (are)
- a) getchar()
  - b) fscanf()
  - c) scanf()
  - d) gets()
- 22) The function used for random access of a file is
- a) fseek()
  - b) ftell()
  - c) search()
  - d) rewind()
- 23) What is the value of origin used in fseek(fp, position, origin);?
- to represent end of file.
- a) 0
  - b) 1
  - c) 2
  - d) EOF
- 24) What is the value of origin used in fseek(fp, position, origin);?
- to represent the beginning of the file
- a) 0
  - b) 1
  - c) 2
  - d) START
- 25) If an error occurs, the function fseek() returns
- a) non-zero
  - b) zero
  - c) no value
  - d) -1
- 26) The value returned by fseek() on successful action is
- a) non-zero
  - b) zero
  - c) OK
  - d) READY
- 27) The function ftell(fp) returns
- a) the beginning position of the file represented by fp
  - b) the end position of the file represented by fp
  - c) the current position of the file represented by fp
  - d) the middle position of the file represented by fp
- 28) The value returned by successful action of ftell(fp) is
- a) -1
  - b) 0
  - c) long int value representing the current file position
  - d) MAX\_INT
- 29) The value returned by ftell() if an error occurs is
- a) -1
  - b) 0
  - c) Positive value
  - d) MIN\_INT
- 30) The function call f seek ( fp, 0, 0 ); is same as
- a) fp = fopen()
  - b) rewind (fp);
  - c) fclose(fp);
  - d) ftell(f

## **Practice**

1. Write a program that will read a program file as input and output the file properly indented

For example:

### **Source Program:**

```
main()
{
int x,sum=0;
for(x =0;x < 10; x ++)
{
sum+=x;
}
printf("%d",sum);
}
```

### **Output**

```
main()
{
int x, sum=0;
printf("Enter the value for x");
scanf("%d", &x);
for(x =0;x < 10; x ++)
{
sum+=x;
}
printf("%d",sum);
}
```

2. Write a program that will accepts two files as source and destination and copies the source files to the destination file. Simulate the DOS/UNIX copy/cp command. The format is as follows:

```
C:\>COPY source.txt destn.txt
```

```
$ cp soure.txt destn.txt
```

3. Write a program that will print out only those lines from a file, which are containing more than 80 characters. Also print out the line numbers of these lines.
4. Write a program that will concatenate two files: that is, append the contents of one file at the end of another and write the result into a third file. You must be able to execute the command at DOS prompt as follows:

```
C:\>CONCAT source.txt destn.txt
```

5. A file contains a C program, but by mistake it has been typed it out in capitals. As it is very long, we can't retype it. So write a program that will convert the files contents to small letters and save in the same file.

-----



Section - II

Programming

with

C++

# Chapter – IX: Introduction to C++

---

## Introduction to C++ as a programming language:

C++, is nothing but known as “*C with classes*”. It was developed at AT&T Bell Laboratories in the early 1980s by Bjarne Stroustrup. Two languages basically contributed to the design of C++ - C, which provided low-level features and Simula-67, which provided the class concept. C++ thus is an example of a hybrid language.

## C Versus C++:

C++ language adds keywords that are not reserved by the C language. Those keywords legitimately can be used in a C program as identifiers for function and variables. Although C++ is said to include all of C, clearly no C++ compiler can compile such a program.

C programmers can omit function prototypes. C++ programmer cannot.

A C function prototype with no parameters must include the keyword void in the parameter list. A C++ prototype can use an empty parameter list.

Many standard C functions have counterparts in C++, and C++ programmers view them as improvements to the C language. For example:

- The C++ new and delete memory allocation operators replace standard C's malloc and free functions.
- The standard C++ string class replaces the character array processing functions declared in the standard C library's <cstring> header file.
- The C++ iostream class library replaces standard C's stdio function library for console input and output.
- C++'s try/catch/throw exception handling mechanism replaces Standard C's setjmp() and longjmp() functions.

## A brief Description of C++ language:

C++ is a procedural programming language with object-oriented extensions. This means that you can design and code programs as procedural modules, and you can define, instantiate (create from a class) objects. The procedural modules in C are called functions. Object declarations are called classes.

Every C++ program begins with a function called `main()` and terminates when the `main()` returns. The `main()` calls lower level functions. A function starts execution at its first, topmost statement and continues until last, bottommost statement executes or the function executes a return statement from inside the function body. Each function, upon completion, returns to its caller. Execution then continues with the next program statement in the caller function.

In a C++ program, though the program is read from top to bottom, the functions do not have to be coded in that sequence. All declarations of functions and variables must be coded in the program above any statements that reference them i.e., a function must be declared before it is called and a variable must be declared before it is used in the statement. A function can be declared by providing a function prototype that describes its name, return value and parameters to the compiler. A function's prototype must be declared ahead of any calls to the function, the function definition can be put anywhere else in the program. The functions declaration and all calls to it must match the prototype with respect to the return type and the types of its parameters.

➤ ***A parameter is a value that a function declares in its definition. Parameters represent values that are passed into the function when the function is called.***

Functions may contain parameters. The caller of the function passes arguments that the function uses as the values for its parameters. The argument types must match or be compatible – with respect to type conversion – with the types of the parameters declared for the function.

Some functions return a value and other may not. The caller of a function that returns a value can code the function call on the right side of an assignment statement, assigning the returned value to a named data variable. A function call can also be coded as the argument that provides the parameter value to another function call, as an initializer to a local data variable, or as an element in an expression.

Each function consists of one or more block of statements. The blocks are nested (one enclosed within another). Each block can have its own local variables, which remain in scope as long as statements in the block are executing. A C program also can have variables declared outside any function. Those variables are said to have global scope, because that statements in all the functions in the same source file can reference them.

Statements fall into one of the following categories:

- Declarations: Declare variables and functions.
- Definitions: Define instances of variables and functions.
- Procedural Statements: Executable code statements that reside inside a function's definition.

A variable declaration specifies the variable's storage class, data type, type qualifier, level of indirection (if it is a parameter), name, and dimension (if it is an array). A function declaration (more frequently called its prototype) declares the function's return type, name, and the number and types of its arguments.

A variable definition includes the components of a declaration and may include an initialiser if the variable has an initialising value. A definition defines the instance of the variable and reserves memory for it. A function definition contains the function's executable code.

Generally, a variable's declaration and definition are the same statement. A function's declaration (prototype) and definition are usually in different places. If the function's definition appears in the source code file ahead of all calls to the function however, then the function's definition may also serve as its prototype.

Procedural statements are assignments, expressions, or program flow control statements. An expression can stand on its own or be on the right side of an assignment statement. Expressions consist of variables, constants, operators, and function calls. Paradoxically, assignments are themselves expressions.

C++ uses the control structures of structured programming, which include sequence (one statement executing after another), iteration (for and while loops), and selection (if-then-else and switch-case control structures). C++ also permits unstructured programming with goto statement.

Classes are aggregate definitions that consist of data members and member functions. A class encapsulates the implementation and interface of a user-defined data type and constitutes an abstract data type.

The class's implementation – its aggregate private members – usually is hidden from the class user (the programmer who instantiates objects of the class). The class's public interface usually is revealed to the class user in the form of methods: class member functions that operate on the data members of the class.

The class user instantiates objects of the class and invokes the class methods against that object by calling the class's member functions for the object. A class is said to have behaviour, which is another way of saying that an object of the class responds to its methods in ways that are understood by the class user at an abstract level without necessarily revealing the detail of that behaviour.

C++ supports exception handling, which permits a program to make an orderly jump to a defined location higher in the program's executing hierarchy. C++ also supports parameterised types or generality through its template mechanism, a device that enables the programmer to define and instantiate objects of generic data type.

C++ programs consist of multiple source-code modules that are compiled into object-code modules, which are then linked into a single executable program module. Much of the object code in a typical C++ program comes from packaged libraries of previously compiled, reusable software components implemented as classes or functions.

A C++ source-code module is a text-file usually represented in ASCII that you can read and modify with any text editor, such as window-98's notepad.

Unlike languages such as COBOL, BASIC, and FORTRAN, C++ has no built-in input/output statements. Instead, input and output are implemented by C++ classes in standard class libraries. Classes in C++ perform many features that are intrinsic part of other languages instead. Data conversions, string manipulations, and output formatting are three examples of operations that C++ supports with classes and functions taken from standard libraries rather than with intrinsic language features. C++ is a small language capable only of declaring and defining variables, assigning expressions to variables, and calling functions – only one of which, `main()`, is defined as a part of the language proper. C++'s power comes in the way that is extended with classes and functions. User-defined classes and functions further extend the language to support specific problem domains.

## main() Function

Both C and C++ languages require that programs at the minimum should have a `main()` function.

```
int main()
{
 return 0;
}
```

The above example of the usage of main function tells us the following things about the C++ function in general. The first line informs us that the main function always has the identifier `main` and it will return an integer value. `int` in C++ is a keyword, about which we will discuss in details in later

chapters, which informs the functions integer return type. The parenthesis '()' after the function is to contain its parameter list. In this case, there is no parameter, so it's empty.

The function body always starts with the left brace '{' and ends with the right brace '}'. Within them are written the various statements, which are executed when the function is called upon. A brace-surrounded group of statements is called a statement block. In this case there's only one statement. The return statement terminates the function and consequently the program. Please note that the statement has ended with a semicolon. In C++ each statement ends with a semicolon ';'.

A function stops executing either when it has executed it's last statement or when it comes across a return statement. Note, that the return statement can be positioned anywhere inside the function.

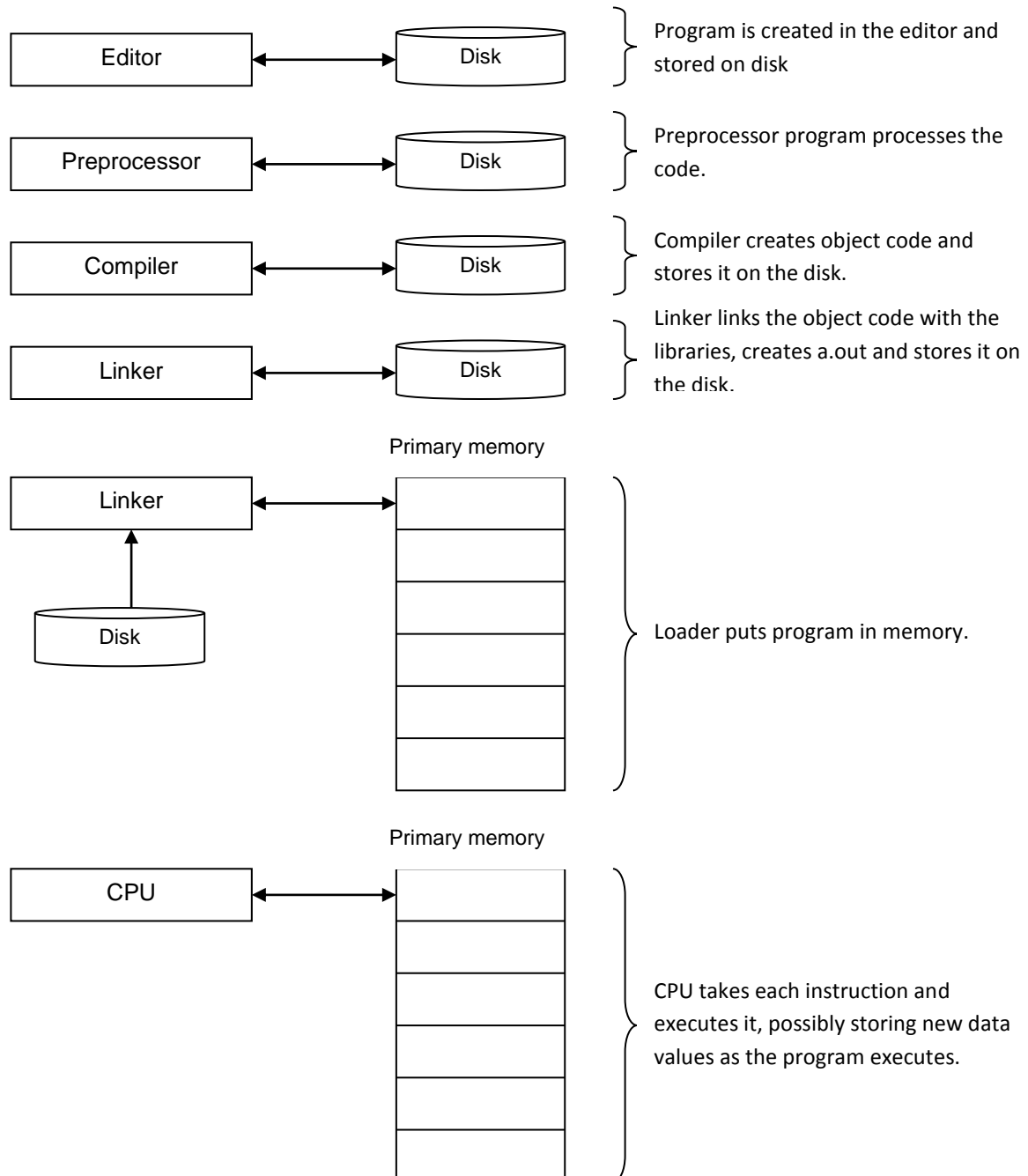
The operating system calls main() and main() returns to the operating system. It's sort of the entry and departure point of your program. In no other place in the program can you call the main() function.

-----

# Chapter -X: Programming in C++

---

## Basics of a Typical C++ Environment



## Phases of C++ Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute

## My first C++ Program

```
#include<iostream.h>
```

```
main ()
{
cout<<"hello, world\n";
}
/*wish.cc*/
```

## Anatomy of a C++ program

```
#include<iostream.h> ----- include information about standard library
```

```
main () ----- Define a function named main
```

```
{ ----- Statements of main are enclosed in braces
```

```
cout<<"hello, world\n"; ----- Main calls library object cout to print this
sequence of characters; \n represents the
new line character
}
```

```
/*wish.cc*/ ----- Comment entry
```

## Component of C++ Program

- comment
- # include directive
- cin
- cout
- >> operator
- << operator
- function
- main()
- data type
- variable



- operator

### Comment:

Comments are used for better understanding of the program statement. The comment entries start with a `/**` symbol and terminates at the end of one line. Comment entries are not compiled.

```
//wish.cc (program name)
```

```
//This program displays the message 'God Bless All'
```

```
//message 'MAY GOD BLESS ALL' is displayed
```

If it's a multi line comment it starts with `/*` and ends with `*/` e.g.,

```
/* This is a program written in C++ to automate the
```

```
Inventory Control System of ABC International Ltd. */
```

### # Include Directive

The `#include` directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. In the program illustrated `"iostream.h"` is included in the source file `"wish.cc"`. The `< >` brackets indicates the compiler to search the header file in the path specified for the header files. The `" "` indicate the compiler to search the header file in the home directory.

### Function

- All C++ programs comprise of one or more function.
- A function name and a function body define a function.
- A function name is defined by a word followed by parentheses.
- All programs must have a function called `main ( )`.
- The execution of a program starts with the `main( )` function.
- The keyword `void` signifies that the function does not return any value.
- A function body is surrounded by curly braces `{ }`. The braces delimit a block of program statements. Every function must have a pair of braces

### Main ()

Every C++ program has a `main ( )`. It provides the entry and exit point to the program. The program begins executing with the first statement in `main ( )` and terminates when the `main()` returns. Both C and C++ programs require minimum `main ( )`.

### Fundamental Data Type

The fundamental data types are at the lowest level, that is, those, which are used for actual data representation in memory. The fundamental data types and their functions are listed in the

following table – Table 1.1. The size of the variable is depended on its type. The size of each type depends on the C++ implementation and is usually expressed as the number of bytes an object of the type occupies in memory. This book uses the data types with sizes typical of 32-bit C++ compiler for the PC.

| Data type | No of bytes | Representation                 |
|-----------|-------------|--------------------------------|
| Bool      | 1           | True / false                   |
| Char      | 1           | Character and strings          |
| wchar_t   | 2           | Foreign language character set |
| Int       | 4           | Integers                       |
| Float     | 4           | Real numbers                   |
| Double    | 8           | Real numbers                   |

## Variables

Variables are fundamental to any language. Values can be assigned to variables, which can be changed in the course of program execution. Each variable declaration in a program can provides the variable's type and identifier. The value assigned to a variable is placed in the memory allocation to that variable. Variables can be created using the keywords `bool`, `char`, `wchar_t`, `int`, `float` and `double`. Variables can have other properties, as well. Integer type specifiers can include `unsigned`, `long`, or `short` to further define the data-type. A `double` can also be a `long double` increasing the precision. We can also use `static`, `extern`, `register`, and `auto` storage classes. There are `const` and `volatile` qualifiers are also available which we will be discussing in the later chapters.

## Identifier

A C++ program consists of many elements – variables, functions, classes, and so on – all of which have names. The name of a function, variable, or class is called its identifier.

- An identifier consists of letters, digits, and the underscore character.
- An identifier must begin with a letter. (Underscores are allowed in the first character, but leading underscores are reserved for identifiers that the compiler defines.)
- Identifiers are case sensitive. For example: `add()` and `Add()` are different.
- An identifier may be of any length, but only first 32 characters are significant.
- An identifier may not be one of the reserved C++ keywords.

## Bool Type

A `bool` variable is a two-state logical type that can contain one of two values: `true` or `false`. If you use a `bool` variable in an arithmetic expression, the `bool` variable contributes the integer value 0 or 1 to

the expression depending upon whether the variable is false or true respectively. If you convert an integer to a bool, the bool variable become false if the integer is zero or becomes true if the integer is non-zero. Variable of type bool typically are used for runtime Boolean indicators and can be used in logical tests to alter program flow.

Sample:

```
#include<iostream.h>

int main()

{
 bool senior; //bool variable

 senior = true; //set to true

 //test the senior variable

 if(senior)

 cout<<"Senior citizen rates apply";

 return 0;

}
```

## Char type

A char variable contains one character from the computer's character set. Characters in PC implementation of the C++ language are contained in 8-bit bytes using ASCII character set to represent character values. A program declares a character variable with a char type specification.

```
char ch;
```

This declaration declares a variable of type char with the identifier ch.

Sample

```
#include< iostream.h>

void main()

{

 char ch; //char variable

 ch='y'; //assign 'y' to ch

 cout<<ch; //displaying 'y'

}
```

🔑 ***The char data type is, in fact, an integer 8-bit numerical type that you can use in numeral expressions just as you use any other integer type. Unless they are unsigned, char variables behave like 8-bit signed integers when you use them in arithmetic and comparison operations.***

***Unsigned char c;***

## Defining Strings

In order to store more than one character, strings are defined.

```
char cWord [10];
```


Allocates ten contiguous bytes to hold a string. The last byte is used to store the string terminator (\0), also known as NULL character.

### Sample:

```
#include< iostream.h >
void main()
{
 char svar1[10]="Anukampa";
 cout<<svar1;
}
```

## wchar\_t Type

C++ includes the wchar\_t type to accommodate character sets that require more than eight bits. Many foreign language character sets have more than 256 characters and cannot be represented by char data type. The wchar\_t data type is typically 16 bit wide.

 ***The standard C++ iostream class library includes classes and objects to support wide characters. To use wchar\_t data type instead of cout, wout is used.***

### Sample

```
#include< iostream.h>
void main()
{
 wchar wc; //wide char variable
 wc='a'; //assign 'a' to ch
 wout<<wc; //displaying 'a'
}
```

## Integer Variable

The basic integer is a signed quantity that you declare with the int type specifier like:

```
int age;
```

An integer can be signed , unsigned, long, short, or a plain signed integer.

```
long int amount; //long integer
```

```

long quantity; //long integer
signed int total; //signed integer
signed kellie; // signed integer
unsigned int offset; //unsigned integer
unsigned offset; //unsigned integer
short smallamt; //short integer
short int tyler; //short integer
unsigned short landon; //unsigned short integer
unsigned short int woody; //unsigned short integer
}

```

The int keyword can be omitted when long, short, signed, or unsigned are specified. With older 16-bit compiler a long integer is of 16 bits and short is of 8 bits. With a 32-bit compiler a long integer is of 32 bits and short is of 16 bits.

### Sample

```

#include< iostream.h>

void main()
{
 int amt; //an int variable
 amt=100; //assign a value
 cout<<amt; //display the int
}

```

### Float Variable

Float variables are used to store floating-point numbers.

```

float amount; //single precision
double BigAmount; //double precision
long double ReallyBigAmout; //long double precision

```

Standard C++ doesn't specify the range of values that floating-point numbers can contain. These range depend on the particular implementation of C++ language.

```
#include< iostream.h>

void main()

{

 float fvar1=10.6;

 cout<<fvar1;

}
```

## Keywords

C++ reserves certain keywords, which can't be used as identifiers.

| Standard C++ Keywords    |              |                  |             |          |
|--------------------------|--------------|------------------|-------------|----------|
| <i>Non – identifiers</i> |              |                  |             |          |
| Asm                      | Do           | inline           | short       | typeid   |
| Auto                     | Double       | int              | signed      | typename |
| Bool                     | dynamic_cast | long             | sizeof      | union    |
| Break                    | Else         | mutable          | static      | unsigned |
| Case                     | Enum         | namespace        | static_cast | using    |
| Catch                    | Explicit     | new              | struct      | virtual  |
| Char                     | Extern       | operator         | switch      | void     |
| Class                    | False        | private          | template    | volatile |
| Const                    | Float        | protected        | this        | wchar_t  |
| const_cast               | For          | public           | throw       | while    |
| continue                 | Friend       | register         | true        |          |
| default                  | Goto         | reinterpret_cast | try         |          |
| delete                   | If           | return           | typedef     |          |

| <i>International C++ Keywords</i> |       |       |        |  |
|-----------------------------------|-------|-------|--------|--|
| And                               | Bitor | or    | xor_e  |  |
| and_eq                            | Compl | or_eq | not_eq |  |
| bitand                            | Not   | xor   |        |  |

## Output Using cout

The statement

```
cout<<"Hello World";
```

Causes the enclosed text to be displayed on the screen.

## Input using cin

```
#include< iostream.h>
```

```
void main()
```

```
{
```

```
 float ftemp, fcels;
```

```
 cout<<"Enter the temperature[F]";
```

```
 cin>>ftemp;
```

```
 fcels=(ftemp-32)*5/9;
```

```
 cout<<"Equivalent Celsius is <<fcels;}
```

The statements:

```
 cin>>ftemp;
```

in the program causes the program to wait for the user to input a value from the keyboard. The value the user enters is stored in variable ftemp.

## Constants

Constants are what some languages call literals and others call immediate values. The constant values are used explicitly in expressions. A constant is distinguished from a variable in two ways.

- It has no apparent compiled place in memory except inside the statement in which it appears.
- The constant can't be addressed; neither the value can be changed.

## Character Constants

Character constants specify values that a char variable can contain. A character constant can be coded with an ASCII expression.

```
A1 = 'x'; //ASCII char constant
```

```
A2 = '\x2f'; //char constant expressed as hex value
```

```
A3 = '\013'; //char constant expresses as octal value
```

## Escape Sequence

The `\` (backslash) begins an escape sequence. It tells the compiler that something extra is coming. The escape sequences apply to character constants and string constants.

| Constant Escape Sequences |                          |
|---------------------------|--------------------------|
| Escape Sequence           | Description              |
| <code>\'</code>           | Single Quote             |
| <code>\*</code>           | Double Quote             |
| <code>\\</code>           | <b>Backslash</b>         |
| <code>\0</code>           | Null (zero) character    |
| <code>\0nnn</code>        | Octal number (nnn)       |
| <code>\a</code>           | Audible bell character   |
| <code>\b</code>           | Backspace                |
| <code>\f</code>           | Formfeed                 |
| <code>\n</code>           | Newline                  |
| <code>\r</code>           | Carriage return          |
| <code>\t</code>           | Horizontal tab           |
| <code>\v</code>           | Vertical tab             |
| <code>\x</code>           | Hexadecimal number (nnn) |

## Integer Constants

An integer constant specifies a long or short, signed or unsigned integer value.

Some compilers support signed short integer values of the range  $-32768$  to  $32767$ , and unsigned short integer values of  $0$  to  $65535$ . These are the ranges that you can represent in a 16-bit integer.

In some compilers the `int` and `long int` are both 32 bits. They support signed long integer values of the range  $-2147483648$  to  $2147483647$ , and unsigned long integer values of  $0$  to  $4294967295$ . These are the ranges that you can represent in a 32-bit integer.



An Integer constant can be specified as a decimal, hexadecimal, or octal value as shown in the following example.

```
Amount = 218; //Decimal integer constant

HexAmt= 0x12fe; //Hexadecimal integer constant

OctalAmt=0177; //Octal integer constant
```

The leading 0x specifies that constant is a hexadecimal expression, which can contain the digits 0-9, and the letters A-F in mixed uppercase or lowercase. A leading zero alone specifies the constant is octal and may contain digits 0-7.

A constant can also be specified as long or unsigned by adding the L or U suffix to the constant.

```
LongAmount = 52388L; //long integer constant

LongHexAmt= 0x4fea2L; //long Hexadecimal constant

UnsignedAmt=40000U; //unsigned integer constant
```

The suffixes can be uppercase or lowercase. On compilers where integer and long integer both occupies the same length, the L suffix is unnecessary; but it should be used if you expect the program to be portable across compiler platforms that support other int lengths.

## Floating-point Constants

A floating-point constant consists of integer and fractional parts separated by a decimal point. Some floating-point constants use scientific, or exponential notation to represent numbers too big or too small to express with normal notation. For example:

```
Pi = 3.14159; //regular decimal notation

SmallNumber = 1.234E-4.0 //1.234 x 10 to the -40th power

BigNumber = 2.47E201 // 2.47 x 10 to the 201st power
```

Floating constants are default double type unless you provide a suffix on the constant:

```
FloatNumber = 1.23E10F //float constant

LongDoubleNumber = 3.45L //long double constant
```

The suffixes can be uppercase or lowercase.

## Address Constants

Variables and functions have memory addresses and C++ enables you to reference their addresses with address constants as shown below:

```
CntPtr = &Ctr; // Address of a variable
FunctPtr = &DoFunction // address of a function
```

## String Constants

“The Word is Beautiful” is a string constant, which is also called string literal. A string is coded as a sequence of ASCII characters surrounded by double quote characters. Escape sequences inside the string constant work the same as they do in character constant. Example:

```
cp = "GOD is Good";
cout<< "\n East or West India is the best";
cout<<"\a GOD Bless India";
```

The first statement apparently assigns a string constant to a variable, but it really assigns the address of the string constant to a pointer variable. The compiler finds a place in a memory for the string constant and compiles its address into the statement.

The same thing happens in the second and third statement in the example. The compiler passes the addresses of the string constants to the cout object. The string constant in the second and third statement includes escape sequences. The second statement’s escape sequence is \n – the newline character and the third statement escape sequence is \a – the audible alarm character.

Adjacent string constants concatenate to form a single string constant. This feature allows coding long string constants on multiple source-code lines. Example:

```
#include <iostream.h>

int main()
{
 cout<<"This is the beginning of a very long message\n"
 "that spans several lines of code.\n"
 "This formal allows a program to build long\n "
 "string constants without going past the \n"
 "program editor's right margin. \n ";
 return 0;
}
```

## Expressions

Statements in a function body consist of individual expressions terminated by the semicolon (;) character. All statements and declarations in C are terminated that way. The statement is not complete until the semicolon appears.

An expression is a combination of constants, variables, function calls, and operators that, when evaluated, returns a value.

### *Sample Expressions:*

```
1+2;
```

```
counter * 3;
```

```
GrossPay - (FICA+GrossPay*WithHoldingRate);
```

Expressions are not independent code of statement. Though they return values, they have no effect because the program does nothing with the returned value. Expressions take on meaning when are placed on the right hand side of assignment statements, or when used as arguments in a function call. The numerical value that the expression returns has a type. The implicit type of an expression depends on the types of variables and constants that contribute to the expression. Therefore, the expression might return an integer of any size, an address, or a floating-point number of any precisions.

## Assignments

An assignment statement assigns to a variable the value returned by an expression. The variable's contents are the value of the expression after the assignment statement.

```
Amount = 1+2;
```

```
Total = counter * 3;
```

```
Netpay = GrossPay - (FICA+GrossPay*WithHoldingRate);
```

Each of the assignment statements assigns an expression's returned value to a named variable.

The variable that receives an assigned value is called **lvalue**, because its reference appears on the left side of this assignment. The expression that provides the assigned value is called the **rvalue**, because it is on the right hand side of the assignment.

## Operators in expression

- ❑ Arithmetic Operator
- ❑ Assignment Operator
- ❑ Relational Operator
- ❑ Logical Operator
- ❑ Increment & Decrement Operator
- ❑ Bitwise Operator
- ❑ Unary Operator
- ❑ Ternary operator

### Arithmetic Operator

In addition to the four arithmetic operators of +, -, \*, /, C also provides the remainder or modulo operator represented by the % symbol. The modulo operator returns the remainder when an integer is divided by another.

#### *Sample Program:*

```
#include<iostream.h>

void main()

{

 int ivar1=11,ivar2=5,ivar3;

 ivar3=ivar1%ivar2;

 cout<<"The remainder is"<<ivar3;

}
```

### Assignment Operator

C++ offers a condensed approach to perform calculations using arithmetic assignment operators combining an arithmetic operator (+, -, /, %) and the assignment operator (=).

```
icount=icount+2;
```

increments the value of the variable icount by 2. The above statement can be modified as

```
icount+=2;
```

using the arithmetic assignment operator are -=, \*=, /=, %=

### Relational Operator

Relational operators are used to compare two values. The result of the comparison is either true (1) or false (0). The following table gives the list of relational operators

## Logical Operator

Logical operators are used to combine two or more test expressions. C++ provides the AND (&), the OR (||) and the NOT (!) logical operators.

The && operator combines two condition expressions and evaluates to true only if both conditions are fulfilled.

## Increment/Decrement Operator

C provides two unusual operator for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand and -- subtracts 1.

## Bitwise Operator

C provides six operators for bit manipulation; these may only be applied to integral operands, i.e, char, short, int, and long, wther singed or unsigned.

|    |                          |
|----|--------------------------|
| &  | bitwise AND              |
|    | bitwise OR               |
| ^  | bitwise exclusive OR     |
| << | left shift               |
| >> | right shift              |
| -  | one's complement (unary) |

## Unary operator

Unary operators are operators that take only one operand.

**Example:** ++, --, \*, &, (type), sizeof, +, -, ~, !

## Ternary Operator

The conditional expression, can be written with the ternary operator "?:".

**Syntax:** `expr1 ? expr2 : expr3`

The expression expr1 is evaluated first. If it is non-zero(true), then the expression expr2 is evaluated, and is the value of conditional expression. Other wise expr3 is evaluated, and that is the value.

## Precedence

| Operator     | Associativity |
|--------------|---------------|
| ( ) [ ] -> . | Left to right |

|                                   |               |
|-----------------------------------|---------------|
| ! - ++ -- + - * & (type) sizeof   | Right to left |
| * / %                             | Left to right |
| + -                               | Left to right |
| << >>                             | Left to right |
| < <= > >=                         | Left to right |
| == !=                             | Left to right |
| &                                 | Left to right |
| ^                                 | Left to right |
|                                   | Left to right |
| &&                                | Left to right |
|                                   | Left to right |
| ?:                                | Right to left |
| = += -= *= /= %= &= ^=  = <<= >>= | Right to left |
| ,                                 | Right to left |

## Practice

1. Write a c++ program to multiply and divide a given number by 2 without using \* (multiplication) and / (divide) operators? (use bit wise operators)
2. Interchange the value of two variables without using a temporary variable.
3. Write a program to calculate the area of a circle for a given value for PI as 3.14159.
4. Write a program to convert the temperature from Fahrenheit to Celsius and vice versa.
5. Write a program that converts the upper case character to lower case and vice-versa.
6. For a given principal, rate of interest, term and number of years calculate the compound interest. (Hint: Use pow() function defined in math.h)
7. Write a program to evaluate the following polynomial.

$$y = \left(\frac{x-1}{x}\right) + \frac{1}{2}\left(\frac{x-1}{x}\right)^2 + \frac{1}{3}\left(\frac{x-1}{x}\right)^3 + \frac{1}{4}\left(\frac{x-1}{x}\right)^4 + \frac{1}{5}\left(\frac{x-1}{x}\right)^5$$

-----

# Chapter – XI: Conditional and Iterative Constructs

---

## Statement:

An expression such as `x=0` or `i++` or `cout<<...` becomes a statement when it is followed by a semicolon, as in

```
x=0;
```

```
i++;
```

```
cout<<... ;
```

The semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

## Blocks

Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement

## If-else

The if-else statement is used to express decisions.

### Syntax:

|                            |                                                                                                                                                                                                                                                              |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>if (expression)</pre> | <i>Else is optional in if statement. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (expression is zero) and if there is an else part, statement2 is executed instead.</i> |
| <pre>    statement1;</pre> |                                                                                                                                                                                                                                                              |
| <pre>else</pre>            |                                                                                                                                                                                                                                                              |
| <pre>    statement2;</pre> |                                                                                                                                                                                                                                                              |

Since an if simply tests the numeric value of an expression, certain coding shortcuts are possible.

```
if (expression)
```

Can be written instead of

```
if (expression != 0)
```

The construction

```
if (expression)
```

```
 statement ;
```



```

else if (expression)

 statement ;

else if (expression)

 statement ;

else

 statement ;

```

This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. The code for each statement is either a single statement or a group of statements. The last else part handles the “None of the above”

## Switch

### *Syntax:*

```

switch(expression)

{

 case const-expr: statement

 case const-expr: statement

 default: statement

}

```

The switch statement is a multi way decision that tests whether an expression matches one of the numbers of constant integer values, and branches accordingly.

The break statement causes an immediate exit from switch. A break statement can be used to force an immediate exit from while, for and do loop.

## Loops – While:

### *Syntax:*

```

while(expression)

 statement;

```

The expression is evaluated. If it is non-zero, statement is executed and expression is re-evaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.

## Loops – For

### *Syntax:*

```
for (expr1 ; expr2 ; expr3)

 statement;
```

Is equivalent to

```
expr1;

while (expr2)
{

 statement

 expr3;

}
```

The three components of a for loop are expressions . Most commonly, expr1 and expr3 are assignments or function calls and expr2 is a relational expression. Any of the three parts can be omitted, though the colons must remain like

```
for(; ;)

{

 ...

}
```

## Loops – Do – while

The while or for loops test the termination conditions at the top. The do...while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

### *Syntax:*

```
do

{

 statement;

}while(expression);
```

## Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while or do, just as from switch. A break causes the innermost closing loop or switch to be exited immediately.

## Goto and Labels

C provides the infinitely- abusable goto statement, the labels to branch to.

```
for (...)
 for (...)
 {
 ...
 if (disaster)
 goto error;
 }...
```

Error:

```
 clean up the mess
```

## Practice

1. Write a c program to check whether an input year is leap year or not.
2. Write a c program to find maximum of three numbers.
3. Write a program to check out a number is Armstrong or not.
4. Write a program to print the following format

1

22

333

4444

5. Write a program to construct a pyramid of digits:

1

2 3 2

3 4 5 4 3

4 5 6 7 6 5 4

6. Create a class called 'student' that contains a name and a student roll no. Include a member function to get data from the user and function to show data.
7. Write a program to compute the exponential value of a given number x using the series.  
$$e(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$
8. Write a program to generate reverse pyramid of digits.
9. Write a program to print multiplication table of any given number.
10. Write a program to compute the area of right triangle and triangle using switch statement.  
Case 1 find area of right triangle and case 2 find area of triangle.
11. Write a program to find the Fibonacci series up to 50 and find sum and average.
12. Write a program to find the Factorial of a number given by user using recursive method.
13. Write a program to check the input number is palindrome or not.
14. Write a program to convert a decimal number to hexadecimal number.
15. Write a program to accept 0,1 or 2. if 0 is entered by user, accept the necessary parameter to calculate the volume of a cylinder. Inputs 1 and 2 correspond to cylinder and cone, respectively. This process goes until the user enters q to terminate the program. (Use switch case statement)
16. Write a program to accept a string, sum all of its ASCII values and print it.
17. Write a program to input a line, count the number of space, vowels, tab characters are present.

# Chapter – XII: Function & Storage Types

---

## Basics of Functions

Functions are the building blocks of C++ programs. A function groups a number of program statements into a single unit. This unit can be invoked from other parts of the program.

## Need of Functions

When programs become large, a single list of instructions becomes difficult to understand. For this reason, functions are adopted. A function has a clearly defined purpose and interface to other functions in the program. A group of functions together form a larger entity called a module.

Another reason for using functions is to reduce the program size. Any sequence of instructions that is repeated in a program can be grouped together to form a function. The function code is stored in only one place in the memory.

## Functions - Example

The following program illustrates the use of a function:

```
/*This program uses a function to calculate the sum of two numbers*/
#include<iostream.h>

void add(); /*function prototyping*/

int main()
{
 cout<< "Calling function add()" ;
 add(); /*A call to the function add()*/
 cout<<"\nA return from add() function";
 return 0;
}

/*Function definition for add*/
void add()
{
 int inum1, inum2, inum3;
 cout<<"Input two numbers";
 cin>>inum1;
 cin>>inum2;
 inum3=inum1+inum2;
 cout<<"\nThe sum of the two numbers is"<< num3; }
```

The function add() is invoked from main( ). The control passes to the function add(), and the body of the function gets executed. The control then returns to main( ) and the remaining code gets executed.

### *Sample output :*

Calling function add()

Input two numbers

10

20

The sum of two numbers is 30

A return from add( ) function.

The function main ( ), which invokes the function add( ), is called the **calling function**, and the function add( ) is **called function**.

## Functions and Arguments

Argument(s) of a function is (are) the data that the function receives when called from another function.

It is not always necessary for a function to have arguments. The function add( ) in last Program did not contain any arguments.

### *Example*

```
#include< iostream.h>
void add(int ivar1,int ivar2)
/* function containing two arguments*/
{
 int ivar3=ivar1+ivar2;
 cout<<"The sum of the two numbers is"<<var3;
}
main()
{
 int inum1,inum2;
 cout<<"Input two numbers";
 cin>>inum1;
 cin>>inum2;
 add(inum1,inum2);/*passing two values to the function add()*/
}
```

## Arguments Vs Parameters

The caller of a function passes expressions to the function as arguments. The apparently interchangeable terms argument and parameter appear frequently when we discuss about the values passed.

A function has parameters, which are the variables into which the values passes are copied before the function begins executing. The function prototype declares the parameter types in its parameter

list. The caller of the argument passes arguments, which are values returned from expressions to be copied into the function's parameter value.

## Declaring Functions Using Prototypes

Standard C++ has stronger type checking. Just as any variable is declared in a program before it is used, it is necessary to declare or prototype a function to inform the compiler that the function would be referenced at a later point in the program. To call a function, the function must be declared with respect to its return and parameter type. The declaration of the function is called prototype.

```
unsigned int BuildRecord(char, int);
void GetRecorder(char, int, int);
int isMonday();
```

is a function declaration or a prototype.

## Unnamed Parameter Types

The parameter lists in the prototype contain type specification with no identifier for the parameters. The parameter identifiers can be put into prototypes, but the identifiers serve as documentation only. They need not correspond to the same identifiers in the function definition's parameter list.

## Functions returning void

The void return type means the function returns nothing. A void function can't return a value and it can't be called in context of an expression where it is expected to return a value.

## Function with no parameters

```
int isMonday();
```

This prototype defines a function with no parameters. Arguments can't be passed to a function with an empty parameter list.

## Function with variable parameter list

The standard C library `printf()` and `scanf()` are defined as accepting a variable number of arguments. Function with variable argument lists requires special handling.

## Functions returning nothing and no parameters

These types of functions perform some task like, using data taken from external sources or maintained internally. The caller can invoke the function and is not benefited from its execution directly, except when the caller uses an external data item that the called function modifies.

## Standard Library Prototypes

The prototypes for all the standard library classes and functions are in their respective header files. That is why <iostream> is included.

## Function without prototypes

Prototype is not needed when the function definition itself appears in the code ahead of any call to it. To preserve the top-down representation of a program's execution, generally programmers do not go for it.

## Return from functions

A function stops executing in one of the several ways listed below.

One way, is by calling the standard exit() function, which terminates both the calling function and the program.

Falling through the bottom of its definition can terminate a function. In this case the function should return void, because falling through the bottom returns no value.

A function can execute the return statement from within its body. A void-returning statement can use the return statement without any value.

## Ignoring return value

When a value-returning function is called in context where no value is expected i.e., in which the return value is not used, no value is assigned to anything, not is used as the argument to another function, in this case the function executes, and it dutifully returns its value. But the caller can choose to ignore it.

## Defining and calling a Function

A function is defined when a code is written for its function header and statement body. When a function has a prototype, the function definition must match the prototype exactly with respect to the types of its return value and parameters.

The function definition is:

```
void add()
{
 int inum1, inum2, inum3;
 cout<<"Input two numbers";
 cin>>inum1>>inum2;
 inum3=inum1+inum2;
 cout<<"The sum of the two numbers is"<<inum3;
}
```



A function declaration can be avoided if the function definition appears before the first call to the function.

## Calling Function

In C++ programs, functions that have arguments can be invoked by:

- Value
- Reference

### Call by value

The following program illustrates the invoking of a function by value:

```
/*This function swaps the value of two variables*/
#include< iostream.h>
void swap(int,int);
void main()
{
 int ivar1,ivar2;
 cout<<"Input two numbers";
 cin>>ivar1>>ivar2;
 swap(ivar1,ivar2);
 cout<<ivar1<<ivar2;
}
void swap(int inum1,int inum2)
{
 int itemp;
 itemp=inum1;
 inum1=inum2;
 inum2=itemp;
 cout<<inum1<<inum2;
}
```

Sample output of Program

Input two numbers

15

25

25 15

16 25

values entered for the variable ivar1 are passed to the function swap( ). When the function swap( ) is invoked, these values get copied into the memory locations of the parameters inum1 and inum2, respectively,

| ivar1 | ivar2 | ivar1 | ivar2 |
|-------|-------|-------|-------|
| 15    | 25    | 15    | 25    |
| 15    | 25    | 25    | 15    |
| inum1 | inum2 | inum1 | inum2 |

When the swap function invoked

After the function executed

Thus the variables in the caller function [main()] are distinct from the variable in the called function [swap( )] as they occupy distinct memory locations.

**Note:**

*The values of the variable in the earlier do not get affected when the arguments are passed as value.*

### Call by reference

A reference is a constant pointer to the argument. When arguments are passed by value, the called function creates a new variable of the same type as the arguments, and copies the argument values into it. Passing arguments by value is useful when the function does not need to modify the values of the original variables in the calling program.

### Reference Argument

In passing reference arguments, a reference to the variable in the calling program is passed.

```
/*This program swaps the values in the variable using function
containing reference arguments*/
#include<iostream.h>
void swap(int &,int &);
void main()
{
 int ivar1, ivar2;
 cout<<"Enter two numbers";
 cin>>ivar1>>ivar2;
 swap(ivar1,ivar2);
 cout<<ivar1<<ivar2;
}
```

```

void swap(int &inum1,int &inum2)
{
 int itemp;
 itemp=inum1;
 inum1=inum2;
 inum2=itemp;
 cout<<inum1<<inum2;
}

```

Reference arguments are indicated by an ampersand (&) preceding the argument:

```
int &inum1;
```

The ampersand (&) indicates that inum1 is an alias for ivar1 which is passed as an argument. The function declaration must have an ampersand following the data type of the argument.

```
void swap(int &,int &)
```

The ampersand sign is not used during the function call:

```
swap(ivar1,ivar2)
```

Sample output of Program

Enter two numbers

12

24

24        12

24        12

In Program, the values in the variables ivar1 and ivar2 in the calling program are swapped.

|                                |       |                             |       |
|--------------------------------|-------|-----------------------------|-------|
| ivar1                          | ivar2 | ivar1                       | ivar2 |
| 15                             | 25    | 25                          | 15    |
| inum1                          | inum2 | inum1                       | inum2 |
| When the swap function invoked |       | After the function executed |       |

## Unnamed Function parameter

A C++ function can be declared with one or more parameters that the function does not use. This circumstance often occurs when several functions are called through a generic function pointer. Some of the functions do not use all the parameters named in the function pointer declaration.

### Example:

```
int func(int x, int y)
{
 return x*2;
}
```

Although this usage is correct and common, most C & C++ compilers complain that you failed to use the parameter `y`. C++, however allows to declare function with unnamed parameters to indicate to the compiler that the parameter exists, and the callers pass an argument for the parameter – but that the called function doesn't use it. Example:

```
int func(int x, int)
{
 return x*2;
}
```

## Default function argument

A C++ function prototype can declare that one or more of the function's parameters have default argument values. When a call to the function omits the corresponding arguments, the compiler inserts the default values where it expects to see the arguments.

```
void myfunc(int=5, double=1.23);
myfunc(12,3.45); //overrides both defaults
myfunc(3); //effectively myfunc(3,1.23)
myfunc(); // effectively myfunc(12,1.23)
```

## Inline Functions

You can tell the C++ compiler that a function is **inline**, which in turn, compiles a new copy of the function each time it is called. The inline function execution eliminates the function-call overhead of traditional functions. Inline functions are only used when the function is small or when relatively there are few calls to them. The C++ standard does not define where an inline function must be declared as such and under what conditions the compiler may choose to ignore the inline declaration, except to say that the compiler may do so. You can declare an inline function (for may be performance purpose) but the compiler may overrule you without saying it.

```
#include<iostream.h>
#include<cstdlib.h>
inline void error_msg(char *s)
{
 cout<<'\\a'<<s;
 exit(1);
}
```

```
int main()
{
 error_msg("You called ?");
}
```

Using inline supports two idioms.

- It offers an improved macro facility.
- It permits the programmer to break a large function with many nested level of statement blocks into several smaller inline functions. This usage improves a program's readability without introducing unnecessary function call overheads.

## Recursion

All C++ functions are recursive, which means that a function can call itself, either directly or indirectly, by a lower function that is executing as a result of a call made by the recursive function.

Functions are recursive because each execution of the function has private copies of the arguments and local data objects, and those copies are distinct from the copies owned by other executions of the same function.

### Example:

```
include<iostream.h>
/*printfd : print n in decimal*/
void printfd(int n)
{
 if (n<0)
 {
 cout<<\"-\";
 n = -n;
 }

 if (n/10)
 printfd(n/10) ;
 putchar(n % 10 + '0');
}
```

## Overloading of Function

In C++ we can assign the same function name to multiple functions but the functions should have different parameter list. Then all versions of the function would be available at the same time. This is called *function overloading*. A function is overloaded to get a different algorithm on similar data. Another reason to overload a function is to get the same result from data values that can be represented in different formats.

```
#include<iostream.h>
```

```

#include<ctime.h>
// The first version of display function
void display_time(const struct tm *tim)
{
 cout<<"1. It is now "<< asctime(tim);
}
// The second version of display function
void display_time(time_t* tim)
{
 cout<<"2. It is now "<< ctime(tim);
}
//The main()
int main()
{
 time_t tim=time(0);
 tm* ltim=localtime(&tim);
 display_time(ltim);
 display_time(&tim);
return 0;
}

```

The above program uses the Standard C data formats `time_t` and `struct tm`, loading then with the value with the value of current date and timing using the standard C `time()` and `localtime()` functions. Then, the program calls its own overloaded `display_time()` function for each of the formats. The results are as follows:

1. It is now Mon Jan 27 12:05:20 2000
2. It is now Mon Jan 27 12:05:20 2000

## Type safe Linkage

If several functions are given the same name, what reconciles the apparent conflicts between similarly named functions that are declared in different translation units (source code files)? It seems that the linker (It links the object code with the libraries, creates .exe file and stores it on the disk.) cannot know which function to use in resolving a reference to one of those functions. Function linkages are not type-safe which means that confusion may arise while the data type checking of the function is done.

C++ solves this problem by applying a process called **name mangling** to the compiler's internal identifier of a function. The mangled name includes tokens that identify the function's return type and the type of its arguments. Calls to the function and the function definition itself are recorded in

the re-locatable object files as references to the mangled name – which is unique even though several functions can have the same unmangled name.

Mangled names also transcend the use of prototypes to ensure that functions match their calls. Unlike C, the C++ type checking system cannot be overridden simply by using different prototypes for the same function.

Algorithms for name mangling vary among compilers, but that is of no concern to programmers.

## Linkage Specification

Although C++ mangles function names, other languages do not. (in particular, C compilers and assembly language assemblers). This presents a problem, because a C++ program must be able to call functions that are compiled or assembled in other languages, and a C++ program must be able to provide functions that are called from other languages. If the C++ compiler always mangled function names internally, references to external functions in other languages would not be resolved properly.

All the standard C functions are compiled by the C compiler component of any C++ development system. Consequently their names are not mangled internally. Other function libraries may be employed in the project too. Without some method for telling C++ compiler not to mangle references to those functions, they never can be called from a C++ program.

C++ employs the linkage specification to make functions compiled by compilers of other languages accessible to a C++ program. Unless the C++ compiler is told otherwise, it assumes that external identifiers are subject to name mangling. That is why the C++ compiler is to be told when a function has been compiled with different linkage conventions.

The flowing code shows how a linkage specification tells the C++ compiler that a C compiler compiled the function in a header file.

```
extern "C" // the linkage specification
{
#include "myhdr.h" //tells C++ that functions in the //library
were compiled with C
}
int main()
{
 return foobar(); //calls a C function
}
```

A typical C library header file is given below.

```
/* - - - typical Standard C library header file - - - */
#ifdef __cplusplus
extern "C" {
#endif
/* Header file contents */
ifdef __cplusplus
}
#endif
```

## Scope of a variable Vs. Lifetime

The term scope is often used to describe an identifier's lifetime, which is the period of time from when the program creates an object to when the object is destroyed. This usage is not altogether accurate and reflects a typical degree of ambiguity in the way we talk about programming.

A variable declared within the function's statement block exists from the point of the declaration to the point that the program exits the statement block. As long as program execution stays within that statement block, or within a block nested inside the statement block, the variable is alive. The period of time is its lifetime. If while the variable is still alive, the program calls another function, then the variable effectively goes out of the scope until the function returns. The variable is no longer in scope because the executing function can't reference the variable.

## Scope rules

Data storage types determine how storage is allocated to the variable. The storage types supported by C++ are :

- Auto
- Static
- Extern
- Register

### Auto Storage Type

Data pertaining to a function is lost when the function has been executed completely. Variables defined in a function are in memory and retain their value only as long as the function is in execution.

In C++, such data has been classified as data of storage type auto. So far, all functions have been written using auto type data. Declarations like:

Example:

```
int ivar; char cvar; invoice ivar1;
```



are, by default, treated as:

```
auto int ivar1; auto char cvar1; auto invoice ivar1;
```

The word auto may be used in declarations for clarity.

### Static Storage Type

As opposed to auto type data, C++ also offers static type data, which, as its name suggests, retains its value even after the function to which it belongs has been executed. The following example illustrates the difference between auto and static type data.

#### Example:

```
#include<iostream.h>
void dummy(void);
void main()
{
 int ivar;
 for(ivar=0;ivar<3;ivar++)
 {
 dummy();
 }
}
void dummy(void)
{
 int ictr=1;
 cout<<"Function executed" << ictr << "times";
 ictr++;
}/* program 1*/

/*Program to differentiate between auto and static type data*/
#include<iostream.h>
void dummy();
void main()
{
 int ivar;
 for(ivar=0;ivar<3;ivar++)
 {
 dummy();
 }
}
```

```

void dummy(void)
{
 static int ictr=1;
 cout<<"Function executed" << ictr << "times";
 ictr++;
}/* program 2 */

```

In the programs shown, the function dummy() is used to print the number of times the function has been called. Execution stops when the value of the variable ictr in main reaches three. The output of Program – 1 would be:

Function executed 1 time

Function executed 1 time

Function executed 1 time

Whereas that of Program – 2 would be

Function executed 1 time(s)

Function executed 2 time(s)

Function executed 3 time(s)

This is because, in Program – 1 the variable ictr is declared as auto type data. Therefore, every time the function dummy( ) is called, the variable gets created and initialised to 1. But the static declaration in Program – 2 ensures that the variable is declared and initialised only once, that is, when the function is invoked the very first time.

***Arrays declared as static can be initialised within a function at the time of declaration.***

### Extern Storage Type

Apart from auto and static types, a variable can also be declared in a manner such that it is available to all functions in a program file, that is, it is a global variable. A variable declared outside a function is called a **global variable**.

#### Example:

An example of this type of data is given below

```

#include<iostream.h>
int ival;
void main()
{
 cout<<"Enter value:";
 cin>>ival;
 disp();
}

```

```

/*program - 3*/
void disp()
{
extern int ival;
//extern is used to refer to ival declared in last Program
cout<<"Value entered is"<<ival;
}/*program - 4*/

```

In the above example, Program 3 declares a global variable called ival and also accepts a value from the user. Then, it calls a function disp( ) which is in another program file, namely Program 4.

Program 4 refers to the variable ival using the extern declaration since the ival is declared in Program 3 and not in program 4. Sample output after the two programs are compiled together and linked would be as follows:

Enter value: 10

Value entered is 10

### Register Storage Type

A register declaration advises the compiler that the variable in question will be heavily used. The idea is that the register variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

#### Example:

```

register int x;
register int c;

```

A variable declared with the register storage class is the same as an auto variable, except that the program cannot take the variable's address. The register storage class is a relic whose purpose is to allow the programmer to specify conditions under which that program's performance can be improved, if certain local, automatic variables are maintained in one of the computer's hardware registers. This states the programmer's intention to use the variable in ways that might work best if the variable resided in the hardware register rather than in the computer's main memory. The register storage class is only a suggestion to the compiler that the variable occupy a register. The compiler can ignore the suggestion.

The following table summarizes the three storage types.

## A Comparison

| Storage Type | Created                                                                     | Initialized                                                                                       | Can be accessed                                 |
|--------------|-----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|-------------------------------------------------|
| Auto         | Each time the function is invoked                                           | At the time of declaration or later                                                               | Only in the function in which it is declared    |
| Static       | The first time the function is invoked                                      | At the time of declaration so that the value is not reset; static arrays can also be initialized. | Only in the function in which it is declared.   |
| Extern       | Created at the time when declared as global, but not where declared extern. | At the time of declaration or later.                                                              | In any function in same/different program file. |

## Type Qualifiers

C++ includes two type qualifiers, the keywords `const` and `volatile`, that further define the nature and behaviour of variables.

### The `const` type qualifier

A `const` variable is one that the program may not modify, except through initialisation when the variable is declared. The phrase “`const` variable” seems to be an oxymoron. How can something be constant and variable at the same time? Nonetheless, the usage is common among programmers.

```
#include<iostream.h>
int main()
{
 const int maxctr=30;
 for(int ctr=100;ctr<maxctr;ctr+=50)
 cout<<"Ctr = "<<ctr<<endl;
 return 0;
}
```

The `const` variable might or might not occupy memory, have an address, and be used in any context that does not modify the contents of the variable. Whether it has an address or not depends upon how you use it and on the C++ compiler implementation. If you take the address of the `const` variable, the compiler must give it a memory location. Otherwise, the compiler is free to treat a reference to the expression as if it were coded as a constant in an expression. When a variable is qualified as a `const`, the compiler prevents the program from modifying the variable's content.

## The volatile type qualifier

A volatile variable is the opposite of a const variable. The volatile type qualifier tells the compiler that the program can change the variable in unseen ways. Those ways are implementation dependent. One possibility is that a variable can be changed by an asynchronous interrupt service routine. The compiler must know about such a variable so that the compiler does not optimise its access in ways that defeat the external changes.

```
#include<iostream.h>
volatile int value = 300;
int main()
{
 int counter;
 for (counter=100;counter< value ; counter += 50)
 cout<<"Counter = "<<counter<<endl;
 return 0;
}
```

Suppose that a program posts the address of a variable in an external pointer and that an interrupt service routine elsewhere in the program or in the system modifies the contents of the variable by dereferencing the pointer. If the compiler has optimised the function by using a register for the variable while the program uses its contents, the effects of the interrupt could be lost. The volatile type qualifier tells the compiler not to make such optimisations.

## Dynamic Memory Allocation

### New :

Pointers provide the necessary support for C++'s powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running.

In the programs discussed in the earlier sections, it has been necessary to declare arrays to some approximate size. This technique can waste memory if the amount of data is much less than the maximum, and it is not always possible to predict what size of the array will be. It would be desirable to start the program and then allocate memory as the need arises. The new operator provides this capability.

The syntax for the new operator is:

```
<variable>=new <type>;
```

where

<variable> - pointer variable

<type> - char, int or float

The type of variable mentioned on the left hand side and the type mentioned on the right hand side should match.

Consider the following examples,

```
char *cptr;
```

```
cptr=new char;
```

allocates the number of bytes equivalent to a char, that is, one byte.

```
int *iptr;
```

```
iptr=new int;
```

allocates four bytes of memory and assigns the starting address to iptr. The syntax of new operator can also be modified to allocate memory of varying requirement. For example

```
char *cptr;
```

```
cptr=new char[10];
```

Allocates ten bytes of memory and assigns the starting address to cptr.

### Delete:

The delete operator is used to release memory.

The syntax is

```
delete <variable>
```

where

<variable> - pointer variable

### Practice

1. Write a program to check whether a input string is palindrome or not using function.
2. Write functions named as `create_mtrx()`, `add_mtrx()`, `mul_mtrx()` for matrix .
3. Write a function to swap two variables using call by reference.
4. Write a function of tower of Hanoi using recursion.
5. Write a program for adding integer parameters passed as command line arguments.
6. Write a calculator program using functions.
7. Write a function that accepts a number, reverses and returns it.
8. Write a function that will accept two character arrays as parameter, concatenate the second array at the end of the first array.
9. Write a function to accept the radius of the circle and find the area of it.
10. Write a function (recursive) to find the factorial of a number.

-----

# Chapter – XIII: OOPs

---

## Object Oriented Programming Structure

### Class:

A user defined data type that may consist of data members and member functions.

When both, function that operate on the data and the data are combined in a single unit, this unit is called a Class. Example: Human being

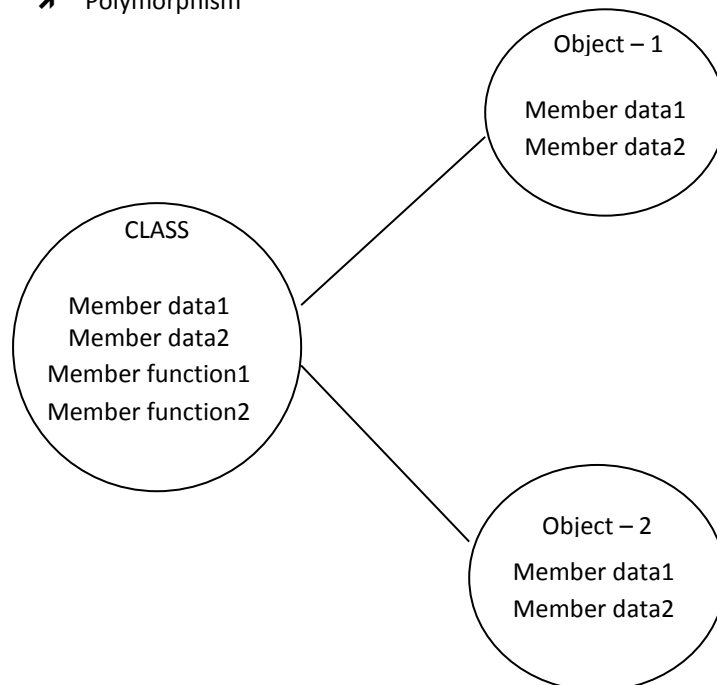
### Object:

Objects are instance of a class. Example: Ram, Shyam, Radha etc.

The functions in an object are called member functions, which are generally known as methods and the data in an object are called the member data in Object Oriented Languages.

### Characteristics of an Object Oriented Language:

- Class
- Inheritance
- Reusability
- Polymorphism



**All object share the same copy of member function but maintain a separate copy of the member data.**



## **Class**

### ***Base Class:***

The class from which another class has been derived is called the base class.

### ***Derived Class:***

The class, which inherits the property of a base class is known as the derived class.

Each existence of a derived class includes all members of the base class. Since the derived class inherits all properties of the base class, the derived class has a larger set of properties than the base class. The derived class may also can override some or all the property of the base class.

### **Data Access Specifier:**

An object defined outside the class can access only the public members of the class. So private members of a class cannot be directly accessed from outside the class. Even though the derived class inherits all the member data and functions from the base class, private members of the base class are not directly accessible from the derived class. For this purpose Protected members are to be declared which behave like public members to the derived class and like private members with respect to the rest of the program.

### ***Public Data & Functions:***

Can be accessed outside the class

### ***Private Data & Functions:***

Cannot be accessed outside the class

### ***Protected Data & Functions:***

Can be accessed by the class and its derived class

## **Inheritance**

Inheritance is the process of creating a new class called the derived class, from the existing class, called the base class.

### **Reusability:**

A programmer can take an existing class and, without modifying it, can add additional features and capabilities to it. Just deriving a new class from an existing class can do all these.

## Polymorphism:

The word polymorphism is derived from two Latin words poly(many) and morphs (forms). The concept of using operators or functions in different ways, depending on what they are operating on, is called polymorphism.

### *A sample program (header file - 1)*

```
//Computer.h
class computer
{
private:
 int ram_space;
 int proc_cap;
public:
void accept()
{
 cout<<"Enter the Memory Size";
 cin>>ram_space;
 cout<<"Enter the Processor Capacity";
 cin>>proc_cap;
}
void display()
{
 cout<<"Memory Size"<<ram_space<<endl;
 cout<<"Processor Capacity"<<proc_cap<<endl;
}
};
```

### *A sample program (header file – 2)*

```
//Computermm.h
class comp_media : public computer
{
private:
 int cd_drive;
public:
void accept()
{
 computer :: accept();
 cout<<"Enter the cd drive capacity";
 cin>>cd_drive;
}
```

```

void display()
{
 computer :: display();
 cout<<"The CD Drive capacity"<<cd_drive<<endl;
}
};

```

### *A sample program*

```

#include<iostream.h>
#include "computer.h"
#include "computermm.h"
int main()
{
 computer c;
 comp_media cm;
 c.accept();
 c.display();
 cm.accept();
 cm.display();
}

```

## **Constructor**

When an instance of a class comes into the scope, a special function called the constructor executes. You can declare one or more constructor functions when you declare the class. If you do not declare at least one constructor function, the compiler provides a hidden default constructor function for the class, which may or may not do anything.

The runtime system allocates enough memory to contain the data members of a class when an object of the class comes into scope. The system does not necessarily initialise the data members. Like objects of intrinsic data types, the data members of external objects are initialised to all zeros. The runtime system does not initialise local objects. The data member may be assumed to contain garbage values. The class's constructor function must do any initialisation that the class requires. The data memory returns to the system when the automatic data object goes out of the scope. Dynamically allocated class objects work the same way except that you must use new and delete operators to allocate and de-allocate memory for the object.

If the constructor function has an empty parameter list, the declaration of the object does not require the parenthesis. A constructor function returns nothing. You do not declare it as void but it is

void by default. You may define multiple overloaded constructor function for a class. Each of them must have a distinct parameter list.

### Constructors with Default Argument

```
#include<iostream.h>

class box
{
 int height,width,depth;
public:
 box(int ht=1; int wd = 2; int dp=3)
 {
 height=ht; width = wd; depth = dp;
 }
 // Member function
 int volume()
{
return height * width * depth;
}
};

int main()
{
 box thisbox(7,8,9);
 box defaultbox;
 int volume=thisbox.volume();
 cout<<volume;
 volume=defaultbox.volume();
 cout<<volume;
 return 0;
}
```

### Default Constructor

A constructor with no parameters or a constructor with default arguments for all its parameters is called default constructor. If you do not provide any constructors, the compiler provides a public default constructor, which usually does nothing. If you provide at least one constructor of any kind, the compiler does not provide a default constructor. Default constructors are important, as you will learn later.

## *Overloaded Constructor*

A class can have more than one constructor function. Such constructor functions for a class, however must have different parameter lists with respect to the number and types of parameters so that the compiler can tell the constructor apart.

### *Example : A class with two constructors*

```
#include<iostream.h>

class box
{
 //Private data members
 int height,width,depth;
public:
 //overloaded constructors
 box()
 { // does nothing }

 box(int ht, int wd, int dp)
 {
 height = ht;
 width = wd;
 depth = dp;
 }
 int volume()
 {
 return height * width * depth;
 }
};

int main()
{
 //define two box objects
 box thisbox(7,8,9);
 box otherbox;
 //assign the otherbox value to this box
 otherbox=thisbox;
 //get and display the volume
 int volume=otherbox.volume;
 cout<<volume;
 return 0;
```

```
}
```

## Destructor

When a class object goes out of the scope, a special function called the destructor is called. You define the destructor when you define the class. The destructor function name is always that of the class with a tilde (~) as a prefix.

There is only one destructor function per class. A destructor function takes no parameter and returns nothing. The destructor's job is to undo whatever the class object has done that needs to be undone, such as releasing the allocated dynamic heap memory.

## Practice

1. Create a C++ class for a 'Stock item' abstract data type. It should have the attribute of stock level (an integer) and unit price (a float). Define methods to return the values of these two attributes and set them using parameters. Add two more methods to allow stock receipts, and stock issues, updating the stock level as appropriate.
2. Create 'bank account' abstract data type having the following member data and functions: account number, account holder, current balance, get account number (), get account holder(), get current balance(), set account number(), set account holder(), deposit(), withdrawal() . Depending upon the debit or credit done by the customer update the balance field.
3. Create a C++ class to represent a person with attributes of name, year of birth and height in metres. Define methods to set these three attributes. Add a method which will return a person's approximate age when given a year as a parameter. Add another method, which will return their height in centimetres.
4. Create a 'string' class that is having the following member functions: get string(), copy string(), reverse string(), concatenate string(), set string().

-----

# Chapter – XIV: Inheritance

---

## Implementation of Inheritance in C++

The philosophy behind inheritance is to portray things as they exist in the real world; it is a very important feature of Object Oriented Programming. It has many advantages, the most important of them being the reusability of the code. Once a class is defined and debugged, it can be used to create new subclasses. The reuse of existing classes saves time and effort.

The class from which, another class has been derived is called the base class. The class inherits the properties of the base class is called the derived class. Each instance of the derived class includes all members of the base class. Since the derived class inherits all properties of the base class, the derived class has a larger set of properties than its base class. However, the derived class may override some or all the properties of the base class.

Any class can be a base class. More than one class can be derived from a single base class, and a derived class can be the base class to another.

If class B is inherited from A, then the syntax of declaring a derived class is :

```
class B: public A
```

The single colon in the derived class declaration is used to specify the base class.

As an object defined outside the class can access only the public member of the class, therefore private members of a class cannot be directly accessed from outside the class. This holds true even for a derived class. Even though the derived class inherits all member data and functions from the base class, private members of the base class are not directly accessible from the derived class. The idea behind is never to compromise on encapsulation implemented through data hiding.

So to solve this problem, the protected access specifier is used. The protected members of a class can be accessed by its member functions, or within any class derived from it. Protected members behave like public members with respect to the derived class, and like private members with respect to the rest of the program.

The following program implements inheritance using the protected access specifier.

### Example

```
#include<iostream.h>
class furniture
{
protected:
 int iColor, iWidth, iHeight;
};
```



```

class bookshelf : public furniture
{
private:
 int iNo_shelves;
public:
 void accept(void)
 {
 cout<<"\nEnter the Color \t";
 cin>>iColor;
 cout<<"\nEnter the Width \t";
 cin>>iWidth;
 cout<<"\nEnter the Height \t";
 cin>>iHeight;
 cout<<"\nEnter the No of Shelves\t";
 cin>>iNo_shelves;
 }
 void display(void)
 {
 cout<<"Color is "<<iColor<<endl;
 cout<<"Width is "<<iWidth<<endl;
 cout<<"Height is "<<iHeight<<endl;
 cout<<"No of Shelves are "<<iNo_shelves<<endl;
 }
};

class chair : public furniture
{
private:
 int iNo_legs;
public:
 void accept(void)
 {
 cout<<"\nEnter the Color \t";
 cin>>iColor;
 cout<<"\nEnter the Width \t";
 cin>>iWidth;
 cout<<"\nEnter the Height \t";
 cin>>iHeight;
 cout<<"\nEnter the No of legs\t";
 cin>>iNo_legs;
 }
 void display(void)
 {
 cout<<"Color is "<<iColor<<endl;
 cout<<"Width is "<<iWidth<<endl;
 cout<<"Height is "<<iHeight<<endl;
 cout<<"No of Legs are "<<iNo_legs<<endl;
 }
};

```

```

int main()
{
 bookshelf BS1;
 chair C1;
 BS1.accept();
 BS1.display();
 C1.accept();
 C1.display();
 return 0;
}

```

## Overloading Base Class Members

In C++, a base class member can be overloaded by defining a derived class member with the same name as that of the base class member.

### Example

```

//override.cpp
#include<iostream.h>
class A
{
public:
 void display(void)
 {
 cout<<"\n Base";
 }
};
class B: public class A
{
public:
 void display(void)
 {
 cout<<"\n Derived";
 }
};
void main()
{
 B b1;
 b1.display();
}

```

In the above-mentioned program the function display in the class A has been overridden by declaring the same function once again in the derived class, namely class B. If the above program will be executed then it will display – Derived.

The decision compiler takes if the same function is present in both the derived class and the base class, is as follows:

- If the function is invoked from an object of the derived class, then the function in the derived class is executed.

- If the function is invoked from an object of the base class, then the base class member function is invoked.

### Scope Resolution with Overloaded Members

To access the base class member from a derived class, the scope resolution ‘::’ operator is used.

#### Example

```
#include<iostream.h>
class furniture
{
private:
 int iColor, iWidth, iHeight;
public:
 void accept(void)
 {
 cout<<"\nEnter the Color \t";
 cin>>iColor;
 cout<<"\nEnter the Width \t";
 cin>>iWidth;
 cout<<"\nEnter the Height \t";
 cin>>iHeight;
 }
 void display(void)
 {
 cout<<"Color is "<<iColor<<endl;
 cout<<"Width is "<<iWidth<<endl;
 cout<<"Height is "<<iHeight<<endl;
 }
};
class bookshelf : public furniture
{
private:
 int iNo_shelves;
public:
 void accept(void)
 {
 furniture::accept();
 cout<<"\nEnter the No of Shelves\t";
 cin>>iNo_shelves;
 }
 void display(void)
 {
 furniture :: display();
 cout<<"No of Shelves are "<<iNo_shelves<<endl;
 }
};
class chair : public furniture
{
private:
 int iNo_legs;
```

```

public:
 void accept(void)
 {
 furniture::accept();
 cout<<"\nEnter the No of legs\t";
 cin>>iNo_legs;
 }

 void display(void)
 {
 furniture :: display();
 cout<<"No of Legs are "<<iNo_legs<<endl;
 }
};

void main()
{
 bookshelf BS1;
 chair C1;
 BS1.accept();
 BS1.display();
 C1.accept();
 C1.display();
}

```

The bookshelf class overrides the accept() and display() of class furniture by declaring functions with the same names. The accept() of the bookshelf class invokes the accept() of the furniture class using the scope resolution operator. The display() of the bookshelf class does the same too.

### Base Class Initialisation

If a new class is derived from a base class, the constructor(s) of the base class is (are) not inherited. Recollect that the object of the derived class includes all the data members of the base class except the constructor and destructor. Since the constructor(s) is (are) not inherited. They have to be explicitly invoked from the derived class, if the base class members to be initialised. If a base class constructor is not explicitly invoked in the derived class, the compiler invokes the default constructor of the base class.

### Example

```

#include<iostream.h>

class furniture
{
private:
 int iColor, iWidth, iHeight;
public:
 furniture() //default constructor
 {
 iColor=iWidth=iHeight=0; }
}

```

```

 furniture(int iC, int iW, int iH)//constructor with three
 //arguments
 {
 iColor = iC
 iWidth = iW
 iHeight = iH;
 }
 void display(void)
 {
 cout<<"Color is "<<iColor<<endl;
 cout<<"Width is "<<iWidth<<endl;
 cout<<"Height is "<<iHeight<<endl;
 }
};
class bookshelf : public furniture
{
private:
 int iNo_shelves;
public:
 //Calling the base class constructor using the colon
 bookshelf(int iS, int iC, int iW, int iH) :
 furniture(iC, iW, iH)
 {
 iNo_shelves = iS;
 }
 void display(void)
 {
 furniture :: display();
 cout<<"No of Shelves are "<<iNo_shelves<<endl;
 }
};
int main()
{
 bookshelf B(5,4,3,2);
 B.display();
 return 0;
}

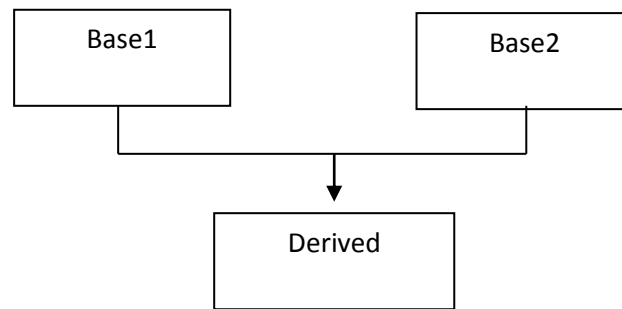
```

In the above program, as the bookshelf class is inherited from the furniture class, an object of the bookshelf class includes all the members to the furniture class. So, the bookshelf class constructor must initialise not only its member, but also the base class members. To initialise the members of furniture class, the constructor of the furniture class is called before the bookshelf class data members are initialised.

Note: When an object of the derived class is created, the base class constructor is executed before the derived class constructor. Destructors are executed in the reverse order.

## Multiple Inheritance

Multiple Inheritance is the concept where a subclass inherits properties from multiple base classes.



If the class derived, inherits from both classes base1 and base2, then the syntax is:

```
class Derived: public Base1, public Base2
{
//Body
};
```

Example

```
//multi_inhert.cpp
#include<iostream.h>
class Base1
{
 protected:
 int iVar1;
 public:
 void show1(void)
 {
 cout<<iVar1<<endl;
 }
};
class Base2
{
 protected:
 int iVar2;
 public:
 void show2(void)
 {
 cout<<iVar2<<endl;
 }
};
//Inherit multiple base classes
class derived : public base1, public base2
{
public:
 void set(int iTx , int iTy)
 {
 iVar1=iTx;
 iVar2=iTy;
 }
};
```

```

int main()
{
 derived dvar;
 dvar.set(10,20);
 dvar.show1();
 dvar.show2();
 return 0;
}

```

### Ambiguities in Multiple Inheritance

When a class inherits from multiple base classes, a whole lot of ambiguities creep in like what happens when two base classes contain a function of the same name.

#### Example

```

class base1
{
public:
void disp(void)
{
 cout<<"Base1"<<endl;
}
};
class base2
{
public:
void disp(void)
{
 cout<<"Base2"<<endl;
}
};
class derived : public base1, public base2
{
 //empty class
};
int main()
{
 derived dvar;
 dvar.disp(); //Ambiguous
 return 0;
}

```

Here, the reference to disp() is ambiguous because the compiler does not know whether disp() refers to the member in class Base1 or Base2. This ambiguity can be resolved using the scope resolution operator.

```

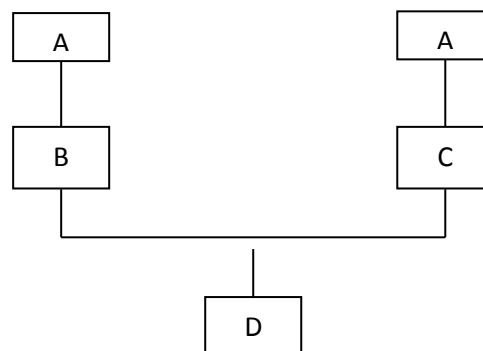
int main()
{
 derived dvar;
 dvar.base1::disp();
 dvar.base2::disp();
 return 0; }

```

This ambiguity can also be resolved by overriding; that is, the members can be redefined in the derived class.

```
class derived : public base1, public base2
{
void disp(void)
 {
 base1::disp();
 base2::disp();
 cout<<"Derived Class";
 }
};
```

Another ambiguity that arises in multiple inheritance is the possibility of the derived class having multiple copies of the same base class.



Class D inherits from two base classes, namely B and C. B and C in turn are derived from class A. As a result class D would have two copies of Class A.

### Example

```
class A
{
public:
 int Avar;
};
class A
{
public:
 int Avar;
};
class B: public A
{
public:
 int Bvar;
};
class C: public A
{
public:
 int Cvar;
};
```



```

class D:public B, public C
{
public:
 int Dvar;
};
int main()
{
 D d;
 d.Avar=10; //This is ambiguous.
 return 0;
}

```

On compiling, the above program would flag an error because the variable reference Avar in Dtemp is ambiguous. This ambiguity sets in because class D inherits one copy of Avar through class B and another through class C.

There are two ways of resolving this ambiguity.

```

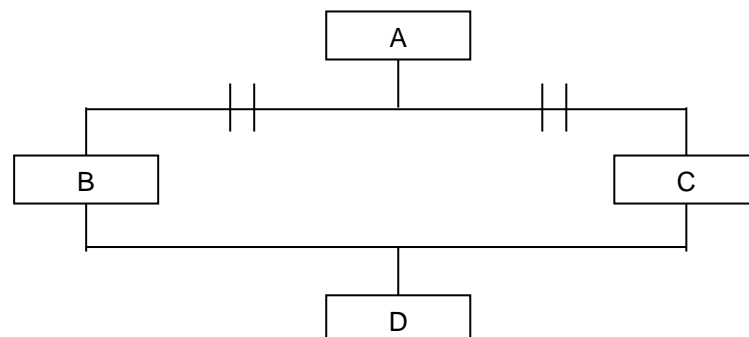
int main()
{
 D d;
 d.B::Avar=10;
 d.C::Avar=10;
 return 0;
}

```

The other way is to use virtual base classes.

### Virtual Base class

The principle behind virtual base class is, when the same class is inherited more than once via multiple paths, multiple copies of the base class members are created in memory. By declaring the class inheritance as virtual, only one copy of the base class is inherited. A base class inheritance can be specified as virtual by writing the virtual qualifier.



The class definitions are modified as follows:

```
class A
{...};
class B: virtual public A
{...};
class C: virtual public A
{...};
class D: virtual public B, public C
{...};
int main()
{
 D d;
 d.Avar=10; //No ambiguity, as only one copy of Avar exists.
 return 0;
}
```

In the above-mentioned example, members of class A are inherited by class B and C using the virtual keyword. Class A is a virtual base class. Class D inherits only one copy of member data Avar from class. As a result any reference to member variable Avar through an object of class D does not lead to any ambiguity.

### *Invocation of Constructor and Destructor*

Constructors are invoked in the following order.

- Virtual base class constructor – in order of inheritance
- Non-Virtual base class constructor – in order of inheritance
- Member object constructors – in order of declaration
- Derived class constructor

Destructors are invoked in the reverse order.

#### Example:

```
#include<iostream.h>
class C1
{
public:
 C1()
 {
 cout<<"Constructor C1 is invoked\n";
 }
 ~C1()
 {
 cout<<"Destructor C1 is invoked\n";
 }
};
```

```
class C2
{
public:
 C2()
 {
 cout<<"Constructor C2 is invoked\n";
 }

 ~C2()
 {
 cout<<"Destructor C2 is invoked\n";
 }
};

class C3 : virtual public C1
{
public:
 C3()
 {
 cout<<"Constructor C3 is invoked\n";
 }
 ~C3()
 {
 cout<<"Destructor C3 is invoked\n";
 }
};

class C4 : virtual public C1
{
public:
 C4()
 {
 cout<<"Constructor C4 is invoked\n";
 }
 ~C4()
 {
 cout<<"Destructor C4 is invoked\n";
 }
};

class C5
{
public:
 C5()
 {
 cout<<"Constructor C5 is invoked\n";
 }
 ~C5()
 {
 cout<<"Destructor C5 is invoked\n";
 }
};
```

```

class C6 : public C2, public C3, public C4
{
private:
 C5 Evar;
public:
 C6()
 {
 cout<<"Constructor C6 is invoked\n";
 }

 ~C6()
 {
 cout<<"Destructor C6 is invoked\n";
 }
};

int main()
{
 C6 fvar;
 cout<<"Program Over \n";
 return 0;
}

```

The output is:

```

Constructor C1 is invoked
Constructor C2 is invoked
Constructor C3 is invoked
Constructor C4 is invoked
Constructor C5 is invoked
Constructor C6 is invoked
Destructor C6 is invoked
Destructor C5 is invoked
Destructor C4 is invoked
Destructor C3 is invoked
Destructor C2 is invoked
Destructor C1 is invoked

```

### Practice:

1. Create an abstract class called shape. Derive classes called rectangle and circle. Incorporate the functions `Get_dimension()` and `find_area()`.
2. Imagine a publishing company that markets both book and audiocassette versions of its books. Create a class called 'publication' that stores the title and price of a publication. From this class derive two classes: 'book', which adds a page count; and 'tape', which adds a playing time in minutes. Each of the classes should have a `getdata()` and `putdata()`.
3. In the above class, add another class 'sales' that holds an array of three floats so that it can record the sales of a particular publication for the last three months. Include a `getdata()` function to get three sales amounts from the user, and a `putdata()` to display the sales figures. Alter the book and tape classes so that they are derived from both publication and sales. An object of class book or tape should input or output sales along with its other data.

-----

# Chapter – XV: Binding

---

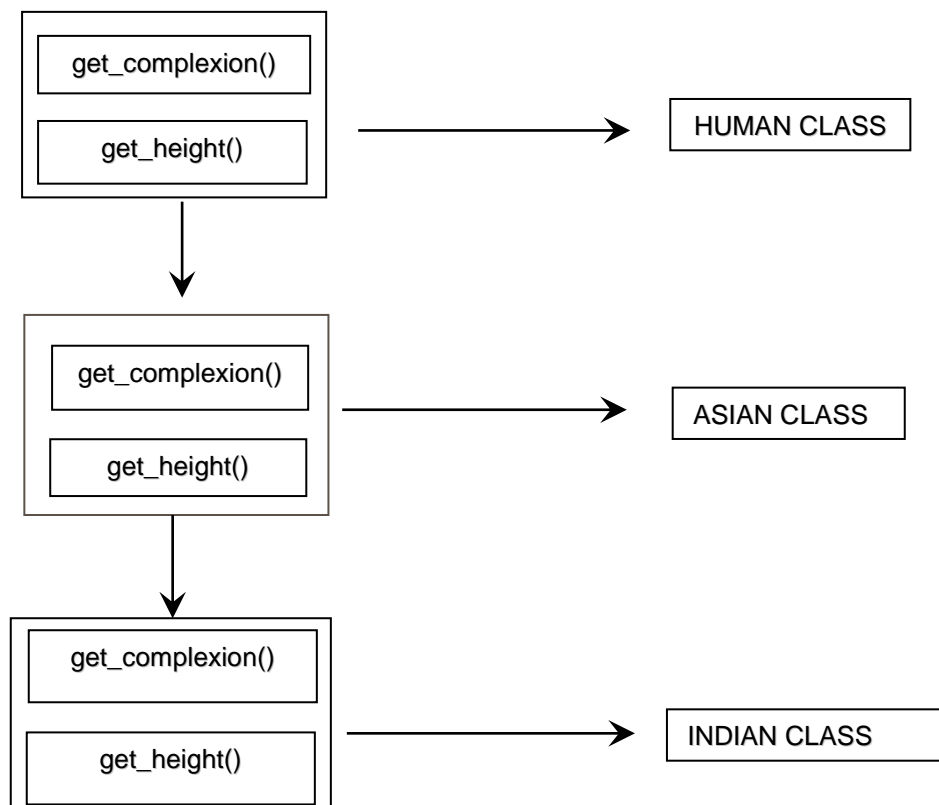
## Binding

Object oriented programming implements the concept of generalization through inheritance. A related features offered by C++ is *late binding*.

This session focuses on this feature and how it can be used to make the code less complex and easier to maintain.

## Concept of Binding

Consider a class hierarchy to represent the family of personal computers. The HUMAN class can be the base class from which ASIAN class is derived which in turn is the base class for the INDIAN class. The following figure represents the class hierarchy.



Consider the following statements:

```
ASIAN Sachin;
```

```
Sachin.get_height();
```

An object of the ASIAN class named Sachin is created and a member function called `get_height()` is invoked. To invoke the function `get_height()`, the compiler should know to which class this function

belongs. In this case there is no ambiguity because Sachin is the object of the ASIAN class. The compiler will know that the function `get_height()` of the ASIAN class is being referred.

Any member function of a class can be invoked from the instance of that class. In the following statements:

```
HUMAN *Ptr;
```

```
...
```

```
Ptr->get_height();
```

The compiler needs to associate the `get_height()` function with a particular class. Here again, there is no ambiguity. The compiler associates the function with the HUMAN class, based on the type of the pointer `ptr`. Thus the compiler associates the function with the class by identifying the type of the object or pointer that is used to invoke the function. *Process of associating a function to an object is called binding.*

Note: In C++, there exists a special relationship between the base class and the derived class. A pointer to a base class objects can point to an objects of any of the derived classes. For example, a pointer to the HUMAN class can point to an object of the HUMAN, ASIAN, INDIAN class. Thus, the following assignments are valid:

```
HUMAN * Ptr;
```

```
Ptr = new ASIAN;
```

```
Ptr = new INDIAN;
```

### Virtual Functions:

#### *Implementing late binding using virtual functions*

While pointers are necessary for late binding, they are not sufficient to implement it. Consider the following statement:

```
HUMAN*ptr;
```

```
Ptr=new ASIAN
```

```
Ptr->get_height();
```

The compiler always invokes the member function of the HUMAN class although `ptr` is pointing to an object of the ASIAN class. This happens because; *the binding takes place at the time of compilation, when the contents of the pointer are not known.* Thus, using base class pointers alone is not sufficient to implement late binding.

To ensure late binding, the concept of virtual function needs to be understood. *A virtual function is a function that is declared as virtual in a base class and is redefined by a derived class.* To declare a function as virtual, its declaration is preceded by the keyword `virtual`. A function is declared virtual because its execution depends on the context (contents of the base class pointer used to invoke it), which is not known at the time of compilation.

Late binding is ensured in the HUMAN class by declaring the function of the base class as virtual. The following code segment contains the modified declaration of the function `get_height()` making the function virtual.

```
class HUMAN
{
 .
 virtual int. get_height();
 .
};
class ASIAN: public HUMAN
{
 .
 int. get_height();
 .
};
class INDIAN: public ASIAN
{
 ...
 int. get_height();
 ...
};
```

The `get_height()` member function of the HUMAN class is a virtual function. *When a virtual function is inherited, its virtual nature is also inherited.* In the above code segment, the function `get_height()` in ASIAN and INDIAN classes are virtual.

### Pure Virtual Function:

A pure virtual function is a virtual function with no body. Often the virtual functions of the base class are not used. For example, assume that the HUMAN class you have seen earlier, is derived from a microcomputer class. If the microcomputer class also has a `get_height()` function, it will never be invoked. This is because the microcomputer class is too generalized to have a definite RAM size. Only the functions in the derived class will be invoked. In such situations, the body of the virtual function can be removed.

*A pure virtual function can be declared by equating it to zero.* For example, the statement

```
virtual int. get_height()=0;
```

declares the function `get_height()` to be a pure virtual function.



## Abstract Class

An abstract class is one that contains at least one pure virtual function. An abstract class is used as a base class for deriving specific classes of the same kind. It defines properties common to other classes derived from it. Since an abstract class contains one or more functions for which there is no definition, no objects can be created using an abstract class. However, you can create pointers to an abstract class.

This allows an abstract class to be used as a base class, pointers to which can be used to select the proper virtual function.

## Void Pointer:

Pointers are used to point to different data types. A char type pointer points to character variables, int type pointer points to integer variables, and so on. C++ provides a general-purpose pointer that can point to any data type. This is called the void pointer.

```
//void .cc

//This programs illustrates the use of a void pointer

#include<stream.h>

void main()

{

 int iNum1;

 float fNum1;

 int *iptr; //pointer to int

 float *fptr; //pointer to float

 void *vptr; //pointer to void

 iPtr=&iNum1;

 //iPtr=&fNum1; //error, int pointer pointing to float

 //fptr=&iNum1; // error, float pointer pointing to int

 vPtr=&iNum1; //ok, void pointer pointing to int

 vPtr=&fNum1; //ok, void pointer pointing to float

}
```

## Practice:

1. Which of the following statements are valid?

- a. `HUMAN *ptr_to_any_Human;`  
`ptr_to_any_Human = new HUMAN;`  
`ptr_to_any_Human -> get_height();`
- b. `ASIAN vectra;`  
`HUMAN any_Human;`  
`vectra = &any_Human;`  
`vectra->get_height();`
- c. `HUMAN *ptr_to_any_Human;`  
`INDIAN infinity;`  
`ptr_to_any_Human = &infinity;`  
`ptr_to_any_Human -> get_height();`

2. Create a class 'Furniture' in C++ having the attributes as colour, height, and width. It has two derived classes: 'chair' having the extra attribute as No\_of\_legs and 'bookshelf' having the extra attribute of No\_of\_shelves. Now

- a. Create an array of pointers to furniture class.
- b. Display the following menu to accept user's choice.
  - Do you want to
  - 0. Exit
  - 1. Order a chair
  - 2. Order a book shelf
- c. Depending on the choice, allow data entry for one item of furniture. Put the newly created objects into array of pointers.
- d. Repeat choice (1) and (2) until choice is (0).
- e. On entering (0), display the items stored and exit from the program.

-----

# Chapter – XVI: Polymorphism

---

## Polymorphism

The term polymorphism has been derived from two words: **poly**, which means many, and **morph**, which means forms, i.e., an object present in different physical forms. In real world programming polymorphism is broadly divided into two parts:

➤ **Static Polymorphism:** Exhibited by overloaded functions.

➤ **Dynamic Polymorphism:** Exhibited using late binding.

### Static Polymorphism

Static Polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions on the basis of number, type, and sequence of their arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to the calls at the compilation time. The form of association is called early binding. The term early binding stems from the fact that when program is executed, the calls are already bound to the appropriate functions.

The resolution is on the basis of number, type, and sequence of arguments declared for each form of the function. Example:

```
void add(int, int);

void add(float, float);
```

### Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when the program is executed. The term late binding refers to the resolution of the function to their associated methods at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context. The compiler is unable to bind a call to a method since resolution depends on the context of the call.

***Static Binding is more efficient while Dynamic Binding is more flexible.***

Statically bound methods are those, which are bound to their calls at compilation time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statistically bound functions.

## Friend Functions

The concept of data hiding and encapsulation dictates that non-member functions should not be able to access an objects private or protected data. However, there are situations where such rigid discrimination leads to considerable inconvenience. In order to access the non-public members of a class, C++ provides the friend facility. For example, in the following class which represents the metric unit for measuring distance:

```
class mks_distance
{
 private:
 int iMeter;
 int iCm;

 public:
 mks_distance(int,int); //Constructor
 void display_distance(void);
 //Function to display distance in meters
};

mks_distance :: mks_distance(int iMt=0,int iCmt=0)
{
 iMeter=iMt;
 iCm=iCmt;
}

void mks_distance :: display_distance(void)
{
 cout<<"The distance = "<<iMeter<<". "<<iCm<<"meters"<<endl;
}
```

A function compare() is to be written that accepts two objects of mks\_distance type, and returns a zero if they are equal. This function needs to access the private members of the class instances in order to compare the values. The only way to access the private data is to use the public function defined within the class. If the function compare() is declared as the friend function of the class mks\_distance, the function can access the non-public member of all instances of the class. A function can be declared a friend by using the friend clause before the function declaration. The declaration of the friend function must be included within the class of which it is a friend. The modified class declaration is as follows:

```

class mks_distance
{
...
friend int compare(mks_distance &,mks_distance &);
...
}

```

The body of compare() can be declared anywhere in the program.

```

int compare(mks_distance &m1, mks_distance &m2)
{
 //Accessing the private members of objects of mks_distance
 class
 int m_diff=m1.meter-m2.meter;
 int cm_diff=m1.cm-m2.cm;
 return(m_diff*100+cm_diff);
}

```

In the above example, by declaring the function compare() as a friend of mks\_distance, compare() is allowed to access the non-public members of the instances of the class.

**Note:** *By declaring the function as a friend within a class, does not make it a member of the class.*

## Friend Class

Just as a function can be made a friend to a class, an entire class can also be made a friend to another class.

```

class fps_distance
{
...
 friend class mks_distance;
...
}

```

The class mks\_distance is the friend of the class fps\_distance. From class mks\_distance, all the members of class fps\_distance can be accessed.

The following program illustrates an entire class declared as a friend of another class.

```
#include<iostream.h>

class bclass; // Forward declaration of class bclass

class aclass;

{
private:
 int iData1;
public:
 alpha()
 {
 iData1=10;
 }
 friend class bclass;
};

class bclass
{
public:
 void func(aclass A1)
 {
 cout<<"The value of iData1 in class aclass is "<<
A1.iData1<<endl;

 // The friend function can access the private data
 }
};

void main()
{
 aclass A1;
 bclass B1;
 B1.func (A1);
}
```

In the above program, entire class bclass is a friend of aclass. Now all the member functions of bclass can access the private data of aclass.

## Operator Overloading

Operator overloading makes Abstract Data Types (ADT) more natural and closer to fundamental data types.

Most fundamental data types have pre-defined operators associated with them. For example: the C++ data type int, together with operators +, -, \*, and /, provides an implementation of the mathematical concepts of an integer. To make a user-defined data type as natural as a fundamental data type, the user-defined data type must be associated with the appropriate set of operators.

Lets consider FPS class which implements the British way of measuring distance. In order to add to distances or find the difference between them, we generally call the functions defined in the FPS class. But we can redefine the + or - operator in order to find the result distance. For example:

```
FPS f1, f2, f3;
```

```
.....
```

```
f3=f1+f2;
```

```
f3=f2-f1;
```

In order to compare, we can over load the comparison operators.

Operator overloading refers to giving additional meaning to the normal C++ operators when they are applied to ADTs.

Only the pre-defined set of C++ operators can be over loaded. An operator can be overloaded by defining a function for it.

The function for the operator is declared using the operator key word.

```
int fps_distance::operator == (fps_distance);
```

```
.....
```

```
.....
```

```
int operator == (fps_distance &Fps2)

{

float fTemp1 = iFeet * 12 + inch;

float fTemp2 = Fps2.iFeet + Fps2.finch;

return ((fTemp1 == fTemp2) ? 1:0);

}
```



## Practice:

1. Create a word class and overload the following operators: ==, >, < and +=. Give complete class definition.
2. Create a class called time that has
  - a. Separate int member data for hour, minute and seconds.
  - b. A constructor that initialises the member data to zero.
  - c. A constructor that initialises the member data to a fixed value.
  - d. A member function called display() that displays the time in hh:mm:ss format.
  - e. An overloaded + operator to add two times.
3. Create a FPS class and overload the following operators: ==, > and <. Give complete class definition.

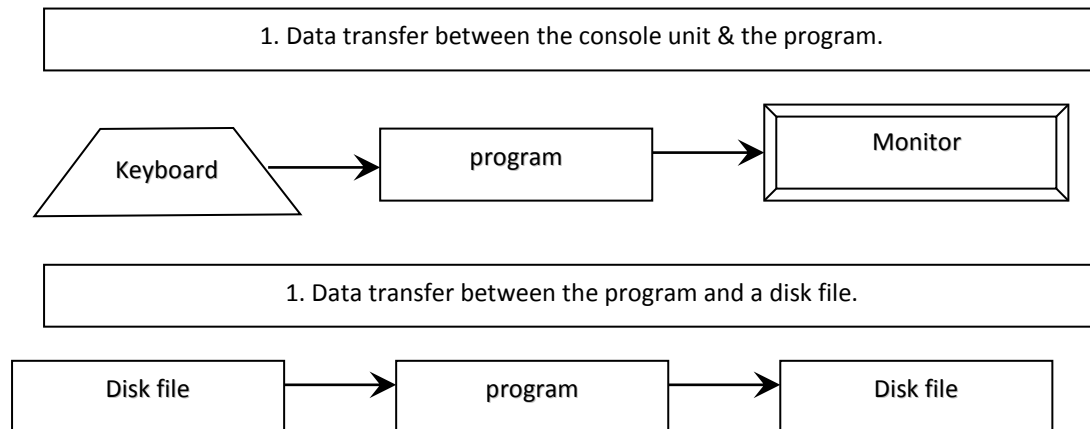
-----

# Chapter – XVI: File Handling

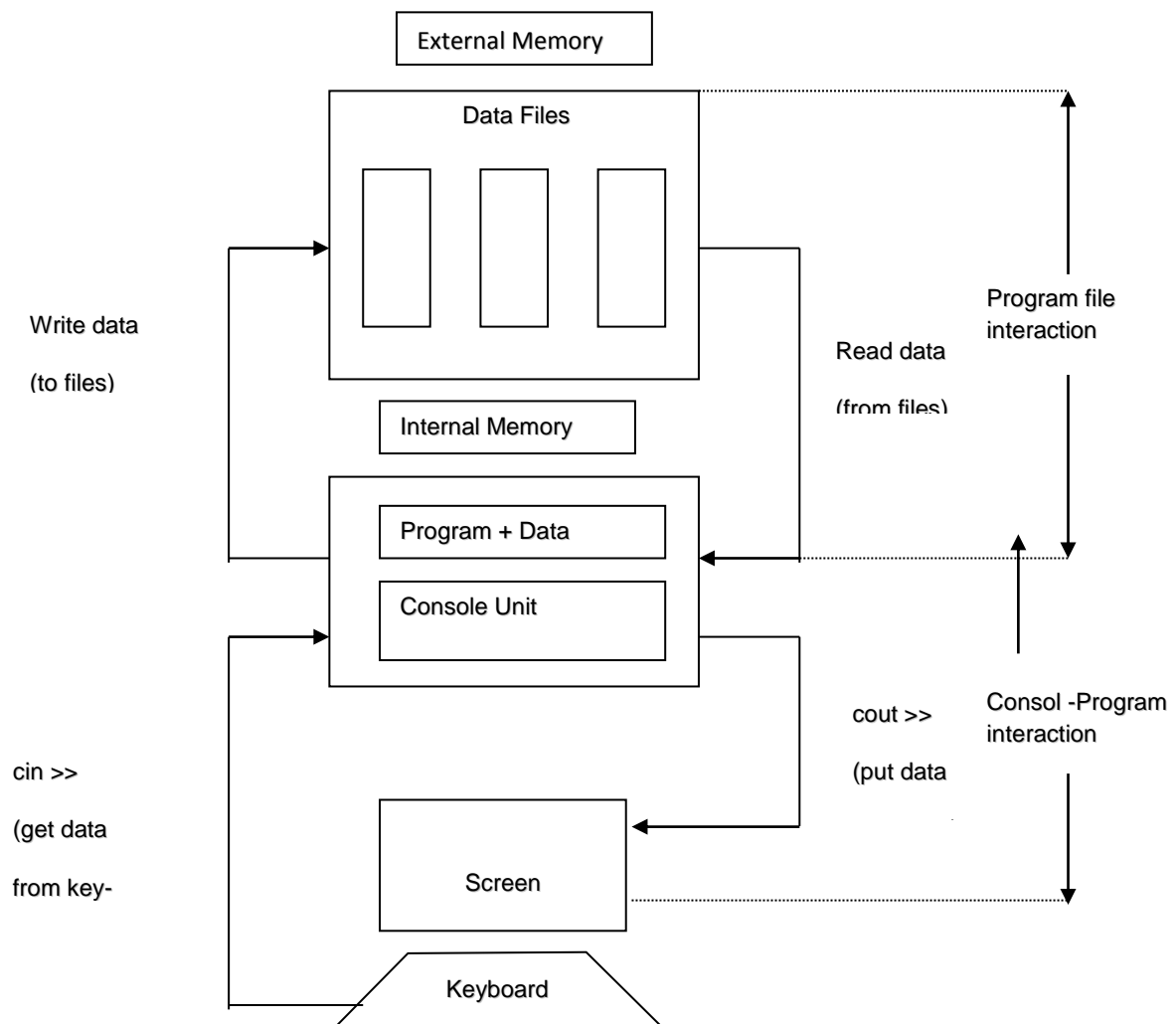
## Data transmission in the program

A program typically involves either of the following data transmission:

Console unit

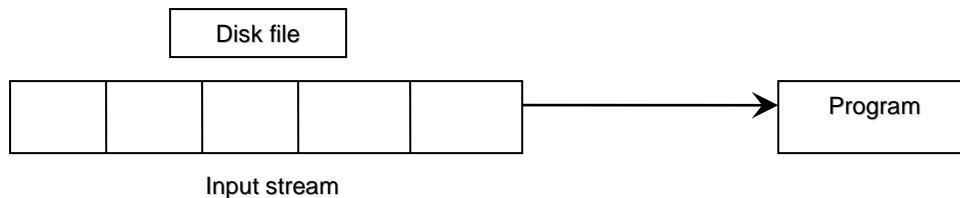


## Console program file interaction

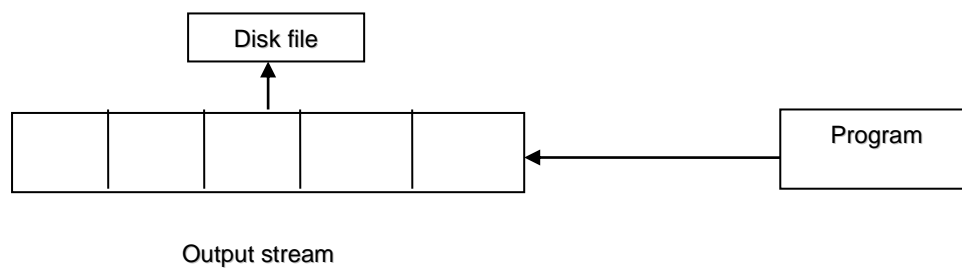


The I/O system of C++ uses file streams as an interface between the programs and the files.

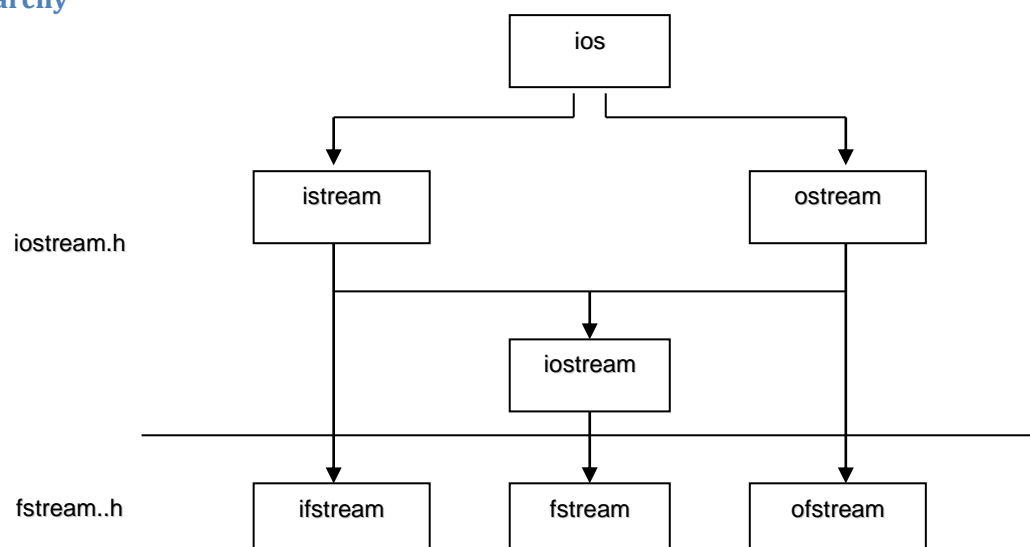
The stream that supplies data to the program is known as input stream. The input stream extracts (reads) data from the file, which involves the creation of an input stream and linking it with the program and the input file.



The stream that receives data from the program is known as output stream. The output stream inserts (writes) data to the file, which involves establishing an output stream with the necessary links with the program, and the output file.



## Class Hierarchy



Stream classes for file operations (contained in fstream.h file)

## Details of File stream Classes

| Class    | Content                                                                                                                                                                               |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ifstream | Provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() function from istream.                       |
| ofstream | Provides input operations. Contains open() with default output mode. Inherits put(), seekp(), tellp(), and write() functions from ostream.                                            |
| Fstream  | Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream. |

Using disk file the following things are to be decided about the file & its intended use:

- Suitable name for the file : employee.dat
- Data type and structure : employee
- Purpose : salary
- Opening method : Input

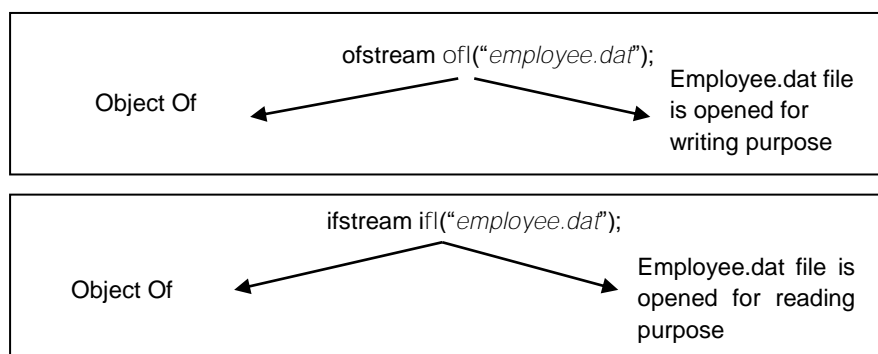
## Opening a file:

A file can be opened in two ways:

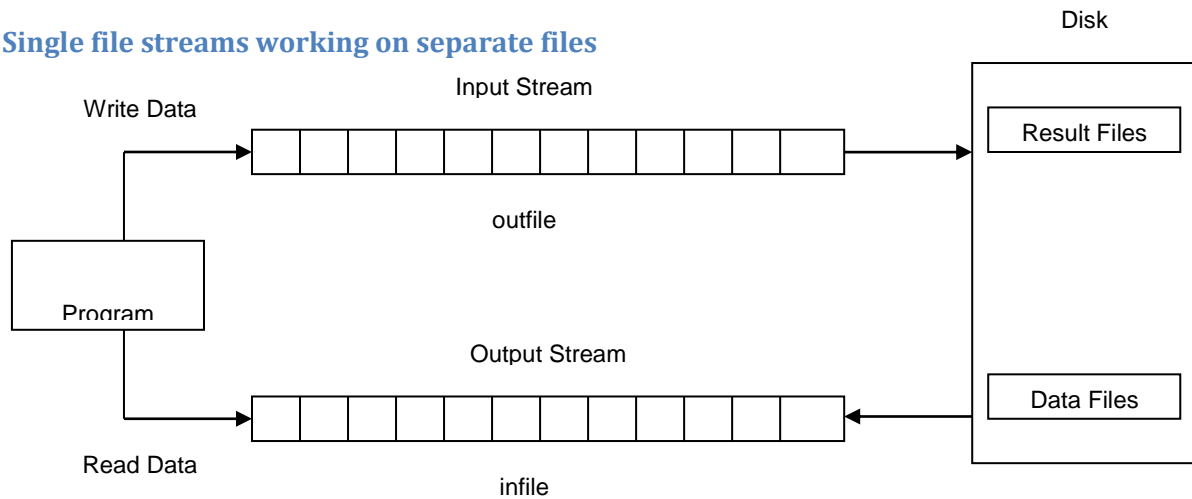
- Using the constructor function of the class
- Using the member function open of the class.

### Opening a File using Constructor:

- Creates a file stream object to manage the stream using appropriate class. That is, the class ofstream is used to create the output stream and the class ifstream to create the input stream.
- Initialise the file object with a desired file name.



## Single file streams working on separate files



## Sample Coding

```
// Example1.cpp

// File input with integers
include <fstream.h>

void main()
{
 ofstream ofil("marks.dat")
 ofil<<85<<" "<<92<<" "<<89;
}
```

In the above program an object called ofil of class ofstream is created and associated with a file. The filename is given within quotes and parentheses. The << operator is used to write the data onto ofil. The contents of the file INT.TXT when typed would be:

85 92 89

The following program reads the contents of the INT.TXT:

```
// Example2.cpp
// File output with integers
include<fstream.h>

void main()
{
 ifstream ifil("INT.TXT");
 int iVar1,iVar2,iVar3;
 ifil>>iVar1;
 ifil>>iVar2;
 ifil>>iVar3;
 cout<<iVar1<<" "<<iVar2<<" "<<iVar3<<endl;
}
```

The above program opens the file INT.TXT in the input mode using an object called ifil of class ifstream. The >> operator is used to read data from ifil into the three variables iVar1, iVar2 and iVar3. The values of the three variables are then displayed using cout.

The output program will be:

85 92 89

The reason behind using a space at the time of insertion is that, in the case of stream extraction, the integer extractor reads all character that is not of integer type. If spaces are not used, the file INT.TXT would contain the string 859289. Given this, if we read the file using

```
ifil>>iVar1;
```

```
ifil>>iVar2;
```

```
ifil>>iVar3;
```

Then iVar1 would contain 859289, and iVar2 and iVar3 would contain nothing.

The following program deals with string input:

```
//example3.cpp
//File input with strings
#include<fstream.h>
void main()
{
 ofstream Sfil("STRING.TXT");
 Sfil<<"Taming of the Shrew";
}
```

The file STRING.TXT will have the following contents:

Taming of the Shrew

The program for reading the file STRING.TXT would be as follows:

```
//example4.cc
// File output with strings
include<fstream.h>
void main()
{
 ifstream sfil("STRING.TXT");
 char cStr[20];
```

```

 sfil>>cStr;
 cout<<cStr<<endl;
 }

```

The output of the program is: Taming

As you can see, the problems of character stream extraction remain even during file input and output. The character extractor does not recognise white spaces.

### File input and Output Using Abstract Data type

A class called bill is being used for discussing file input and output using objects of user-defined classes. The class definition is as follows:

```

class bill
{
public:
 int iBill_no;
 float fBill_amt;
};

```

The following program illustrates the output of objects to a file:

```

//Example5.cpp
//Writing objects to a file
#include<fstream.h>
class bill
{
public:
 int iBill_no;
 float fBill_amt;
};
void main()
{
 ofstream Bfil("billfile.dat");
 bill Bvar;
 Bvar.iBill_no=1;
 Bvar.fBill_amt=100.00;
 Bfil<<Bvar.iBill_no<<" "<<Bvar.fBill_amt;
}

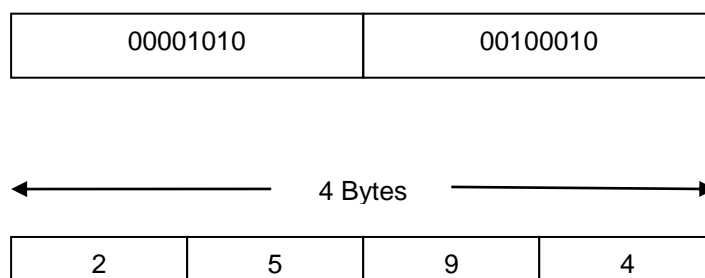
```

The following program illustrates the input of objects from a file:

```
//file6.cpp
//Reading objects from a file
#include<fstream.h>
class bill
{
public:
 int iBill_no;
 float fbill_amt;
};
void main()
{
 fstream Bfil("billfile.dat");
 bill bvar;
 bfil>>bvar.iBill_no>>bvar.fBill_amt;
 cout<<"Bill number is"<<bvar.iBill_no<<endl;
 cout<<"Bill amount is"<<bvar.fBill_amt<<endl;
}
```

## Binary Input and Output

Binary input and output means that the number 2594 will be written as integer representation taking up 2 bytes of space. When reading text files using the >> operator certain character translations occur. For example



## Character input and output

The put() & get() functions, which are members of ostream and istream, respectively, are used to output and input a single character at a time. The following program writes a string, one character at a time to a file:

```
//Example7.cpp
//Writing characters using binary input-output functions
#include<fstream.h>
#include<string.h>
```



```


char str[]="Hello World";
void main()
{
 ofstream ofil("myfile.dat");
 int ivar;
 for(ivar=0;ivar<strlen(str);ivar++)
 {
 ofil.put(str[ivar]);
 }
 ofil.put('\0');
}

```

The following program reads the string from the file character by character using the binary function `get()`.

//example8.cpp

```

/*Program to read character by character from a file using binary input &output*/
#include<fstream.h>
void main()
{
 char ch;
 ifstream ifil("myfile.dat");
 while(ifl)  Checks for the end-of-the-file
 {
 ifil.get(ch);
 cout<<ch;
 }
}

```

### String Input:

`get()` also has some variations.

**`get(char *str, int len, char delim='\n')`**

1. Fetches characters from the input stream into array `str`.
2. Fetching is stopped if `len` characters have been fetched
3. Fetching is also stopped if the `delim` character is encountered.
4. The terminating character is not extracted.

```
//example9.cpp
#include<fstream.h>
void main()
{
 char cword[20];
 cin.get(cword,19);
 cout<<cword;
}
getline(char *str, int len, chat delim='\n')
```

1. Is similar to get().
2. Extracts the terminator also.

```
//example10.cpp
#include<fstream.h>
void main()
{
 char cword[20];
 cin.getline(cword,20);
 cout<<cword;
}
```

### Integer Input:

While get() & put() can be used for characters, for numeric values, the following member functions are available.

1. read()
2. write()

The read() & write() function has to be called with two arguments.

- 1.The address of the bufer into which the data to be read.
- 2.The size of the buffer.

```
ifl.read((char *)&v,sizeof(v));
```

```
ofil.write((char *)&v,sizeof(v));
```

```

//exampl11.cpp
#include<fstream.h>
void main()
{
 ofstream ofil("marks.dat");
 int inum=0;
 cin>>inum;
 ofil.write((char *)&inum, sizeof(int));
}

//exampl12.cpp
#include<fstream.h>
void main()
{
 ifstream ifil("marks.dat");
 int inum=0;
 ifil.read((char *)&inum, sizeof(int));
 while(ifil)
 {
 cout<<inum;
 ifil.write((char *)&inum, sizeof(inum));
 }
}

```

## Open function

|                                      |        |                                  |
|--------------------------------------|--------|----------------------------------|
| <code>ifstream ifil;</code>          | —————→ | Creates an unopened input stream |
| <code>Ifil.open("marks.dat");</code> | —————→ | Associates the stream to a file  |

## Close Function

|                                      |        |                                  |
|--------------------------------------|--------|----------------------------------|
| <code>ofstream ofil;</code>          | —————→ | Creates an unopened input stream |
| <code>ofil.open("marks.dat");</code> | —————→ | Associates the stream to a file  |
| <code>ofil.close();</code>           | —————→ | Closes the file                  |

## File Pointers and Manipulators

| Mode      | Explanation                                                                     |
|-----------|---------------------------------------------------------------------------------|
| App       | Start reading or writing at the end                                             |
| In        | Open for reading                                                                |
| Out       | Open for writing                                                                |
| Ate       | Seek the end-of-the-file                                                        |
| Nocreate  | Error when opening if file does not already exist.                              |
| Noreplace | Error when opening for output if file already exists, unless app or ate is set. |

### Usage:

```
Ofstream ofil("marks.dat",ios::app);
```

## Random Access

When the user manipulates the file pointers himself so that data can be read from, and written to, an arbitrary location of a file.

The seekg() & tellg() function allows to set and examine the get pointer, and the seekp() and tellp() functions perform the same actions on the put pointer.

The tellg() or tellp() can be used to find out the current position of the file pointer in the file. The seekg() member function takes two arguments.

### Example:

|                                       |   |                                                                  |
|---------------------------------------|---|------------------------------------------------------------------|
| <code>fil.seekg(10,ios::beg);</code>  | → | Position the get pointer 10 bytes from the beginning of the file |
| <code>fil.seekg(10,ios::cur);</code>  | → | Position the get pointer 10 bytes                                |
| <code>fil.seekg(-10,ios::cur);</code> | → | Position the get pointer 10 bytes back from the current position |
| <code>fil.seekg(-10,ios::end);</code> | → | Position the get pointer 10 bytes back from the end of the file  |

The tellg() member function doesnot have any arguments. It returns the current byte position of the get pointer related to the beginning of the file.


### Example:

```
int iposition = fil.tellg();
```

```
seekp()
```

Put pointer. Defined in ostream class.


```
tellp()
```



```
seekg()
```

Get pointer. Defined in istream class.

```
tellg()
```



```
//examp113.cpp
#include<fstream.h>
class bill
{
public:
 int iBill_no;
 float fBill_amt;
};
void main()
{
 bill Bvar;
 fstream fil("BILL.DAT",ios::in);

 fil.seekg(0,ios::end);
 int iend;
 iend-=tellg();
 cout<<"the size of the file is"<<iend<<endl;
 cout<<"Size of one record is"<<sizeof(bill);
 int norec=iend/sizeof(bill);
 cout<<"There are "<<norec<<"records in the file"<<endl;
}
```

## Practice

1. Happy journey is a committed tour and travel company. It has devised many innovative packages for its customers who wants to take a holiday. There are three kinds of tours:

- a. Discover India
- b. Honeymoon Bash
- c. Pilgrimage Package

These tours start everyweek. A customer can avail a package belonging to any category, starting on any given date. The customer can also specify the number of people accompanying the customer.

The following information are recorded regarding the customer :

- a. Customer name
- b. No. of people accompanying
- c. Package Category (D/H/P)
- d. Cost
- e. Tour start date

Store these details about the customers in a file and read/write data from the file. The following things the automated customer information package should support.

- a. Add Data
  - b. Display Data
  - c. Query Data
  - d. Exit
2. Write a program that emulates the DOS copy command, i.e., it should copy the contents of one file to another file. Invoke the program with two command line arguments – the source file and the destination file. Example:

```
$mycpy srcfile.cpp destfile.cpp
```

In the program, check whether the user has typed the correct number of arguments, and the files specified can be opened. Display an error message if the destination file already exists.

3. Write a program that reutns the size in bytes of a program entered on the command line.

Example:               \$size demo.cpp

must return the size of the file demo.cpp.

-----