PERIYAR INSTITUTE OF DISTANCE EDUCATION (PRIDE)

PERIYAR UNIVERSITY

SALEM - 636 011.

B.Sc. COMPUTER SCIENCE

SECOND YEAR

PAPER – IV : OBJECT ORIENTED PROGRAMMING WITH C++

Prepared by :
M. TAMIZHCHELVAN, M.C.A., M.Phil.,
Lecturer, Department of Computer Science,
Vivekanandha College of
Arts and Sciences for Women,
Elayampalayam, Tiruchengode,
Namakkal (District) – 637 205.

B.Sc. COMPUTER SCIENCE

SECOND YEAR

PAPER – IV : OBJECT ORIENTED PROGRAMMING WITH C++

UNIT- I	**Principles of Object – Oriented Programming**
UNIT -II	**Beginning with C++ - Tokens, Expressions & Control Structures – Functions in C++**
UNIT -III	**Classes and Objects – Constructors and Destructors**
UNIT-IV	**Operator Overloading and Type Conversions – Inheritance : Extending Classes- Pointers, Virtual Functions & Polymorphism**
UNIT-V	**Managing Console I/O Operations – Working with Files- Templates – Exception**

B.Sc. COMPUTER SCIENCE

SECOND YEAR

PAPER – IV : OBJECT ORIENTED PROGRAMMING WITH C++

INTRODUCTION

Dear Students,

Totally this book covers five units. The first unit deals with principles of object-oriented programming, basic concepts of object oriented programming, benefits of OOP, object-oriented languages and applications of OOP.

The second unit deals with applications of C++, structure of C++ program, data types, operators, control structures and functions in C++ with examples.

The third unit deals with classes, objects, arrays, constructors and destructors with examples.

The fourth unit deals with operator overloading, type conversions inheritance, pointers, virtual functions, polymorphism with examples.

The fifth unit deals with managing console I/O operations, working with files, class templates, function templates and exception handling with examples.

PRIDE would be happy in you could make use of this learning material to enrich your knowledge and skills to serve the society.

B.Sc. COMPUTER SCIENCE

SECOND YEAR

PAPER – IV : OBJECT ORIENTED PROGRAMMING WITH C++

UNIT-I:

Principles of Object-Oriented Programming: Software Evolution – A Look at Procedure-Oriented Programming – Object Oriented Programming Paradigm – Basic Concepts of Object Oriented Programming – Benefits of OOP – Object-Oriented Languages – Applications of OOP.

UNIT-II:

Beginning with C++: What is C++ - Applications of C++ - Structure of C++ Program – A Simple C++ Program – More C++ Statements – An Example with class. Tokens, Expressions and Control Structures: Introduction – Tokens – Keywords – Identifiers and Constants – Basic Data Types –User-defined Data Types – Derived Data Types – Symbolic Constants – Type Compatibility – Declaration of Variables – Dynamic Initialization of Variables – Reference Variables – Operators in C++ - Scope Resolution Operator – Member Dereferencing Operators – Memory Management Operators – Manipulators – Type Cast Operators – Expressions and Their Types – Special Assignment Expressions – Implicit Conversions – Operator Overloading – Operator Precedence – Control Structures – Functions in C++: Introduction – The Main Function – Function Prototyping – Call By Reference – Return By Reference – Inline Functions – Default Arguments – Const Arguments – Function Overloading – Friend and Virtual Functions.

UNIT-III:

Classes and Objects: Introduction – Specifying A Class – Defining Member Functions – A C++ Program with Class – Making an Outside Function Inline – Nesting of Member Functions – Private Member Functions – Arrays within a Class – Memory Allocation For Objects – Static Data Members - Static Member Functions - Arrays Of Objects – Objects as Function arguments – Friendly Functions – Returning Objects – Const Member Functions – Pointers to Member – Local Classes. Constructors and Destructors: Introduction – Constructors - Parameterized Constructors- Multiple Constructors in a Class – Constructors with Default Arguments –Dynamic Initialization of Objects – Copy Constructor – Dynamic Constructors – Constructing Two Dimensional Arrays – Const Objects – Destructors .

UNIT – IV

Operator Overloading and Type Conversions: Introduction – Defining Operator Overloading – Overloading Unary Operators - Overloading Binary Operators -Overloading Binary Operators using Friends - Manipulation of strings using Operators –Rules for Overloading Operators – Type Conversions – Inheritance Extending Class : Introduction - Defined Derived Classes – Single Inheritance – Making A Private Inheritable - Multilevel Inheritance – Multiple Inheritance – Hierarchical Inheritance –Hybrid Inheritance – Virtual Base Classes – Abstract Classes - Constructors in Derived Classes - Member Classes : Nesting of Classes. Pointers, Virtual Functions and Polymorphism : Introduction – Pointers to Objects - This pointer – Pointer to Derived Classes – Virtual Functions – Pure Virtual Functions.

UNIT – V

Managing Console I/O Operations: Introduction – C++ Streams – C++ Stream Classes – Unformatted I/O Operations – Formatted Console I/O Operations – Managing Output with Manipulators. Working With Files: Introduction – Classes For File Stream Operations – Opening and Closing a File – Detecting End of a File – More about Open(): File Modes – File Pointers and their Manipulations - Sequential Input and Output Operations – Updating a File: Random access – Error handling During File Operations – Command Line Arguments. Templates: Introduction – Class Templates – Class Templates With Multiple Parameters – Function Templates – Function Templates with Multiple Parameters – Overloading of Template Functions – Member Function Templates Exception Handling: Introduction – Basics of Exception Handling – Exception Handling Mechanism – Throwing Mechanism – Catching Mechanism – Rethrowing An Exception – Specifying Exceptions.

TEXT BOOK:

1) “Object-Oriented Programming With C++”,

E.Balagurusamy, TMH, New Delhi, 2nd Edition

UNIT-I

1.0 **Principles of Object_ Oriented programming**

UNIT-I

1.0 Principles of Object Oriented Programming

1.1 Software Crisis

Developments in software technology continue to be dynamic. New tools and techniques are being used. The software engineers and industry continuously look for new approaches to software design and development. These rapid advances appear to have created a situation of crisis within the industry. The following issues need to be addressed to face this crisis:

- How to represent real-life entities of problems in system design?
- How to design systems with open interfaces?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are to learnt to any changes in future?
- How to improve software productivity and decrease software cost?
- How to improve the quality of software?
- How to manage time schedules?
- How to industrialize the software development process?

Many software products are either not finished, or not used, or else are delivered with major errors. Changes in user requirements have always been a major problem. In a changing world with a dynamic business environment, requests for change are unavoidable and therefore systems must be adaptable and tolerant to changes.

Software products should be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are:

1. Correctness
2. Maintainability
3. Reusability
4. Openness and interoperability
5. Portability
6. Security
7. Integrity
8. User-friendliness

1.2 Software Evolution

The evolution of software technology to the growth seems to be a tree. Like a tree, the software evolution has had distinct phases or “layers” of growth. These layers were built up one by one over the last five decades, with each layer representing an improvement over the previous one. In software systems, each of the layers continues to be functional, whereas only the uppermost layer is functional.

To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend, implement and modify.

Since the invention of the computer, many programming approaches have been tried. These include techniques such as *modular programming, top-down programming, bottom up programming and structured programming.* The primary concern is to handle the increasing complexity of programs that are reliable and maintainable.

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.3 A Look at Procedure-Oriented Programming

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming

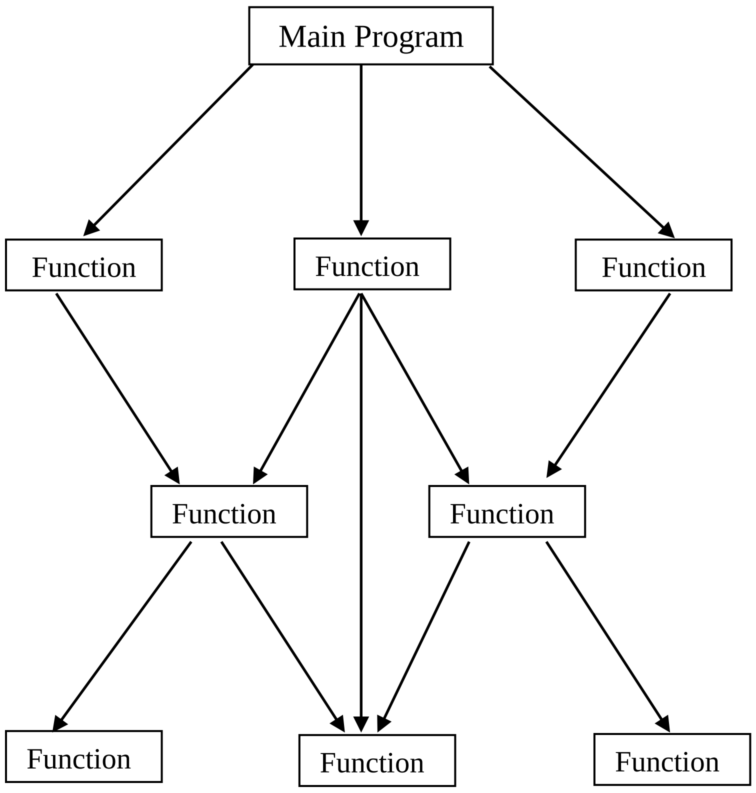


Figure. Typical structure of procedure-oriented programs.

A number of functions are written to accomplish these tasks. The primary focus is on functions. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another. While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data.

In a large program it is very difficult to identify what data is used by which function. Another serious drawback with the procedural approach is that it does not model real world problems very well.

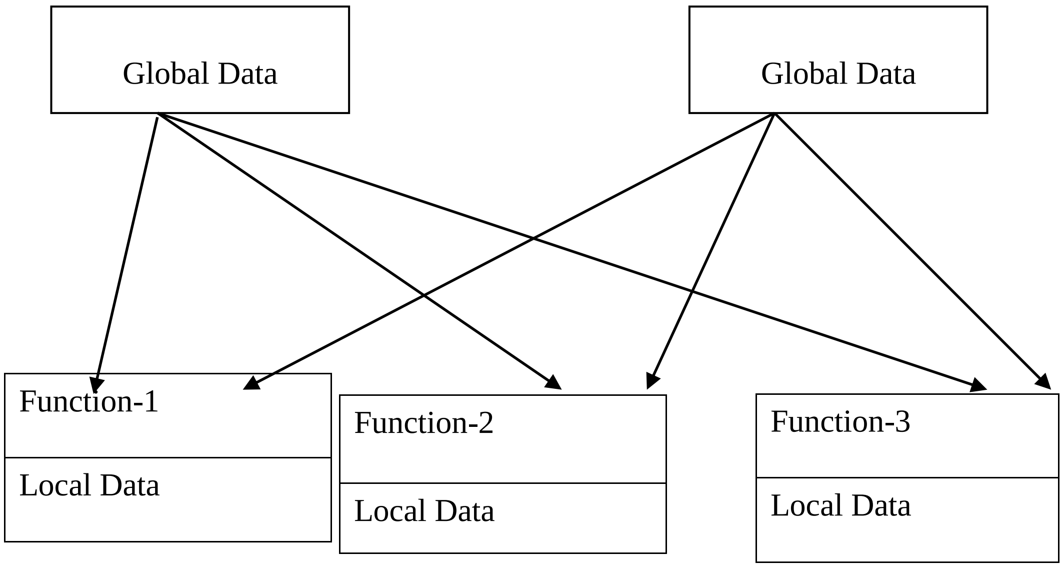


Figure. Relationship of data and functions in procedural programming

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

1.4 Object-Oriented Programming Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 1.6. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of their objects.

Some of the features of object-oriented programming are:

- Emphasis is on data rather than procedure
- Programs are divided into what are known as objects

- Data structure are designed such that they characterize the objects
- Data is hidden and cannot be accessed by the external functions
- Objects may communicate with each other through functions
- New data and functions can be easily added whenever necessary.
- Functions that operate on the data of the object are tied together in the data structure.
- Follows bottom-up approach in program design.

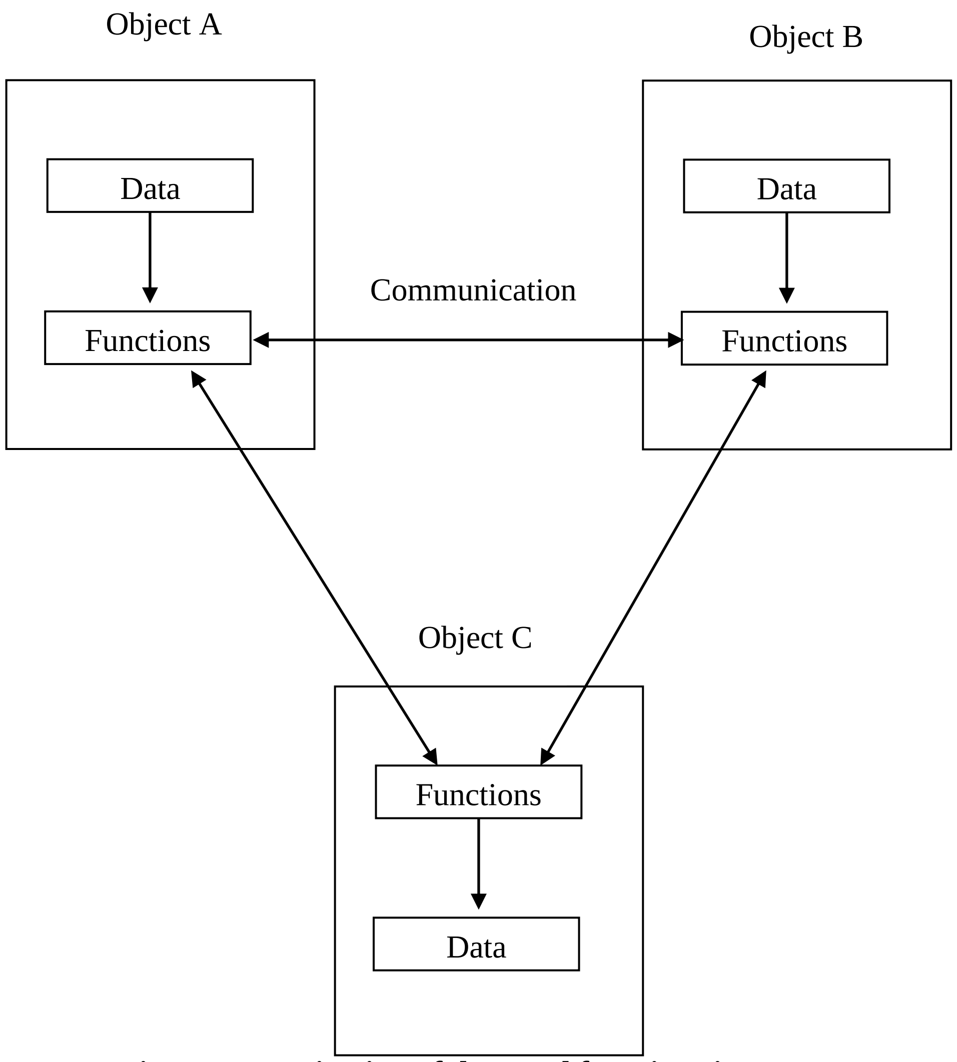


Figure. Organization of data and functions in OOP

We define "object-oriented programming as an approach that provides a way of modularizing programs by creating copies of such modules on demand." Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

1.5 Basic concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

1.5.1 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

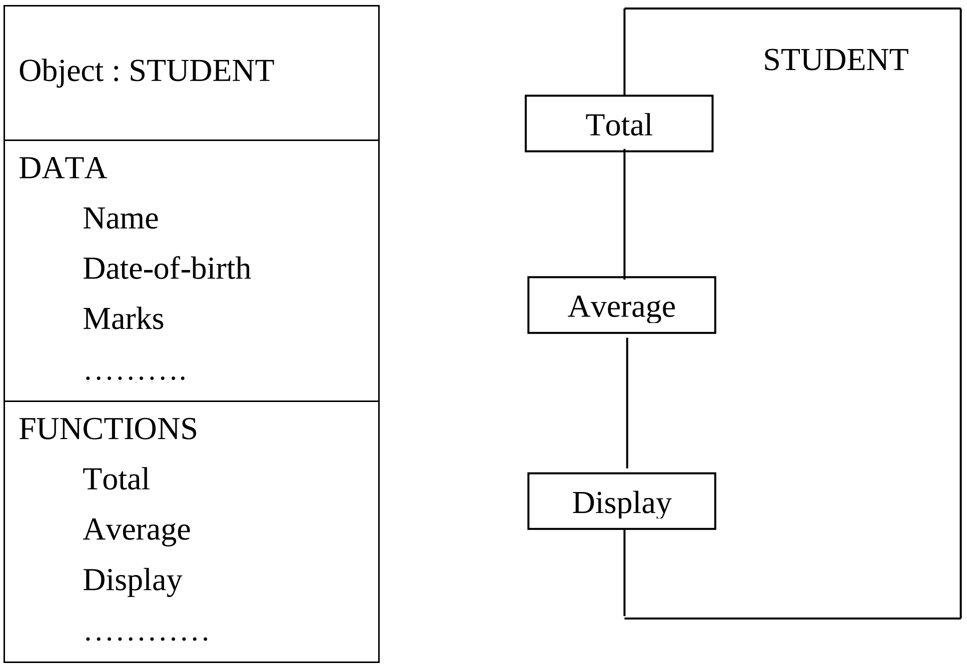
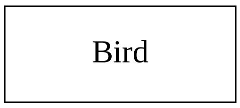


Figure. Two ways of representing an object



In OOP, a problem is considered as a collection of a number of entities called objects. Objects are instances of classes.

1.5.2 Classes

Objects contain data, and code to manipulate that data. The entire set of data of an object can be made a user defined data type with the help of a class. Once a class has been defined, we can create any number of objects belonging to that class. A class is a collection of objects of similar type. For example mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. If fruit has been defined as a class, then the statement

```
fruit  mango;
```

Will create an object **mango** belonging to the class **fruit**.

1.5.3 Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.

Abstraction refers to the act of representing essential features without including the background details. Classes use the concept of abstraction and are defined as a list of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are called *data members* because they hold information. The functions that operate on these data are called *methods or member functions*.

1.5.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of inheritance provides the idea of reusability.

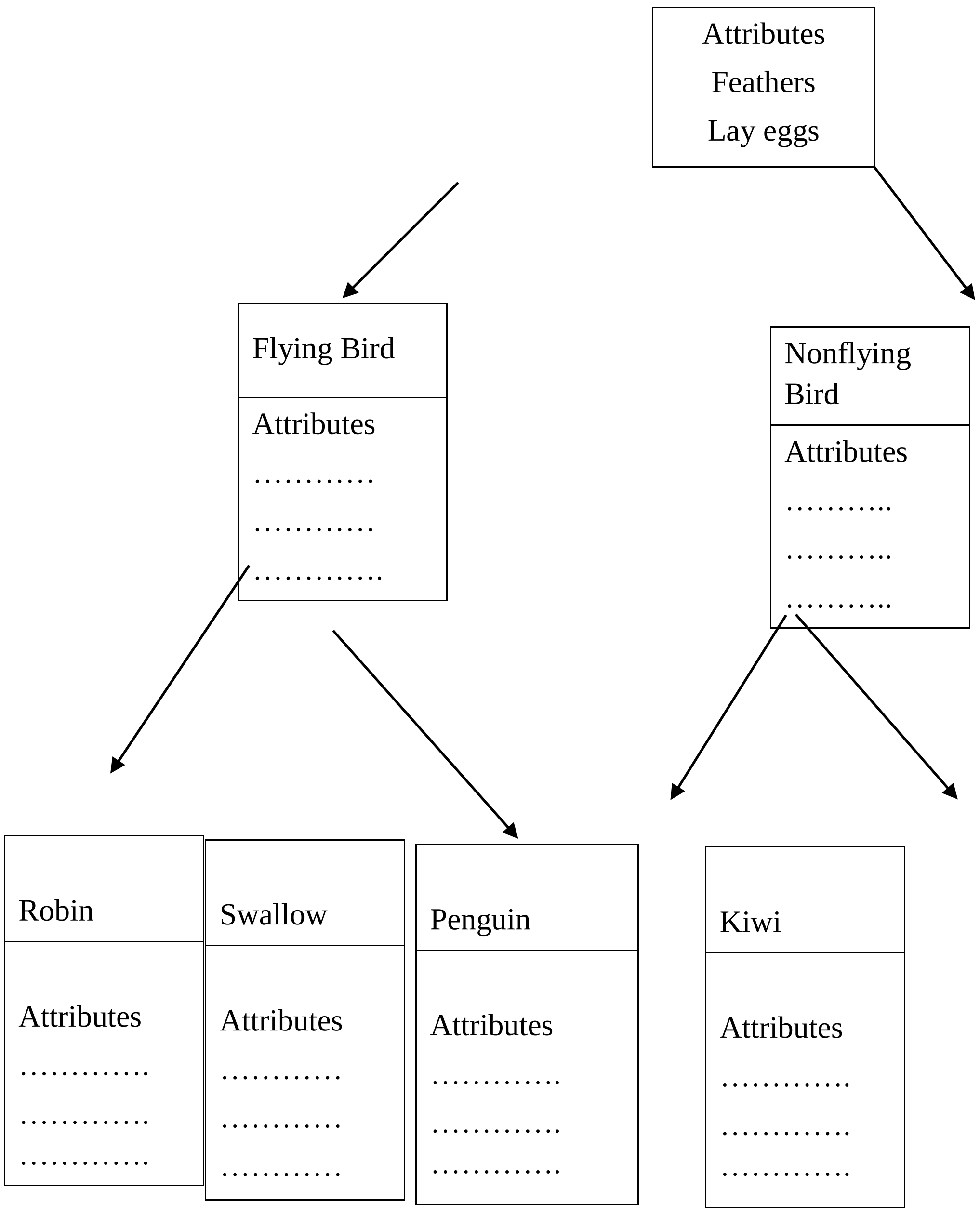


Figure. Property inheritance

This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both classes.

1.5.5 Polymorphism

Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition.

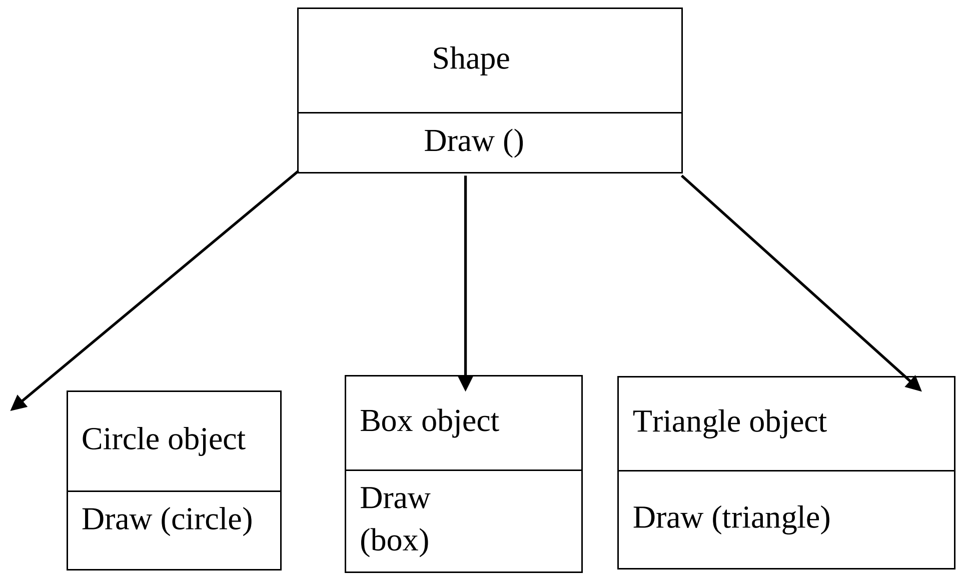


Figure. ***Polymorphism***

The process of making an operator to exhibit different behaviors in different instances is known as operator over loading. Figure shows that a single function name can be used to handle different number and different types of arguments. Using a single function name to perform different types of tasks is known as function over loading. Polymorphism is used in implementing inheritance.

1.5.6 Dynamic binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

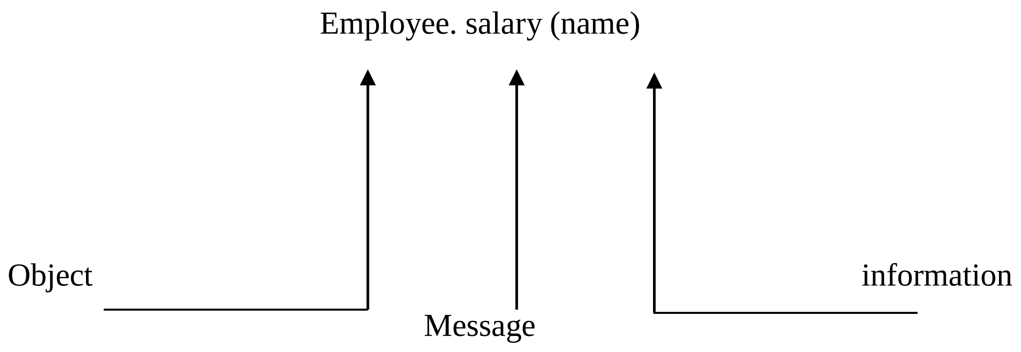
1.5.7 Message passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language involves the following basic steps:

1. Creating classes that define objects and their behavior
2. Creating objects from class definitions
3. Establishing communications among objects

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent. Example,



1.6 Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.

- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object oriented programming can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. Developing software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

1.7 Object Oriented Languages

OOP concepts can be implemented using languages such as C and Pascal. A language that is specially designed to support the OOP's concept makes it easier to implement them. The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object based programming languages, and
2. Object oriented programming languages

Major features that are required for object-based programming are:

- Data
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding.

Object-oriented programming incorporates all of object-based programming features along with two additional features such as inheritance and dynamic binding. Object-oriented programming can be characterized by

Object–based features + inheritance + dynamic binding

Languages that support these features include C++, Smalltalk, Object Pascal and java. There are a large number of object-based and object-oriented programming languages. Some popular general purpose OOP languages are

Simula, Smalltalk, Objective C, C++, Ada, Object Pascal, Turbo Pascal, Eiffel & Java.

All languages provide for polymorphism and data hiding. However, many of them do not provide facilities for concurrency, persistence ad genericity. Eiffel, Ada and C++ provide generic facility which is an important construct for supporting reuse. C++ has now become the most successful, practical, general purpose OOP language and is widely used in industry today.

1.8 Applications of OOP

Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming has been in the area of user interface design such as windows.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real time systems
- Simulation and modeling
- Object oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- NEURAL networks and parallel programming
- Decision support and office automation system
- CIM/CAM/CAD Systems

The OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

State whether the following statements are TRUE or FALSE:

(a) In procedure-oriented programming, all data are shared by all functions.

(b) The main emphasis of procedure-oriented programming is on algorithms rather than on data.

(c) One of the features of OOP is the division of programs into objects that represent real-world entities.

(d) Wrapping up of data of different types into a single unit is known as encapsulation.

(e) One problem with OOP is that once a class is created it can never be changed.

(f) Inheritance means the ability to reuse the data values of an one object by other objects.

(g) Polymorphism is extensively used in implementing inheritance.

(h) Object-oriented programs are executed much faster than conventional programs.

(i) Object-oriented systems can scale up better from small to large.

(j) Object-oriented approach cannot be used to create databases.

Questions:

1. List the principles of object-oriented programming.
2. What are the major issues facing the software industry today?
3. Briefly discuss the software evolution?
4. What is procedure-oriented programming? What are its main characteristics?
5. What is object-oriented programming? How it differs from procedure-oriented programming?
6. What are the unique advantages of an object-oriented paradigm?
7. Distinguish between the following terms:
   (a) Objects and classes
   (b) Data abstraction and data encapsulation
   (c) Inheritance and polymorphism
   (d) Dynamic binding and message passing
8. Explain the basic concepts of object-oriented programming in detail.
9. What are the benefits of OOP?
10. Write short notes on Object Oriented Languages.
11. List a few areas of application of OOP.

NOTES

NOTES

UNIT-II

UNIT-II

2.1.0 Beginning with C++

2.1.1 What is C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA in the early 1980s. Stroustrup wanted to combine the best features of Simula67 and C and create a more powerful language that could support object-oriented programming features. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of simula67. Since the class was a major addition of the original C language. Stroustrup initially called the new language 'C with classes'. In 1983 the name was changed to C++. C++ is an enhanced version of C. During the early 1990s, the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardized these changes and added several new features to C++.

C++ is a superset of C. Almost all C programs are also C++ programs. The most important facilities that C++ adds on to C are classes, inheritance, function overloading and operator overloading. These features enable creating abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language. The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C.

2.1.2 Applications of C++:

C++ is a versatile language for handling very large programs. C++ programs are easily maintainable and expandable. When a feature needs to be implemented, it is very easy to add to the existing structure of an object.

It is suitable for virtually any programming task including development of editors, communication systems and any complex real-life application systems.

- It is used in computer animation.
- It is used to design compilers.
- It is used to access relational databases.
- It is used in simulation and modeling.
- It is used to develop expert system.
- It is used to develop computer games.

2.1.3 A SIMPLE C++ PROGRAM:

Consider an example of a C++ program that prints a string on the screen.

```
#include <iostream.h>          // include header file
main()
{
cout<<"C++ is better C.";   // C++ statement
}
// End of example
```

Program Features

Like C, the C++ program is a collection of functions. The example contains only one function, main(). As usual, execution begins at main(). Every C++ program must have a main(). C++ is a *free-form* language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

Comments

C++ introduces a new comment symbol // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and what ever follows till the end of the line is ignored. There is no closing symbol. Double slash comment is basically single line comment. Multiline comments can be written as follows:

```
//This is an example of
// C++ program to illustrate
// some of its features
```

The C comment symbols /*, */ are still valid and more suitable for multiline comments. The following comment is allowed:

```
/* This is an example of
C++ program to illustrate
Some of its features
*/
```

For example, the double slash comment cannot be used in the manner as shown below:

```
for (j=0; j<n; /* loops n times */ j++)
```

Output operator

The only statement in program is an output statement. The statement

cout << "C++ is better C.";

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, **cout** and **<<.** The identifier **cout** is a predefined object that represents the standard output stream in C++. The standard output stream represents the screen. It is also possible to redirect the output to other output devices.

The operator << is called the *insertion or put to* operator. It inserts the contents of the variable on its right to the object on its left. The object **cout** has a simple interface. If **string** represents a string variable, then the following statement will display its contents:

cout << string;

Note that the operator << is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as operator overloading.

The iostream.h File

We have used the following #include directive in the program

#include <iostream.h>

This directive causes the preprocessor to add the contents of the iostream.h file to the program. It contains declarations for the identifier cout and the operator <<.

The header file iostream.h should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use iostream.hpp; yet others iostream.hxx.

We must include appropriate header files depending on the contents of the program and implementation. For example, if we want to use the very familiar printf() and scanf() functions, the header file stdio.h must be included.

The following table provides lists of C++ standard library header files that may be needed in C++ programs.

Header file	**Contents and Purpose**	**New Versio**
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging.	<cassert>
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase	<cctype>

	letters and vice versa.	
<float.h>	Contains the floating-point size limits of the system.	<cfloat>
<limits.h>	Contains the integral size limits of the system.	<clmits>
<math.h>	Contains function prototypes for math library functions.	<cmath>
<stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them.	<cstdio>
<stdlib.h>	Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions.	<cstdlib>
<string.h>	Contains function prototypes for C-style string processing function.	<cstring>
<time.h>	Contains function prototypes and types for manipulation the time and date.	
<iostream.h>	Contains function prototypes for the standard input and standard output functions.	<iostream>
<iomanip.h>	Contains function prototypes for the stream manipulators that enable formatting of streams of data.	<iomanip>
<fstream.h>	Contains function prototypes for functions that perform input from files on disk and output to files on disk.	<fstream>

***Table**. Commonly used old-style header files*

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive, like

Using namespace std;

Return Statement

In C++, **main()** returns an integer type value to the operating system. Therefore, every **main()** in C++ should end with a **return(0)** statement; otherwise a warning or an error might occur. Since **main()** returns an integer type value, return type for **main()** is explicitly specified as **int**. The default return type for all functions in C++ is int.

2.1.4 More C++ Statements

Let us consider another C++ program. Program to read two numbers form the keyboard and display their average on the screen. C++ statements to accomplish this are shown in Program.

```
#include <iostream.h>
using namespace std;
int main()
{
   float number1, number2, sum, average;
   cout << ”Enter two numbers: ”;   //prompt
   cin >> number1;       // Reads numbers
   cin >> number2;       //from keyboard

   sum = number1 + number2;
   average = sum/2;

   cout<< ” Sum=   “ <<sum<< ”/n”;
   cout<< ” Average=” <<average<< ”\n”;
   return 0;
}
```

The output of program is:

```
Enter two numbers: 6.5  7.5
Sum = 14
Average = 7
```

Variables

The program uses four variables number1, number2, sum, and average. They are declared as type float by the statement.

float number1, number2, sum, average;

All variables must be declared before they are used in the program.

Input operator

The statement

cin >> number1;

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It takes the value from the keyboard and assigns it to the variable on its right.

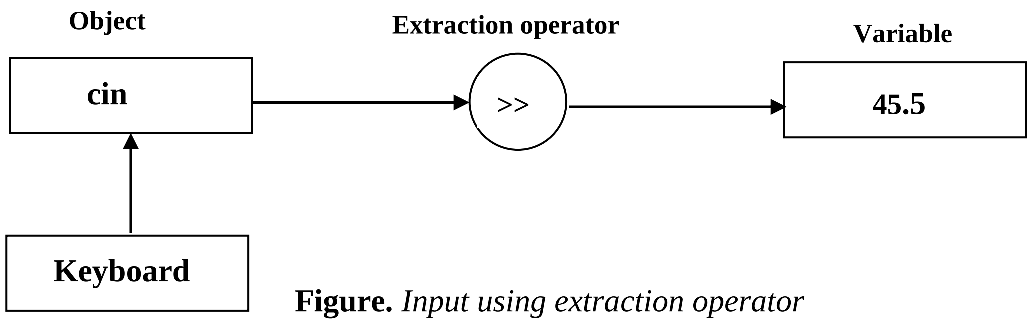


Figure. *Input using extraction operator*

Cascading of I/O operators

We have used the extraction operator << repeatedly in the last two statements for printing results. The statement

cout<< “Sum= “ << sum << “\n”;

first sends the string “Sum=” to **cout** and then sends the value of **sum**. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called ***cascading***. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << “Sum  = “ << sum << “\n”
         << “Average = “ <<average << “\n”;
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << “Sum = “<< sum << “,”
         << “Average = “<< average<< “\n”;
```

The output will be:

```
Sum = 14, Average = 7
```

We can also cascade input operator >> as shown below:

cin>> number1 >> number2;

The values are assigned from left to right. If we key in two values such as 10 and 20, then 10 will be assigned to **number1** and 20 to **number2.**

2.1.5 An Example with Class

One major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types. Program shows the use of class in C++ program

```
#include <iostream.h>
using namespace STD;
class person                          // NEW DATA TYPE
{
     char name[30];
     int age;
     public:
       void getdata(void);
       void display(void);
};
void person :: getdata(void)// MEMBER FUNCTION
{
       cout<<  “Enter name: ”;
       cin>> name;
       cout<< “Enter age: ”;
       cin>>age;
}
void person :: display(void)// MEMBER FUNCTION
{
        cout<<”\nName: ”<<name;
         cout<<”\nAge: ”<<age;
}
int main()
{
  person p;      // OBJECT OF TYPE person
  p.getdata();
  p.display():
  return 0;
}
```

The output of program is:

```
        Enter Name: Ravinder
        Enter Age: 30
        Name: Ravinder
        Age: 30
```

The program defines **person** as a new data of type **class**. The class **person** includes two basic data type items and two functions to operate on that data. These functions are called *member functions*. The **main** program uses **person** to declare variables of its type. Class variables are known as **objects.** Here, **p** is an object of type **person**. Class objects are used to invoke the functions defined in that class.

2.1.6 Structure of C++ Program

A typical C++ program would contain four sections as shown in figure. These sections may be placed in separate code files and then compiled independently or jointly.

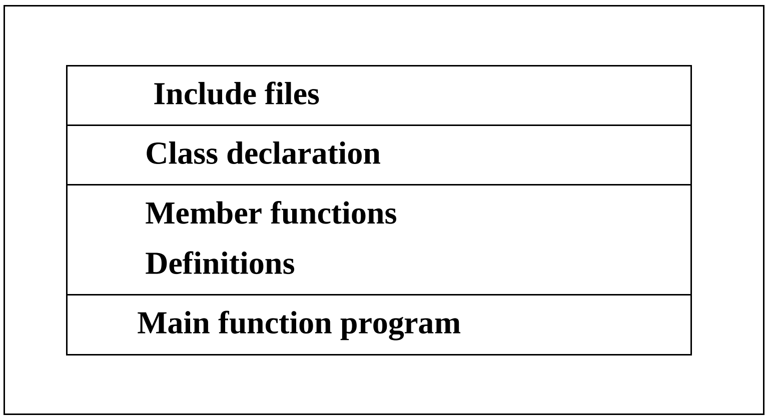


Figure. ***Structure of a C++ program***

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. Finally, the main program that uses the class is placed in a third file which “includes” the previous two files as well as any other files required. This approach is based on the concept of *client-server* model.

2.2 Tokens, Expressions and Control Structures

2.2.1 Introduction

C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions.

2.2.2 Tokens

The smallest individual units in a program are known as tokens. More than one token can appear in a single line separated by white spaces. White space may be blank, carriage return or tab. C++ has the following tokens.

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language.

2.2.3 Keywords

Keywords are reserved words which belong to C++ language. They cannot be used as variable names or other user-defined program elements. They have standard predefined meaning. Keywords should be written in lowercase.

Example

asm char delete extern if operator return struct try void auto class do float inline private short switch typedef volatile break const double for int protected signed template union while case continue else friend long public sizeof this unsigned catch default enum goto new register static throw virtual

2.2.4 Identifiers and Constants

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following are rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared *keyword* cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. C recognizes only the first 32 characters in a name; rather ANSI C++ has no limit. Constants refer to fixed values that do not change during the execution of a program. There are several kinds of literal constants.

Example

123	// decimal integer
12.34	// floating point integer
'C'	//character constant
"C++"	// string constant
37	// octal integer

2.2.5 Basic Data Types

Each variable in C++ has a data type. Data types specify the size and type of values that can be stored. The C++ language offers a set of data types. They can be classified into the following three categories:

i. User-defined type
ii. Built-in type
iii. Derived type

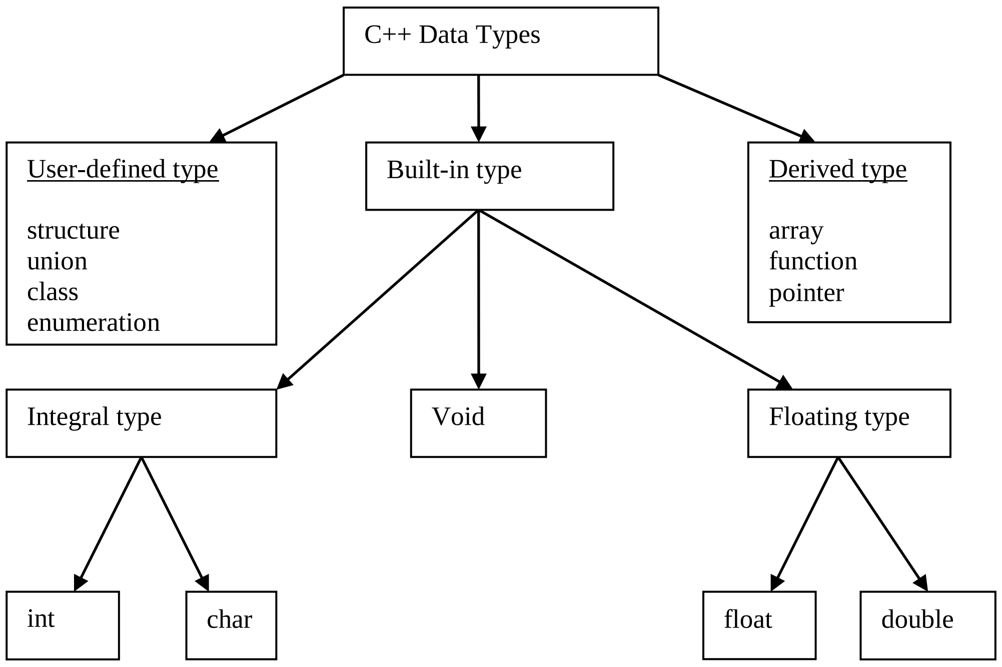


Figure. ***Hierarchy of C++ data types***

Both C and C++ compilers support all the built-in data types. Table lists all combinations of the basic data types and modifiers along with their size and range.

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
Long double	10	3.4E-4932 to 1.1E+4932

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of void is in the declaration of generic pointers. Example:

```
void *gp;       //gp becomes generic pointer
```

A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *) ptr1;
```

2.2.6 User-Defined Data Types

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape {circle, square, triangle};
  enum colour {red, blue, green, yellow};
  enum position {off, on};
```

The enumerated data types differ slightly in C++. In C++, the tag names **shape, colour,** and **position** become new type names. That means we can declare new variables using these tag names. Examples:

```
shape ellipse;                  //ellipse is of type shape
colour background;              // background is of type colour
```

In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value. Examples:

```
colour background = blue;            //allowed
colour background = 7;               //Error in C++
colour background = (colour) 7;      //OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c=red;        //valid, colour type promoted to int
```

By default, enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second and so on. For example,

```
enum colour {red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7. C++ also permits the creation of anonymous enums (i.e., enums without tag names). Example:

```
enum {off, on};
```

In C++, an **enum** defined within a class is local to that class only.

2.2.7 Derived Data Types

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

char string[3] = "xyz"

is valid in ANSI C. But in C++ the size should be one larger than the number of characters in the string

char string[4]= "xyz"; // O.K. . for C++

Functions

A function is a part of a program that can be invoked from other parts of the program as often needed. Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs.

Pointers

Pointers are variables that contain the addresses of other variables. Pointers are declared and initialized as in C. Examples:

```
int *ip; //int pointer
ip =&x;         //address of x assigned to ip
*ip = 10;       //50 assigned to x through indirection
```

C++ adds the concept of **constant pointer** and **pointer to a constant.**

```
.   char * const ptr1 = "GOOD";         //constant pointer
```

We can modify the address that ptr1 is initialized to

```
int const * ptr2 = &m;          //pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed

Pointers are extensively used in C++ for memory management and achieving polymorphism.

2.2.8 Symbolic Constants

There are two ways to create symbolic constants in C++:

1. Using the qualifier **constant** and
2. Defining a set of integer constants using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in anyway. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

```
const int size =10;
char name [size];
```

It is illegal in C. As with long and short, if we use the **const** modifier alone, it defaults to **int.** For example,

const size =10;

means **const int size =10;**

The named constants are just like variables except their values cannot be changed.

C++ requires a **const** to be initialized. In ANSI C, const values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as static. To give a **const** value as external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++. Example:

```
extern const total =100;
```

Another method of naming integer constants is as follows:

```
enum { X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0,1 and 2 respectively. This is equivalent to:

```
const X=0;
const Y=1;
const Z=2;
```

We can also assign values to X, Y and Z explicitly.

```
enum{X=100, Y=50, Z=200};
```

such values can be any integer values.

2.2.9 Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C+ defines **int, short int,** and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char,** and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility. otherwise, a cast must be applied. These restrictions are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **int**s. Therefore,

sizeof (‘x’)

is equivalent to

sizeof (int)

in C. In C++, however char is not promoted to the size of **int** and therefore,

sizeof (‘x’)

equals

sizeof (char)

2.2.10 Declaration of Variables

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduce the errors. Example:

```
int main()
{
        float x;
        float sum = 0;                    //declaration
        for(int i=1; i<5; i++)            //declaration
        {
              cin>>x;
              sum = sum + x;
        }
        float average;
         average= sum/i;                  //declaration
         cout<<average;
         return 0;
}
```

2.2.11 Dynamic Initialization of Variables

Initialization of variables during the run time is called dynamic initialization of variables. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements.

int i;

cin>>i;

float avg=i/10;

The variable **avg** is initialized only during run time, because the value of **i** is available during run time only. Thus both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example

float average;

average = sum/i; *//declare where it is necessary*

can be combined into a single statement:

float average = sum / i; *// initialize dynamically at run time*

dynamic initialization is extensively used in object-oriented programming.

2.2.12 Reference Variables

A reference variable is a name that acts as an alternative name for a previously defined variable. A reference variable is created as follows:

Data-type & reference-name = variable-name

Example:

float total = 100;

float & sum = total;

total is a **float** type variable that has already been declared; **sum** is the alternative name declared to represent the variable **total**. Both the variables refer to the same data object in memory. Now, the statements

cout << total;

and

cout<< sum;

Both variables will print the same value and if we change values, it will reflect in both the variables.

A reference variable must be initialized the time of declaration. C++ also assigns additional meaning to the symbol **&**. Here, & is not an address operator. The notation **float &** means reference to float. Other examples are

int n[10];

int & x = n[10]; *// x is alias for n[10]*

char & a ='\n'; *// initialize reference to a literal.*

The variable **x** is an alternative to the array element **n[10].** The variable **a** is initialized to the new-line constant.

2.7.13 Operators In C++

An operator is a symbol which represents some operations that can be performed on data. The different types of operators are:

1. Arithmetic operators	**+ - * / % ++ --**
2. Relational operators	**> < >= <= = = !=**
3. Logical operators	**&& || !**
4. Assignment operators	**+= -= *= /= %=**
5. Conditional operators	**? :**
6. Bitwise operators	**& | ^ << >> ~**
7. Scope resolution operator	**::**
8. Pointer-to-pointer declarator	**::***
9. Pointer-to-member operator	**->***
10. Pointer-to-member operator	**.***
11. Memory release operator	**delete**
12. Line feed operator	**endl**
13. Memory allocation operator	**new**
14. Field width operator	**setw**

2.2.14 Scope Resolution Operator

C++ is a block structured language. Blocks and scopes can be used in constructing programs. The same variable name can be used to have different meanings in different blocks. A variable declared inside a block is said to be **local** to that block. The scope resolution operator is denoted by a pair of colon (::). This is used: (1) to define a member function outside the class, (2) to refer global variable. It takes the following form:

:: variable-name

This operator allows access to the global version of a variable. For Example

```
int a=50;          // global a
void main()
{
          int a=30;          // a redeclared, local to main
          cout<< " a = "<<a<<"\n";
          cout<< " ::a = "<<::a<<'\n";
```

```
            return 0;
    }
```

The output is:

a = 30;

::a = 50;

Note that **::a** will always refer to the global a.

2.2.15 Member Dereferencing Operators

Member dereferencing operators are operators which are used to access the member functions in a class through pointer. They are:

::*	To declare a pointer to a member of a class.
*	To access a member using object name and a pointer to that member.
->*	To access a member using a pointer to the object and a pointer to that member.

2.2.16 Memory Management Operators

Memory management operators are used to manage the available memory in an efficient manner. There are two memory management operators. They are (i) **new**, and (ii) **delete.** An object can be created by using **new**, and destroyed by using **delete** as and when required. The **new** operator can be used to create objects of any type. It takes the following form:

Pointer-variable = **new** data-type;

The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The data-type may be any valid data type. The pointer-variable holds the address of the memory space allocated.

Examples:

```
p = new int;
q = new float;
```

where **p** is a pointer of type **int** and **q** is a pointer of type **float** alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

pointer-variable = **new** data-type(value);

Here, **value** specifies the initial value**.** Examples**:**

int *p = **new** int(25);

float *q = **new** float(7.5);

New can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

pointer-variable = **new** data-type [size];

Here, size specifies the number of elements in the array. For example, the statement

int p* = new int[10];

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form for the delete operator is

delete pointer-variable;

The pointer variable is the pointer that points to a data object created with **new.**

Examples: **delete p;**

delete q;

if we want to free a dynamically allocated array, we must use the following form of delete: **delete** [size] pointer-variable; Here, the size specifies the number of elements in the array. In C++, you need not specify the size. For example,

delete [ ] p; will delete the entire array pointed to by **p.**

2.2.17 Manipulators

Manipulators are operators that are used to format the data to be displayed, with the insertion operator (<<). The most commonly used manipulators are **endl** and

setw. The **endl** manipulator is used to transfer the control to a new line while printing. It has the same effect as using the newline character "**\n**". For example, the statement

```
....
....
cout<< “m =“<< m <<endl
    << “n =“<< n <<endl
```

```
        << "p ="<< p <<endl;
      ....
      ....
```

If we assume the values of the variables as 2597, 14 and 175 respectively, the output will appear as shown below:

```
m = 2597
n = 14
p = 175
```

Here, the numbers are right-justified. This form of output is possible only if we can specify the common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

cout<<**setw(5)**<<sum<<endl;

This manipulator **setw(5)** specifies a field width 5 for printing the value of the variable **sum.** This value is right-justified within the field as shown below:

		3	4	5

The **symbol** is the new manipulator which represents **Rs.** The identifier symbol can be used wherever we need to display the string **Rs.**

2.2.18 Type Cast Operator

Type cast operator is a operator which is used to convert the predefined data type into a new data type. There are two types of data type conversion. They are

(i) Explicit conversion

(ii) Implicit conversion

C++ permits explicit type conversion of variables or expressions using the type-cast operator. The following two versions of equivalent:

```
(type-name) expression      // C notation
type-name (expression)     // C++ notation
```

examples:

```
average =sum / (float)i;   //C notation
average =sum / float(i);   // C++ notation
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef  int   * int _pt;
```

p = int_pt(q);

2.2.19 Expressions and Implicit Conversations

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands and zero or more operators to produce a value. Expressions may be of seven types, namely, constant expressions, integral expressions, float expressions, pointer expressions, relational expressions, logical expressions and bitwise expressions..

Constant expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m * n – 5
m - 'x'
5 + int(2.0)
```

Where **m** and **n** are floating-point variables

Float expressions are those which, after all conversions, produce floating-point results, Examples:

```
x + y
x *  y /10
5 + float(10)
```

Where **x** and **y** are floating-point variables.

Pointer expressions produce address values. Examples:

```
&m
ptr
ptr  + 1
"xyz"
```

Where **m** is variable and **ptr** is a pointer.

Relational expressions yield results of type bool which take a value true or false. Examples:

```
x <= y
```

a + b == c + d

Logical expressions combine two or more relational expressions and produces boolean type results. Examples:

a>b && x = =1

x ==10 || y == 5

Bitwise expressions are used to manipulate data at bit level. Examples:

x<<3 // Shift three bit position to left

y>>1 // Shift one bit position to right

2.2.20 Special Assignment Expressions

2.2.20.1 Chained assignment

x = (y = 10)

or

x = y = 10;

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. Example:

float a = b = 12.34 is illegal.

This may be written as,

float a=12.34, b=12.34

Embedded Assignment

x = (y = 50) + 10;

(y = 50) is an assignment expression, known as embedded assignment. First , 50 is assigned to y and 60 is assigned to x.

2.2.20.2 Compound Assignment

Like C, C++ supports a compound operator which is a combination of assignment operator with a binary arithmetic operator. For example,

x = x + 10;

This may be written as

x +=10;

The operator += known as *compound assignment operator***.**

We can mix data types in expressions. For example,

float m;

m = 5 + 7.5;

is a valid statement. The operands 5 and 7.5 in the expression are different data types. But while execution the integer data 5 is automatically converted into float type and the operation is performed. So the result is 12.5. Wherever data types are mixed in an expression, C++ performs the conversion automatically. This process is known as *implicit or automatic conversion.*

2.2.21 operator overloading

Overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. For example, the input/output operators << and >> are good examples of operator overloading. The << operator is used for shifting of bits as well as displaying the values of various data types. This has been made possible by the header file *iostream.h* where a number of overloading definitions for << are included. Thus, the statement

cout<<75.86;

invokes the definition for displaying a **double** type value, and

cout<<"well done";

invokes the definition for displaying a **char** value. Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects.

2.2.22 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. The operators present in an expression are evaluated based on the priority or precedence of operators. Associativity means how an operator associates with its operands. Associativity of an operator is either left to right operation or right to left operation. For example, the associativity of = is from right to left. i.e., the right hand side value is assigned to left hand side variable.

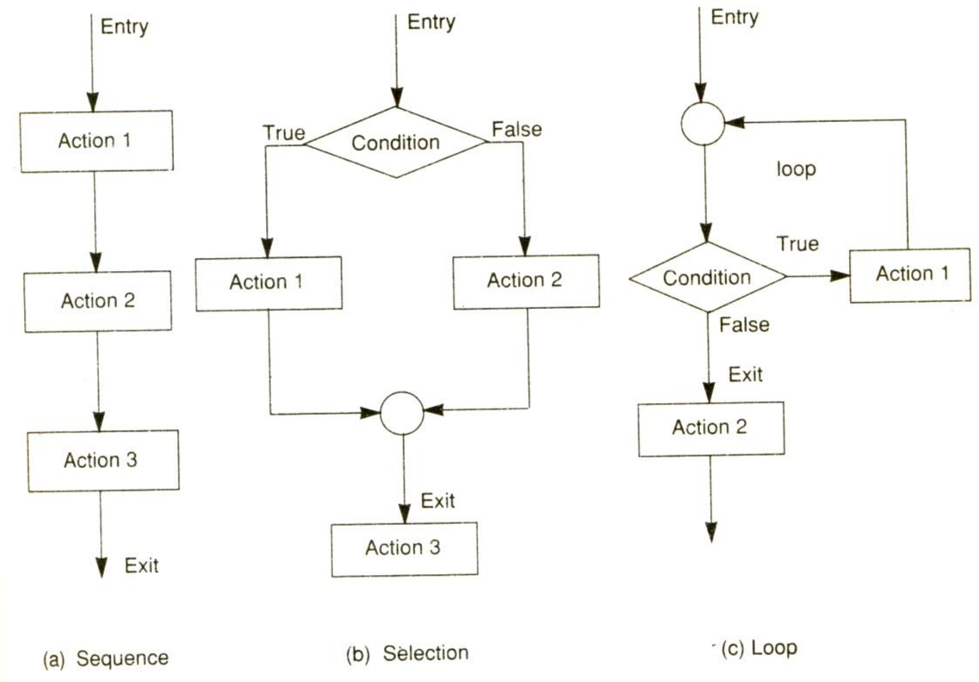
Operator	**associativity**
::	left to right
-> . ( ) [ ] postfix++ postfix--	left to right
Prefix++ prefix-- ~ ! unary + unary-	
Unary* unary &(type) size of new delete	right to left
-> * *	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
<< = >>=	left to right

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

2.2.23 CONTROL STRUCTURES

One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one r any combination of the following three control structures :

1. Sequence structure(Straight line)
2. Selection structure(branching)
3. Loop structure(iteration or repetition)



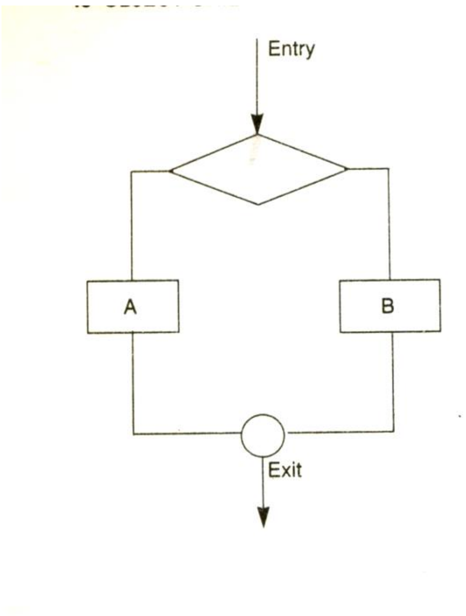
Basic Control Structures

It is important to understand that all program processing can be coded by using only the three logic structures. The approach of using one or more of these basic control constructs in programming is known as structured programming.

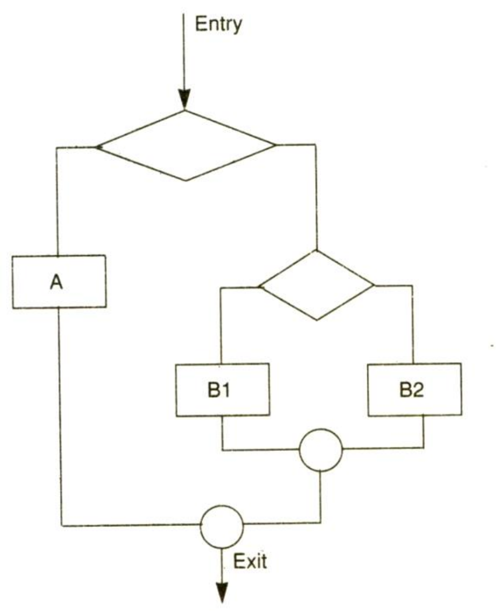
2.2.23.1 The if statement

The if statement is implemented in two forms:

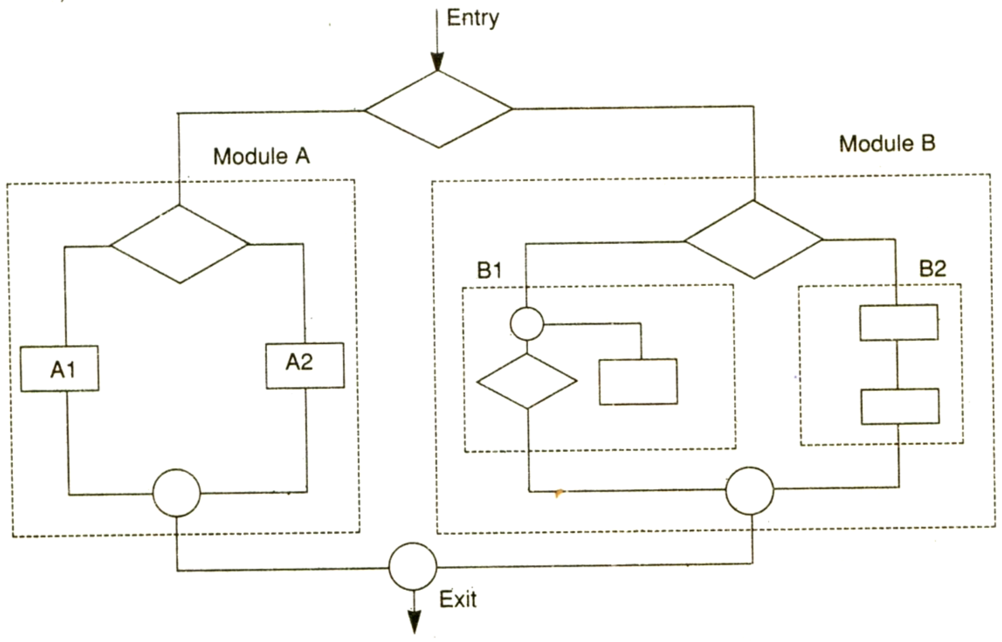
- Simple if statement
- if….else statement



a)First level of Abstraction

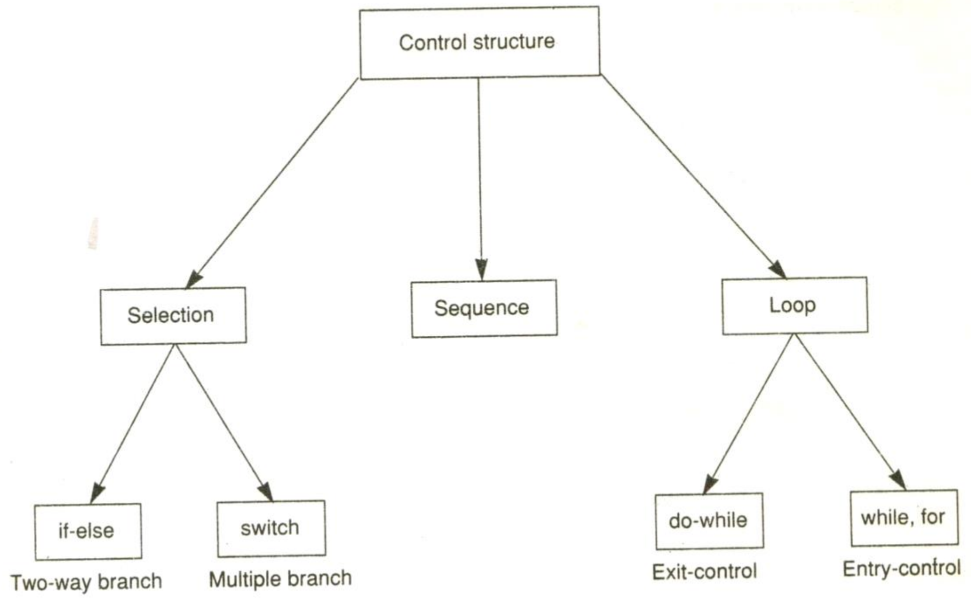


b)Second level of Abstraction



c) Detailed Flow Chart

Different level of Abstraction



Examples:

Form 1

```
 if(expression is true)
{
                action1;
}
action2;
action3;
```

Example

```
if (age<18)
{
  cout<<"The person is not eligible for vote";
}
```

Form 2

```
 If(expression is true)
{
   action1;
}
else
{
    action2;
}
 action3;
```

Example

```
if (age<18)
{
  cout<<"The person is not eligible for vote";
}
else
 {
  cout<<"The person is eligible for vote";
}
```

2.2.23.2 The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
      case1:
      {
        action1;
       }
       case2:
        {
         action2;
         }
         case3:
         {
           action3;
         default:
       {
         action4;
        }
  }
action5;
Example
```

```
 index=marks/10;
switch(index)
{
          case 10:
          case 9:
          case 8:
                    Grade=”Honours”;
                    break;
          case 7:
          case 6:
            Grade=”first class”;
            break;
case 5:
            Grade=”second class”;
           break;
case 4:
            Grade=”third class”;
            break
default:
             Grade=”fail”;
             break;
}
```

2.2.23.3 The do-while statement

The do-while is an exit-controlled loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
action1;
}
while(condition is true);
action2;
```

Example

```
class total
{
 public:
   public void  sum( )
  {
            int sum,n;
            sum=0;
             n=1;
            do
            {
                      sum=sum+n;
```

```
                n++;
          } while(n<=5);
         cout<<"sum="<<sum;
}
};
int main( )
{
   total t;
   t.sum( );
}
O/P is:sum= 15
```

2.2.23.4 The while statement

This is also a loop structure, but is an entry-controlled one. The syntax is as follows:

```
            while(condition is true)
            {
             action1;
            }
            action2;
```

Example

```
class total
{
 public:
   public void  sum( )
  {
         int sum,n;
           sum=0;
            n=1;
           while (n<=5)
           {
                  sum=sum+n;
                    n++;
          }
         cout<<"sum="<<sum;
}
};
int main( )
{
   total t;
   t.sum( );
}
O/P is:sum= 15
```

2.2.23.5 The for statement

The for is an entry-controlled loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value;  test; increment)
   {
        action1;
   }
   action2;
```

Example

```
 class total
{
 public:
   public void  sum( )
  {
        int sum=0;
        for(int i=1;i<=5;i++)
          {
                sum=sum+i;
        }
        cout<<"sum="<<sum;
}
};
int main( )
{
   Total t;
   t.sum( );
}
O/P is:sum= 15
```

2.3.1 FUNCTIONS IN C++

Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible.

2.3.2 THE MAIN FUNCTION

C does not specify any return type for the **main( )** function which is the starting point for the execution of a program.

The definition of **main( )** would look like this:

```
main ( )
{
        //main program statements
}
```

This is perfectly valid because the **main ( )** in C does not return any value.

In C++, the main ( ) returns a value of type int to the operating system. C++, therefore, explicitly defines matching one of the following prototypes:

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

The functions that have a return value should use the return statement for termination. The main( ) function in C++ is , therefore , defined as follows:

```
int    main ( )
{
         ……..
         ……..
          return 0 ;
 }
```

Since the return type of functions is int by default, the keyword int in the main( ) header is optional. Most C++

Compilers will generate an error or warning if there is no return statement. Turbo C++ issues the warning

```
Function should return a value
```

And then proceeds to compile the program. It is good programming practice to actually return a value from main( ).

2.3.3 FUNTION PROTOTYPING

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return value. With function prototyping, a template is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions. There is a major difference between C and C++.

Function prototype is a declaration statement in the calling program and is of the following form:

```
type function-name  (argument-list);
```

The argument-list contains the types and names of arguments that must be passed to the function

Example

```
float volume( int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume( int x, float y,  z);
```

is illegal.

We can either include or exclude the variable names in the argument list of prototypes.

In function definition, names are required because the arguments must be referenced inside the function.

Example:

```
float volume( int a, float b, float c);
{
        float  v= a*b*c;
        ……
        ……
    }
```

The function volume() can be invoked in a program as follows:

```
    float cube1= volume(b1,w1,h1);        //function call
```

The variable b1,w1, and h1 are known as the actual parameters which specify the dimensions of cube1.

We can also declare a function with an empty argument list, as in the following example:

```
    void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to statement

```
    void display(void);
```

2.3.4 CALL BY REFERENCE

Provision of the reference variables in C++ permits us to pass parameters to functions by reference. When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function.

```
void  swap(int a, int b)          //a and b are reference variables
{
        int  t = a;               //dynamic initialization
        a = b;
        b = t;
}
```

Now, if m and n are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of m and n using their aliases(reference variables) a and b.

2.3.5 RETURN BY REFERENCE

A function can also return a reference. Consider the following:

```
    int & max(int &x, int &y)
    {
```

```
                if(x>y)
                        return x;
        else
                return y;
        }
```

Since the return type of max() is int &, the function returns reference to x or y. Then the function call such as max(a,b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns -1 to a if it is larger, otherwise -1 to b

2.3.6 INLINE FUNCTIONS

To eliminate the cost of calls to small functions C++ proposes a new feature called inline function. A inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code. The inline funct6ions are defined as follows:

```
inline function-header
{
        function body
}
```

Example

```
inline double cube(double a)
{
return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

2.3.7 DEFAULT ARGUMENTS

C++ allows us to call a function without specifying all its arguments. In such case, the function assigns a default value of the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype with default values:

```
float amount(float principal, int period, float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

```
value = amount(5000,7);        //one argument missing
```

passes the vale of 5000 to principal and 7 to period and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);
```

passes an explicit value of 0.12 to rate.

2.3.8 Const ARGUMENTS

In C++ argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

2.3.9 FUNCTION OVERLOADING

Function overloading refers to use the same function name to create different functions that perform a variety of tasks. This is known as function polymorphism in OOP.

Using this concept of function overloading to design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Examples:

```
//Declarations
int add(int a, int b);                      //prototype1
int add(int a, int b, int c);               //prototype2
double add(double x, double y);             //prototype3
double add(double p, int q);                //prototype4
//Function calls
cout<<int add(5,10);                        //prototype1
```

```
cout<<int add(5,10,15);              //prototype2
cout<<double add(12.5,7.5);          //prototype3
cout<<double add(15, 10.5);          //prototype4
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution.

2.3.10 FRIEND AND VIRTUAL FUNCTIONS

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. We will discuss them later.

State whether the following statements are TRUE or FALSE:

a) In C++, a function contained within a class is called a member function.

b) In C++, it is very easy to add new features to the existing structure of an object.

c) The concept of using one operator for different purposes is known as Operator Overloading.

d) The output function **printf()** cannot be used in C++ programs.

e) A function argument is a value returned by the function to the calling program.

f) A function can return a value by reference.

Questions:

1) What do we need the preprocessor directive **#include<iostream.h**>?

2) Describe the structure of a C++ Program?

3) What are the applications of C++?

4) Describe with examples the uses of enumeration data types.

5) Why is an array called a derived data type?

6) What is a reference variable? What is the major use of this variable?

7) List at least four new operators added by C++?

8) What is the application of the scope resolution operator :: in C++?

9) What are the advantages of function prototypes in C++?

10) What is the main advantage of passing arguments by reference?

11) When will you make a function ***inline***? Why?

NOTES

NOTES

UNIT-III

UNIT-III

3.1.0 CLASSES AND OBJECTS

3.1.1 INTRODUCTION:

The most important feature of c++ is a “class”. Its significance is highlighted by the facts that Stroustrup initially gave the name “c with classes” to his new language. A class is an extension of the idea of structure used in c. It is a new way of creating and implementing a user-defined data type. we shall discuss, in this.

3.1.2.C STRUCTURES REVISITED:

They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related items. It is a user –definied data type with a template that servers to define its data properties.

```
struct student
 {
      Char name[20];
      Int roll_number;
     Float total_marks;
 };
```

3.1.3 SPECIFYING THE CLASS:

A class is a way to bind the data and its associated functions together. it allows the data to be hidden, if necessary, from external use. when defining a class, we are creating a new abstract *data type* that can be treated

Like any other built-in data type. Generally, a class specification has two types**:**

- **Class declaration**
- **Class function definitions**

The class declaration describe the type and scope of its members. The class functions definitiomns describe how the class functions are implemented.

General form:

```
class class-name
{
    private:
             variable declarations;
              function declarations;
    public:
        variable declarations;
        function declarations;
  };
```

- The variables declared inside the class are know as data members and the functions are know as member functions.
- The binding up of data and functions together into a single class-type variable is referred to as encapsulation.

A simple example:

```
class item
{
        int number;
        float cost;
public:
        void getdata(int a, int b);
        void putdata(void);
};
```

Creating Objects:

Remember that the declaration of class defined any objects of class-name. Once a class has been declared, we can create variable of that type by using the class name.

Example

```
            classname  object1,object2,…..
```

ACCESSING CLASS MEMBERS:

The private data of a class can be accessed only through the member function of that class.

The following is the format for calling a member functions:

Object-name. function-name(actual-arguments);

Example:

```
        x. getdata (100,7.5);
        similarly
        x. putdata ();
```

both can implement in the class.

}

Sample program:

```
class xyz
{
        int x;
        int y:
public:
        int z;
};
……
……
xyz p;
p.x=0;      // error,x is private
p.z=10;    // ok,z is public

…….
…….
```

3.1.4 DEFINING MEMBER FUNCTIONS

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task .therefore, the code for the function body would be identical in the both the classes.

However, there is a suitable difference in this section.

OUTSIDE THE CLASS DEFINITION:

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal function definition, the ANSI prototype

must be used for defining the function header.

```
return-type class-name : : function-name (argument declaration)
{
        function body
}
```

The membership label class-name :: tells the compiler that the function function-name belongs to the class class-name, that is, the scope resolution operator.

Example:

```
void item ::getdata(int a,float b)
{
        number = a;
        cost =b;
}
void item :: putdata (void)
{
cout<< “number”<<”\n”;
cout<<”cost”<<”\n”;
}
```

Since these functions do not return any value, their return type is void. Functions arguments are declared using the ANSI prototype.

SPECIAL CHARACTERISTICS:

The member functions have some special characteristics that are often used in the program development, these characteristics are:

- Several different classes can use the same function name. The ‘membership label’ will resolve their scope.
- Member functions can access the private data of the class. a non-member function cannot do so.
- A member function can call another member function directly, without using the dot operator.

INSIDE THE CLASS DEFINITION:

Another method of defining a member function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
        int number;
        float cost;
public:
```

```
                 void getdata(int a,float b);
                 void putdata(void)
                 {
                    cout<<number<<"\n";
                   cout<<cost<<"\n";
                 }
        };
```

Whenever the function defined inside the class, it is treated as an inline function.

3.1.5 A C++ PROGRAM WITH CLASS:

```
         // CLASS IMPLEMENTATION
          #include <io.stream>
          class item            //class declaration
          {
                   int number;
                   float cost;
          public;
                 void  getdata (int a, float b);
                 void  putdata(void)
          {
                cout <<  "number  :"  <<number   <<"\n";
                cout   << "cost   :"    << cost    <<"\n";
            }
         };
//member function definition…………..
void item: : get data ( int a, float b)//use membership label
{
 number=a;
 cost=b;
 }
// main program……………//
main ()
{
 item  x;
cout<<”\n objects x” << “\n”;
x . getdata (100,299.95);
x . putdata ( );
item  y;
cout<<”\n objects y “ << “\n”;
y. getdata(200,175.50);
y . putdata( );
}
```

Output:

```
  Object x
        number:100
        cost : 299.950012
Object  y
        number : 200
        cost : 175.5
```

3.1.6MAKING AN OUTSIDE FUNCTION INLINE:

One of the objectives of oops is to separate the details of implementation from the class definition.

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition.

EXAMPLE:

```
class item
{
……..
…….
public:
 void getdata(int a,int b);
};
inline void item :: getdata(int a,float b)//definition
{
  number=a;
 cost=b;
}
```

3.1.7 NESTING OF MEMBER FUNCTIONS:

A member function can be called by using its name inside another member function of the same class this is know as *nesting of membership functions.*

EXAMPLE:

```
#include<iostream>
Class nest
{
Int m=2, n=3,a=0;
Public:
    Void input(void);
    Void display(void);
};
Int nest : : input(void)
{
A = M+N;
Display();
}
Int nest :: display(void)
{
 Cout<<"the sum ="<<a;
}
Int main()
{
Nest n;
n.input();
 return 0;
}
```

3.1.8 PRIVATE MEMBER FUNCTIONS:

Although it is normal practice to place all the data item in a private section and all the functions in public, some situations may require certain functions to be hidden from the outside calls.

A private member function can only called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
        int m;
        void read(void);
public;
        void update(void);
        void write(void);
};
```

If s1 is an object of sample, then

```
s1.read();
```

is illegal. However, the function read() can be called by the function update()to update the value of m.

```
void sample :: update(void)
{
  read();
}
```

3.5.9 ARRAY WITHIN A CLASS:

The arrays can be used as member variable in a class. the following class definition is valid.

```
const int size=10;
        class array
        {
                int a[size];
        public:
                void setval(void);
                void display(void);
        };
```

The array variable a[] declared as a private member of the class array can be used in the member functions, like any other array variable. we can perform any operations on it.

3.1.10 MEMORY ALLOCATION FOR OBJECTS:

The member functions are created and placed in the memory space only once when they are defined as a part of a class specification. since all the objects belonging to that class use the same member functions, no separate space is allocated for member function when the objects are created. only space for member variables is allocated separately for each object. Separate memory locations for the objects re essential, because the member variable will hold different data value for different objects.

3.1.11 STATIC DATA MEMBERS:

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a '**C** 'static variable.

CHARACTERISTICS:

- it is initialized to zero when the first object if its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how objects are crated.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class.

3.1.12 STATIC MEMBER FUNCTION:

Like static member variable, we can also have static member functions. a member function that is declared static has the following properties:

- A static function can have access to only other static members(functions or

  variables)declared in the same class.
- A static member function can be called using the class name(instead of its

  objects)as follows:

```
class-name : : function-name;
```

3.1.13 ARRAY OF OBJECTS:

We know that an array can be of any data type including struct. Similarly, we can also have array of variable that are of the type class. such variable are called array of objects. Consider the following definition:

```
 class employee
{
        char name[30];
        float age;
public:
        void getdata(void);
        void putdata(void);
};
```

The identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employees.

EXAMPLE

```
employee manager [3];
```

The array manager contains three objects (manager) namely, manager[0],manager[1],manager[2] of type employee class. Since an array of objects behave like any other array, we can use the usual array accessing methods to access individual elements, and the dot member operator to access the member functions.

Example:

```
manager[i].putdata();
```

Will display the data of the ith element of the array manager . That is, this statement request the object manager[i] to invoke the member function putdata().

Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

3.1.14 OBJECTS AS FUNCTIONS ARGUMENTS:

Like any other data type, an object may be used as a function argument, this can be done in two ways:

- a copy of the entire objects is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the objects is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. the second method is called pass-by-reference. when an address of the object is passed, the called function works directly on the actual object used in the call. this means that any changes made to the object used in the call. this means that any changes made to the objects inside the function will reflect in the actual objects. the pass-by-reference Method is more efficient since it requires to pr\as only the address of the object and not the entire objects.

3.1.15 FRIENDLY FUNCTION:

We have been emphasizing this chapter that the private members cannot accessed from outside the class. that is a non-member function cannot have an access to the private data of a class. however, there could be a situation where we would like two classes to share a particular function.

We would like to use a function income tax to operate on the objects of both these classes. In such situations++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these. classes. such a function need not be a member any of these classes.

To make an outside function “friendly” to a class, we have to simply declare this function as a friend of the class as shown below:

```
class ABC
{
……..
……..
 public:
……..
…….

friend void xyz(void);
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal c++ function definition does not use either the keyword friend or the scope operator ::.

SPECIAL CHARACTERISTICS:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name(e.g.A.x)
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend function often used in the operator overloading which will be discussed here.

3.1.17 CONST MEMBER FUNCTION:

If a member function does not alter any data in the class, then we may declare it as a const member function as follows:

```
void mul(int,int) const;
double get-balance()const;
```

The qualifier const is appended to the function prototypes. The compiler will generated will generate an error message if such functions try to alter the data values.

3.1.18 POINTERS TO MEMBERS:

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be a member can be obtained by applying the operator & to a “fully qualified” class member name. a class member pointer can be declared using the operator ::* with the class name.

EXAMPLE

```
class a
{
  private:
              int m;
   public:
            void show();
};
```

3.1.19 LOCAL CLASSES:

Classes can be defined and used inside a function or a block. Such classes are called local classes.

EXAMPLE:

```
void test(int a)
{
  ……
…….
 class student
{
…..
…..
 };
```

```
        …
        student s1(a);
        …
        }
```

Local classes can use global variables and static variables declared inside the function but cannot use automatically local variables the globel variables should be used with the scope operator (::).

3.2 CONSTRUCTORS AND DESTRUCTORS

3.2.1 INTRODUCTION

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. It also provides another member function called the destructor that destroys the objects when they are no longer required.

3.2.2CONSTRUCTORS

It is special because its name is the same as the class name the constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

example

```
class integer
{
        int m,n;
public:
         integer(void);
  ……..
………
};
integer::integer(void)
{
 m=0,n=0;
}
```

when a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically.

CHARACTERISTICS

- They should be declared in the public section.
- They are invoked automatically when the objects are created.

- They do not have return type, not even void and therefore, and they cannot return values.
- They cannot be inherited, through a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses. An object with a constructor cannot be used as a member of a union.
- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.

3.2.3 PARAMETERIZED CONSTRUCTORS

C++ permits us to achieve This objective by passing arguments to the constructs function when the object are created. The constructors that can take arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

The constructors integer() may be modified to take arguments as shown below:

```
class integer
{
      int m,n;
public:
     integer (int x, int y)
  ……
……..
};
integer ::integer(int x,int y)
{
 m=x:
 n=y;
}
```

When a constructor has been parameterized, the object declaration statement such as integer int1;

INITIALISATION OF VALUES

We must pass the initial values as arguments to the constructor function when an object is declared. This can done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

integer int1= integer (0,100);

This statement creates an integer object int1 and passes the value 0 and 100 to it.The second is implemented as follows:

integer int1(0,100);

This method, sometimes called the shorthand method, is used very often as it is shorter,looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor.

The constructor functions can also be defined as inline functions.

EXAMPLE

```
 class integer
         {
                 int m,n;
         public:
                  Integer(int x, int y)
                 {
                  m=x;y=n;
                 }
         …
         ….
         };
```

However, a constructor can accept a reference to iits own class as a parameter. Thus , the statement

```
class a
{
…
….
public:
a(a&);
};
```

Is valid. In such cases, the constructor is called the copy constructor.

3.2.4 MULTIPLE CONSTRUCTORS IN A CLASS

So far we have used two kinds of constructors.they are

```
integer();
integer(int,int);
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from main().C++ permits to use both these constructors in the same class.

example

```
class integer
{
 int m,n;
public:
 integer()
{
        m=0;
        n=0;
}
integer(int a, int b)
{
        m=a;
        n=b;
}
integer (integer &i)
{
        m=i.m;
         n=i.n;
}
    };
```

This declares three constructors for an integer object.the first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as an argument.

3.2.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float image=0.0);
```

The default value of the argument image is zero then, the statement

```
complex c(5.0);
```

Assign the value 5.0 to the real variable and 0.0 to image (by default). However, the statement

```
complex c(2.0,3.0);
```

Assign 2.0 to real variable and 3.0 to image. The actual parameter, when specified, overrides the default value as pointed out earlier, the missing arguments must be the trailing ones.

3.2.6 DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too that is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors this provides the flexibility of using different format of data at run time depending upon the situation.

3.2.6 CONSTRUCTING TWO-DIMENTIONAL ARRAYS

We can construct matrix variables using the class type objects.

EXAMPLE

```
 #include<iostream>
class matrix
{
int **p;
int  d1,d2;
public:
 matrix(int x,int y);
void get-element(int i,int j,int value)
{
p[i][j]=value;
}
int & put-element(int I,int j)
{
return p[i][j];
}
};
matrix :: matrix(int x, int y)
```

```
{
d1 =x;
d2 = y;
p= new int *[d1];
for(int i=0; i< d1 ;i++)
p[i]= new int[d2];
}
int main()
{
 int m,n;
cout <<"ENTER SIZE OF MATRIX:";
cin>>m>>n;
matrix a(m,n);
cout<<"ENTER MATRIX ELEMENTS ROW BY ROW;
int i,j,value;
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
 cin>> value;
 a.get-element(i,j,value);
}
cout<<"\n";
cout<<a.get-element(1,2);
return 0;
};
```

OUTPUT:

```
ENTER SIZE OF MATRIX: 3 4
ENTER MATRIX ELEMENTS ROW BY ROW
11 12 13 14
15 16 17 18
19 20 21 22
```

Output

```
17
```

The constructor first creates a vector pointer to an int of size d1. then, it allocates, interactively an int type vector of size d2 pointed at by each element p[i].

Thus, space for the elements of a d1* d2 matrix is allocated from free store as show above.

3.2.7 COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object.

EXAMPLE:

```
integer i2(i1);
```

Would define the object i2 and the same time initialize it to the values of i1.

A copy constructor takes a reference to an object of the same class as itself as an argument.

3.2.8 DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects.

This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.

3.2.9 CONSTRUCTING TWO-DIMENSIONAL ARRAYS:

We can construct matrix variables using the class type objects. The example will explain in detail about constructors.

EXAMPLE:

```
        #incluude<iostream>
Clas matrix
{
 Int **p;
 Int d1,d2;
Public;
      Matrix (int x,  int y);
       Void get_element(int I, int j, int value);
{
P[i][j]=value;
}
Int & put_element(int I,int j);
{
Return p[i][j];
}
};
Matrix :: matrix (int x ,int j)
{
Di=x;
D2=y;
P=new int *[d1];
For(int I =0 ; i<d 1; i++)
P[i]= new int[d2];
}
Int main()
{
Int m,n;
Cout<< "ENTER SIZE OF MATRIX";
Cin>>m>>n;
Matrix  A(m,n);
Cout>> "ENTER MATRIX ELEMENTS ROW BY ROW  \n";
Int I,j,value;

For(i=0 ; i<m ;i++)
     For(j=0 ; j<n ;j++)
      {
          Cin>>value;
```

```
        A.get_element(I,j,value);
      }
Cout<<"\n";
Cout<<A.put_element(1,2);
Return 0;
};
```

```
OUTPUT:
ENTER SIZE OF MATRIX:  3 4
ENTER MATRIX ELEMENTS ROW BY ROW:
11  12  13  14
15  16  17  18
19  20  21  22
```

3.2.10 const OBJECTS

We may create and use constant objects using const keyword before object declaration.

```
const matrix x(m,n);
```

Any attempt to modify the values of m and n will generate compile-time error. Further, a constant object can call only const member functions. As we know, a const member is a function

Prototype or function definition where the keyword const appears after the function's signature.

Whenever const objects try to invoke non-const member functions, the compiler generates error.

3.2.11DESTRUCTORS

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a title.

EXAMPLE

```
 ~integer()
    {
    }
```

A destructor never takes any argument nor does it return any value it will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory.

example

```
  matrix  :: ~matrix()
{
 for(int i=0; i<d1 ;i++)
delete p[i];
delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

State whether the following statements are TRUE or FALSE:

1) Data item in a class must always be private.
2) A function designed as **private** is accessible only to member functions of that class.
3) A function designed as **public** can be accessed like any other ordinary functions.
4) Member functions defined inside a class specifier become inline functions by default.
5) Friend functions have access to only public members of a class.
6) Constructors, like other member functions, can be declared anywhere in the class.
7) Constructors do not return any values.
8) A constructor that accepts no parameter is known as the *default constructor*.
9) A class should have at least one constructor
10) Destructors never take any argument.

Questions:

1) What is a class?
2) What are objects? How they are created?
3) How is a member function of a class defined?
4) When do we declare a member of a class **static**?
5) What is a friend function? What are the merits and demerits of using friend function?
6) What is a constructor? Is it required to use constructors in a class?
7) How do we invoke a constructor function?
8) List some of the special properties of the constructor functions.
9) What is a parameterized constructor?
10) What is meant by dynamic initialization of objects?
11) Describe the importance of destructors?

NOTES

UNIT-IV

4.1.0 Operator Overloading and Type Conversions

4.1.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. C++ tries to make the user-defined data types behave in much the same way as the built-in types. C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can give additional meaning to all the C++ operators except Class member access operators (., .*), Scope resolution operator (::), Size operator (**sizeof**), and Conditional operator (?:).

4.1.2 Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function,* which describes the task. The general form of an operator function is:

```
returntype classname :: operator op (arg-list)
{
        Function body           // task defined
}
```

Where *returntype* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes.

The process of overloading involves the following steps:

1. First, create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function **operator** *op* ( ) in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

4.1.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator, when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items. Program shows how the unary minus operator is overloaded.

```
#include <iostream.h>
class space
{
        int x;
        int y;
        int z;
     public:
        void getdata (int a, int b, int c);
        void display (void);
        void operator-( );              // overload unary minus
};
void space :: getdata ( int a, int b, int c)
{
        x=a;
        y=b;
        z=c;
}
void space :: display (void)
{
        cout<< x << " ";
        cout<< y << " ";
        cout<< z << "\n";
}
void space :: operator-( )              // Defining operator-( )
{
        x=-x;
        y=-y;
        z=-z;
}
main ( )
{
        space S;
        S.getdata (10, -20, 30);
        cout<<  "S : ";
        S.display ( );
        -S;                             // activates operator-( )
        cout<<"S : ";
        S.display ( );
}
```

The program produces the following output:

```
                S : 10 -20 30
                S : -10 20 -30
```

Note that the function **operator-( )** takes no argument. This operator function changes the sign of data members of the object **S**. Since this function is a member function of the same class, it can directly access the members of the object which activated it. The function **operator-( )** does not return any value. It can work if the function is modified to return an object. Therefore, the changes made inside the operator function will not reflect in the called object.

4.1.4 Overloading Binary Operators

We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator. Let us illustrate how to add two complex numbers using a **friend** function. A statement like

```
        C=sum(A, B); //functional notation
```

was used. The functional notation can be replaced by an expression

```
        C=A+B;                  //arithmetic notation
```

By overloading the + operator using an **operator+( )** function. The program illustrates how this is accomplished.

```
#include<iostream.h>
class complex
{
        float x;                                //real part
        float y;                          //imaginary part
        public:
        complex( ) { }                    //constructor1
        complex(float real, float imag)   //constructor2
        { x = real; y = imag; }
        complex  operator+(complex);
        void display(void);
};
complex complex :: operator+(complex c)
{
        complex temp;         //temporary
        temp.x = x + c.x;     //float addition
        temp.y = y + c.y;     //float addition
        return(temp);
}
void complex :: display(void)
{
        cout<< x << "+ j" << y << "\n";
}
main( )
{
        complex C1, C2, C3;          //invokes constructor1
        C1=complex(2.5, 3.5);        //invokes constructor2
        C2=complex(1.6, 2.7);        //invokes constructor3
        C3=C1 + C2;                  //invokes operator+( )
        cout<< "C1 = "; C1.display( );
        cout<< "C2 = "; C2.display( );
        cout<< "C3 = "; C3.display( );
}
```

The output of program would be:

```
                C1 = 2.5 + j3.5
                C2 = 1.6 + j2.7
                C3 = 4.1 + j6.2
```

4.1.5 Overloading Binary Operators Using Friends

Friend functions may be used in the place of member functions for overloading a binary operator. The only difference being that a **friend** function requires two arguments to be explicitly passed to it while a member function requires only one.

The complex number program can be modified using a friend operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.

```
        friend complex operator+(complex, complex);
```

2. Redefine the operator function as follows:

```
        complex operator+(complex a, complex b)
        {
                return complex((a.x + b.x), (a.y + b.y));
        }
```

Program illustrates this using scalar multiplication of a vector. It also shows how to overload the input and output operators >> and <<.

```
#include<iostream.h>
const size = 3;
class vector
{
        int v[size];
   public:
        vector( );
        vector( int * x);
        friend vector operator * (int a, vector b); //friend 1
        friend vector operator * (vector b, int a); //friend 2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator >> (ostream &, vector &);
};
vector :: vector( )
{
        for(int i=0; i<size; i++)
                v[i]=0;
}
vector :: vector(int * x )
```

```
{
        for(int i=0; i<size; i++)
                v[i]=x[i];
}
vector operator *(int a, vector b)
{
        vector c;
        for(int i=0; i<size; i++)
                c.v[i] = a * b.v[i];
        return c;
}
vector operator *(vector b, int a)
{
        vector c;
        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}
istream & operator >> (istream & din, vector & b)
{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
ostream & operator << (ostream & dout, vector & b)
{
        dout<<"{"<<b.v[0];
        for(int i=0; i<size; i++)
                dout<<" , "<< b.v[i];
        dout<< ")";
        return(dout);
}
int x[size] = {2,4,6};
main( )
```

```
{
        vector m;               //invokes constructor 1
        vector n = x;           //invokes constructor 2

        cout<< “Enter elements of vector m”<<”\n”;
        cin>>m;
        cout<<”\n”;
        cout<<”m = “<<m<<”\n”;

        vector p,q;
        p = 2 * m;
        q = n * 2;

        cout<<”\n”;
        cout<<”p = “<<p<<”\n”;
        cout<<”q = “<<q<<”\n”;
}
```

Shown below is the output of program:

```
        Enter elements of vector m
        5 10 15
        m = (5, 10, 15)
        p = (10, 20, 30)
        q = (4, 8, 12)
```

4.1.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. For example, statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be manipulated like the built-in types. Since the strings vary greatly in size, we use **new** to allocate memory for each string and a pointer variable to point to the string array**.** Thus, we must create string objects that can hold information about length and location, which are necessary for string manipulations.

```
class string
{
        char *p;        //pointer to string
        int len;        //length of string
      public:
        …..             //member functions
        …..             //to initialize and
        …..             //manipulate strings
};
```

4.1.7 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is, we cannot redefine the plus(+) operator to subtract one value from another.
4. Overloaded operators follow the syntax rules of the original operators. That cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use **friend** functions to overload certain operators. However, member Functions can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values. But, those overloaded by means of a friend function take one reference argument.
8. Binary operators overloaded through a member function take one explicit Argument and those which are overloaded through a friend function take two explicit arguments.

9. When using binary operators overloaded through a member function, the left- hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +,-.*, and / must explicitly return a value. They must not attempt to change their own arguments.

4.1.8 Type Conversions

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable to the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert **x** to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types. Consider the following statement that adds two objects and then assigns the result to a third object.

```
V3 = V1 + V2;          //V1, V2 and V3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. Therefore, we must design the conversion routines by ourselves, if such operations are required. Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from built-in type to class type.
2. Conversion from class type to built-in type.
3. Conversion from one class type to another class type.

Basic to Class Type

The conversion from basic type to class type is east to accomplish. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type. Let us consider an example of converting an ***int*** type to a ***class*** type.

```
class time
{
        int hrs;
        int mins;
    public:
        …..
        …..
        time(int t)             //constructor
        {
        hours = t / 60;         //t in minutes
        mins = t % 60;
        }
};
```

The following conversion statements can be used in a function:

```
time = T1;              //object T1 created
int duration = 85;
T1 = duration;          //int to class type
```

After this conversion, the **hrs** member of **T1** will contain a value of 1 and **mins** member a value of 25, denoting 1 hour and 25 minutes. Note that the constructors used for the type conversion take a *single* argument whose type is to be converted.

Class to Basic Type

The constructor functions do not support this operation. C++ allows us to define an overloaded *casting* operator that could be used to convert a class type data to a basic type.The general form of an overloaded casting operator function, usually referred to as a *conversion* function, is:

```
operator typename( )
{
        …..
        ….. (Function statements)
        …..
    }
```

This function converts a class type data to *typename*.For example, the **operator double()** converts a class object to type **double**, the **operator int( )** converts a class type object to type **int**, and so on. Consider the following conversion:

```
Vector :: operator double( )
{
        double sum = 0;
        for(int i=0; i<size; i++)
                sum=sum+v[i] * v[i];
        return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. The operator **double( )** can be used as follows:

```
double length = double(V1);
        or
double length = V1;
```

where **V1** is an object of type **vector**. Both the statements have exactly the same effect.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

One Class to Another Class Type

There are situations to convert one class type data to another class type. Example:

objX = objY; //objects of different types

objx is an object of class X and obj Y is an object of class Y. The class Y type data is converted to class X type data and the converted value is assigned to the objX. Since the conversion takes place from class Y to class X, Y is known as the source calss and X is known as the destination class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be located, in the source class or in the destination class.

We know that the casting operator function

Operator typename ()

Converts the class object of which it is a member to typename. The typename may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for conversion. This makes it necessary that the conversion constructor be placed in the destination class.

A Data Conversion Example

Let us consider an example of an inventory of products in a store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in program uses two classes and show how to convert data of one type to other.

```
#include  <iostream.h>
//class invent2
Class invent1
{
   Int code;
   Int items;
   Float price;
Public:
  Invent1(int a, int b, float c)
{
 Code = a;
 Items = b;
Price =  c;
}
Void putdata()
{
   Cout << “code:” << code << “\n”;
   Cout << “items: “ << items << “\n”;
   Cout  << “value: “ << price << “\n”;
}
Int getcode() (return code;)
Int getitems() (return items;)
Int getprice() (return price;)
Operator float () (return (items * price) ;)
/* operator invent2() // invent to invent2
{
 Invent2 temp;
Temp.code = code;
Temp.value = price * items;
Return temp;
} */
}; // end of source class

Class invent2 // destination class
{
Int code;
Float value;
Public:
 Invent2()
{ code = 0; value = 0;
Invent 2(int x, float y)
{ code = x; value y;
Void putdata()
{
```

```
Cout << “code : “<< code << “\n”;
Cout << “value : “<< value << “\n\n”;
}
Invent2(invent1 p)
{
Code = p.getcode( );
Value = p.getitems( ) * p.getprice( );
}
};
Main( )
{
Invent1 s1(100,5, 140.0);
Invent2 d1;
Float total_value;

Total_value = s1;
D1= s1;
Cout << “product details – invent1 type” << “\n “;
S1. putdata ( );
Cout << “\n stock value “ << “\n”;
Cout << “value = “ << total_value << “\n\n”;
Cout << “product details – invent2 type” << “\n”;
D1.putdata ( );
}
```

Following is the output of program:

```
        Product details – invent type
Code: 100
Items : 5
Value : 140
Stock value
Value : 700

Product details – invent2 type
Code : 100
```

Value: 700

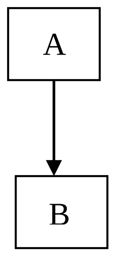
It is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

4.2.0 Inheritance: Extending Classes

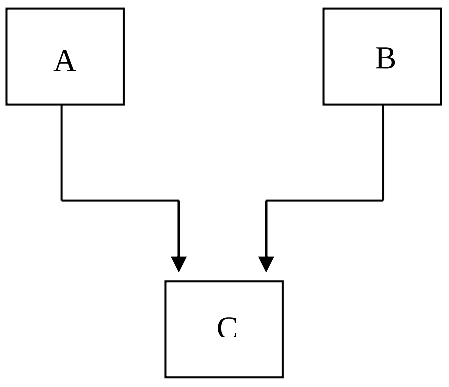
4.2.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

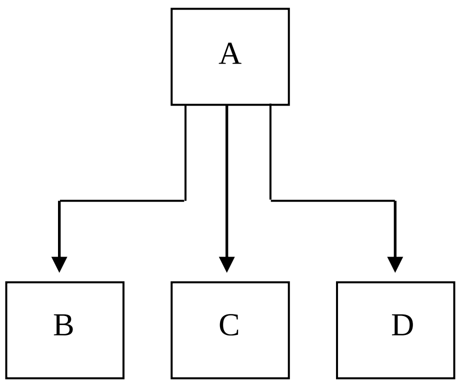
The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the bass class and the new one is called the derived class or subclass. The derived class inherits some or all of the traits form the bass class. A class can also inherit properties form more than one class or form more than one level. A derived class with only one base class, is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. Figurer 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".



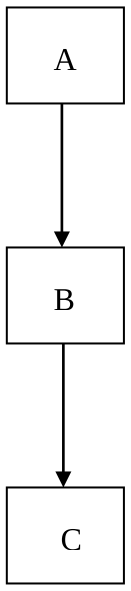
(a) Single inheritance



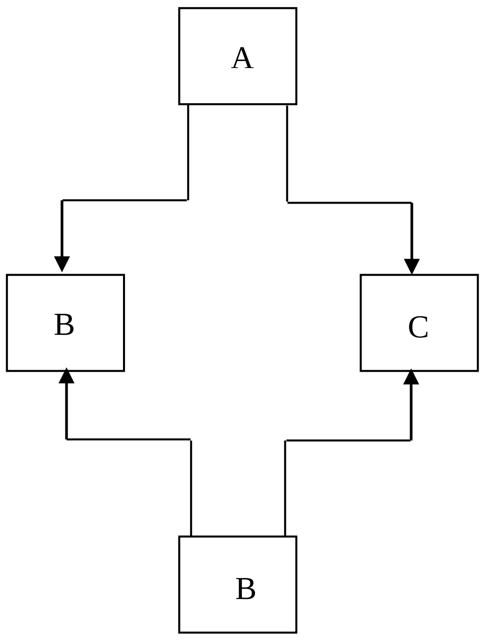
(b) Multiple inheritance
inheritance



(c) Hierarchical



(d) Multilevel inheritance



(e) Hybrid inheritance

4.2.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is :

```
Class derived –class –name      : visibility – mode base – class – name
{
        …..//
        …..// members of derived class
        …..//
};
```

The colon indicates that the derived – class- name is derived from the base – class – name. The colon indicates that the derived – class – name is derived from the base – class – name. the visibility – mode is optional and, if present, may be either private or public. The default visibility – mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Example :

```
class ABC: private XYZ              // private derivation
{
        members of ABC
};
class  ABC: public XYZ              // public derivation
{
        members of ABC
};

Class ABC: XYZ                      // private derivation by default
{
        members  of ABC
};
```

4.2.3 Single Inheritance

A base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

Single Inheritance: Public

```
#include <iostream>
using namespace std;
class B
{
        Int a;                      //private; not inheritable
Public:
        Int b;                      // public; ready for inheritance
        void get_ab();
        int get_a(void);
        void show_a(void);
};
Class D : public B                  // public derivation
{
        int  c;
   public:
        void mul(void);
        void display (void);
};
//--------------------------------------------------------------------
Void B :: get_ab(void)
{
        a = 5; b = 10;
}
int B :: get_a()
{
        Return a;
}
Void B :: show_a()
{
        Cout << ”a = ” << “\n”;
}
void D :: mul()
{
        c  = b * get_a();
```

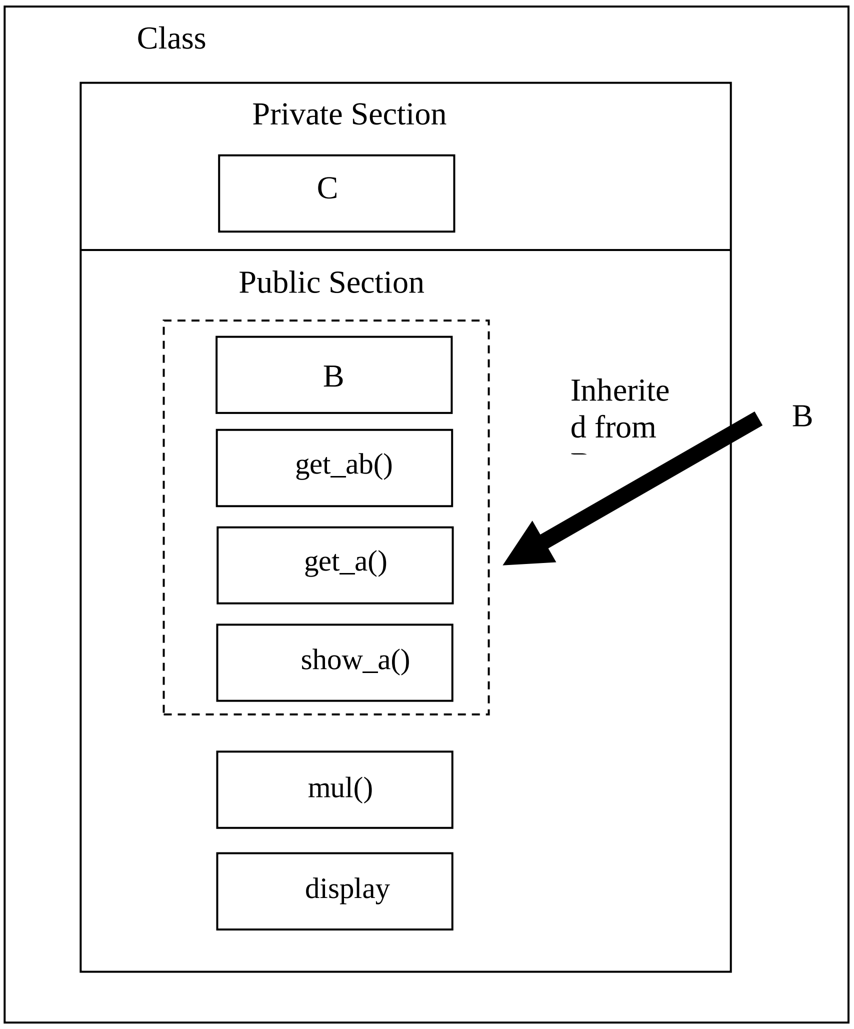
```
          }
          Void D :: display ()
          {
                    cout << “ a = “ << get_a() << “\n”;
                    cout  << “b = “ <<  b << “\n”;
                    cout << “ c = “ << c << “\n\n”;
          }
          //-----------------------------------------------------------------
int  main ()
{
          D d;
          d.get_ ab();
          d.mul();
          d.show_a();
          d.display();

          d.b = 20;
          d.mul ();
          d.display ();
          return 0;
}
```

Given below is the output of program

```
a = 5
a = 5
b = 10
c = 50
a = 5
b = 20
c = 100
```

The class D is a public derivation of the base class B. Therefore, D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is also a public member of the derived class D. The private members of B cannot be inherited by D.



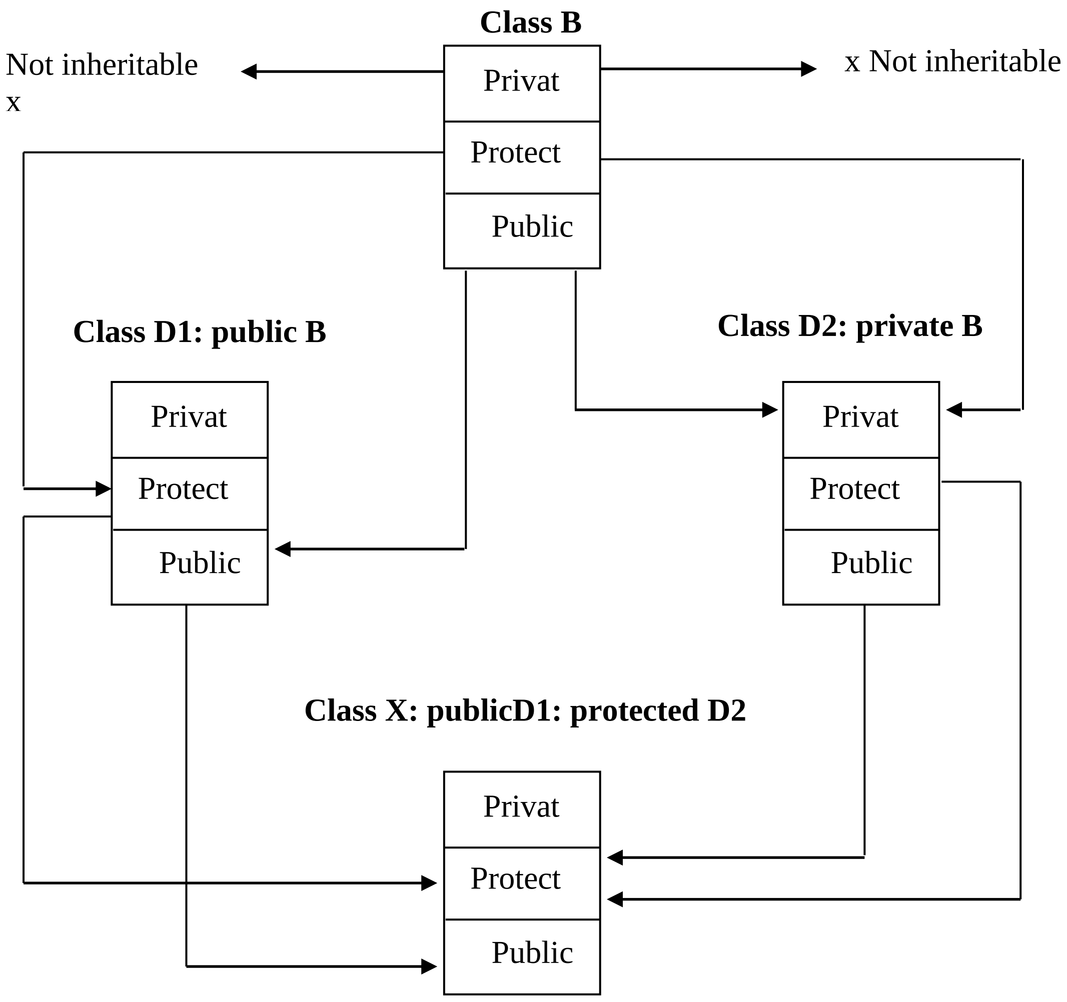
4.2.4 Making a Private Member Inheritable

What do we do if the private data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the private member by making it public. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, protected, which serve a limited purpose in inheritance.

```
class  alpha
{
        private  :                      // optional
                …..                     // visible to member functions
                …..                     // within its class
        protected:
            …..                     // visible to member functions
            …..                     // of its own and derived class
        public:
            …..                     // visible  to all functions
            …..                     // in the program
};
```

When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. The pictorial representation for two levels of derivation



Effect of inheritance on the visibility of member

The keywords private, protected, and public may appear in any order and any number of times in the declaration of a class. For example,

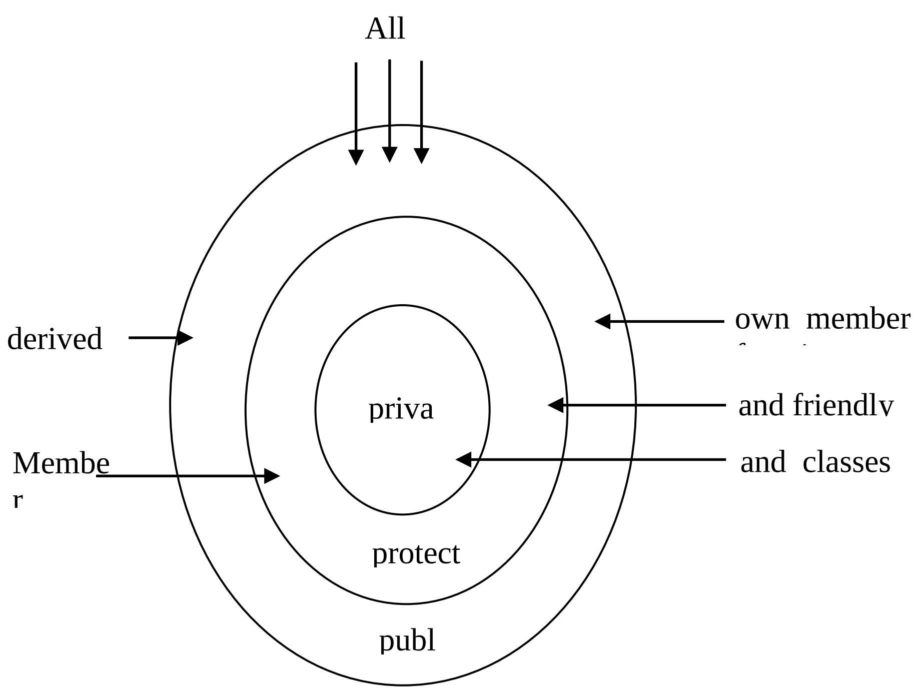
```
class  beta
{
        Protected:
                …….
        public:
                …….
        private:
                …….
        public:
                ……..
};
```
is a valid class definition.

However, the normal practice is to use them as follows:

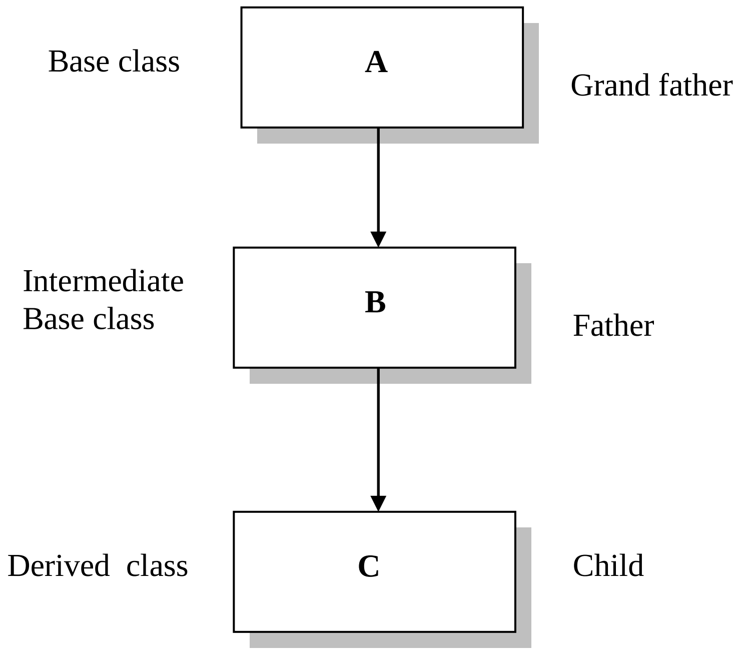
```
class beta
{
        ………                        // private by default
        ………
protected :
        ………
public:
        ………
}
```
It is also possible to inherit a base class in protected mode (known as protected derivation). In protected derivation, both the public and protected members of the base class become protected members of the derived class.

4.2.5 Multilevel Inheritance

The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. the class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.



A simple view of access control to the members of a class



Multilevel inheritance

A derived class with multilevel inheritance is declared as follows:

Class A {……}; // base class

```
Class B: public A {……..};              // B derived from A
Class C: public B {……..};              // C derived from B
```

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class student stores the roll- number, class test stores the marks obtained in two subjects and class result contains the total marks obtained in the test. The class result can inherit the details of the marks obtained in the test and the roll- number of students through multilevel inheritance. Example:

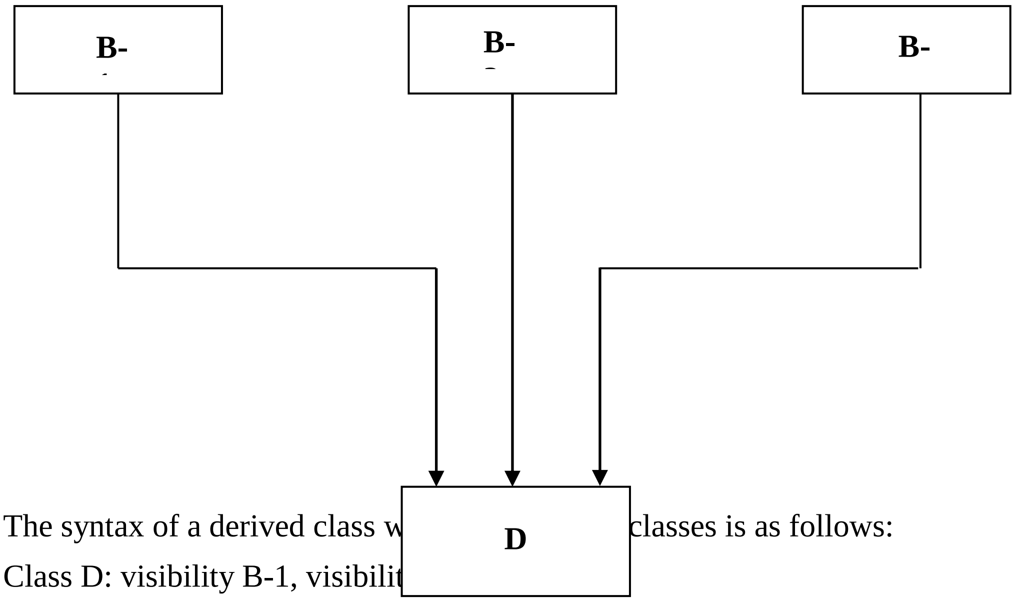
```
Class student
{
        Protected;
        Int roll_number;
Public:
        Void get_number(int);
        Void put_number(void);
};
Void student : : get_number(int a)
{
        Roll_number = a;
}
Void student : : put_number()
{
        Cout << “Roll  Number : “ << roll_number << ”\n”;
}
Class test : public student                    // first level derivation
{
        Protected :
                Float sub1;
                Float sub2;
        Public :
                Void get_marks(float, float);
                Void put_marks(void);
};
Void test : : get_marks(float x, float y)
{
        Sub1 = x;
        Sub2 = y;
}
Void test : : put_marks ()
{
        Cout << “marks in SUB1 = “ << sub1 << “\n”;
        Cout << “marks in SUB2 = “ << sub2 << “\n”;
}
Class result : public test                  // second level derivation
{
        Float total;                        // private by default
  Public:
        Void display (void);
};
Void result : : display (viod)
{
        Total = sub1 + sub2;
        Put_number();
        Put_marks ();
        Cout << “Total  =  “<< total << “\n”;
}
Int main ()
{
        Result student1 ;                   // student1 created
        Student1.get_number (111);
        Student1.get_marks (75.0, 59.5);
        Student1.display();
        Return  0;
}
```

Display the following output:

```
Roll Number : 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

4.2.6 Multiple inheritance

A class can inherit the attributes of two or more classes. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.



The syntax of a derived class w classes is as follows:

Class D: visibility B-1, visibilit

```
{
        ……
        …… (Body of D)
        ……
};
```

Where, visibility may be either public or private. The base classes are separated by commas.

Example :

```
Class P: public M, public N
{
        Public :
          Void display (void);
};
```

Classes M and N have been specified as follows:

```
Class M
{
        Protected :
                int m;
        public :
                void M :; get_m(int);
};
Void M : : get_m(int x)
{
        m= x;
}
Class N
{
        Protected:
                int  n;
        public:
                void get_n(int);
};
Void N : : get_n(int y)
        {
                n  = y;
}
```

The derived class P, as declared above, would, in effect, contain all the members of M and N in addition to its own members as shown below:

```
        Class P
        {
                Protected :
                        int  m;                         // from M
                        int  n;                         // form N
        public:
                void get_m (int);                       // form M
                void get_n (int);                       // from N
                void display(void);                     //own member
        };
```

The member function display () can be defined as follows:

```
        Void P : : display (void)
        {
                Cout << “m = “ << m << “\n”;
                Cout << “n = “ << n  << “\n”;
                Cout << “m*n  = “ <<m*n << “\n”;
        };
```

The main () function which provides the user-interface may be written as follows:

```
        Main()
```

MULTIPLE INHERITNACE

```
# include <iostream>
Using namespace  std;
Class M
{
        protected :
                int n;
        public :
                void get_m(int);
};
Class N
{
        Protected:
                Int n;
        Public :
                Void get_jn(int);
};
Class P : public M, public N
{
        public:
                void display (void);
};
void M :: get_m(int x)
{
        m=x;
}
void N :: get_n(int y)
```

```
{
        n=y;
}
void P :: display (void)
{
        cout <<” m=”<< m<<”\n”;
        cout <<”  n=” <<  m << ”\n”;
        cout <<”m* n=” <<  m *n << ”\n”;
}
int main ( )
{
Pp;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```

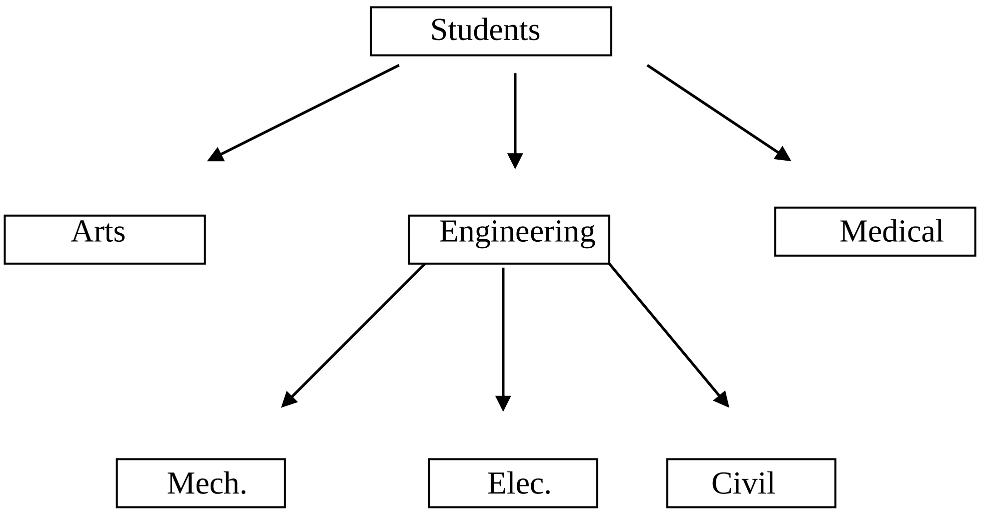
The output of Program 8.4 would be;

```
        m = 10
        n = 20
        m*n = 200
```

4.2.7 HIERARCHICAL INHERITANCE

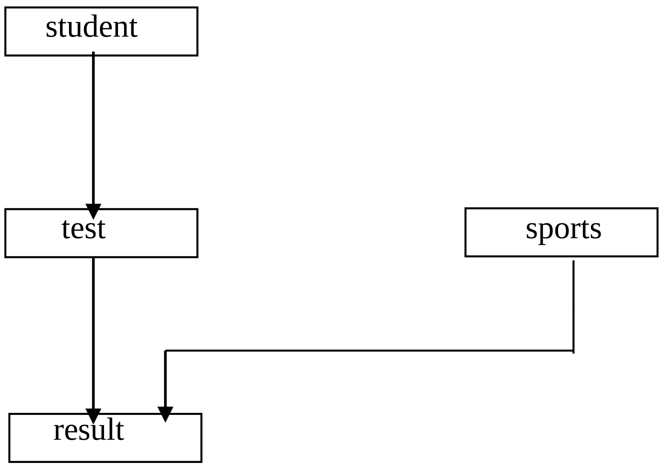
The other application of inheritance is to use as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

The below figure a hierarchical classification of students in a university.



4.2.8 HYBRID INHERITANCE

Assume that we have to give weight age for sports before finalizing the results. The weight age for sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in Fig.



The sports class might look like :

```
class sports
{
        protected:
                float score;
```

```
          public :
                    void get_score(float);
                    void put_score(void);
};
```

The result will have both the multilevel and multiple inheritance and its declaration would be as follows:

```
          class result      :          public test, public sports
          {
                    ……………
                    ……………
          };
```

Where test itself is a derived class from student. That is

```
          class test        :          public student
          {
                    ………….
                    ………….
          }
```

Program 8.5 illustrates the implementation of both multilevel and multiple inheritance.

HYBRID INHERITANCE

```
          # include <iostream>
          Using namespace std;
          class  student
          {
                    protected:
                              int roll_number ;
                    public:
                              void get_number ( int a)
                              {
                                        Roll_number = a;
                              }
                              void put_number(void)
                              {
```

```
                                        cout<<”Roll No: <<roll_number <<”\n”:
                              }
};
Class test          :         public student
{
          protected :
                    float part1, part2;
          public ;
                    void get_marks(float x, float y)
                    {
                              part1 = x; part2 -= y;
                    }
                    void put_marks(void)
                    {
                              cout << “ Marks obtained:”<<”\n”
                                    << “part1 = “<< part1 << “\n”
                                    << “part2 =”<<part2<<”\n”;
                    }
}
class sports
{
          protected:
                    float score;
          public:
                    void get_score(float s)
                    {
                              score = s;
                    }
                    void put_score (void)
                    {
                              cout<< “Sports wt: “ <<scire <<”\n\n”;
                    }
};
Class result        :         public test, public sports
```

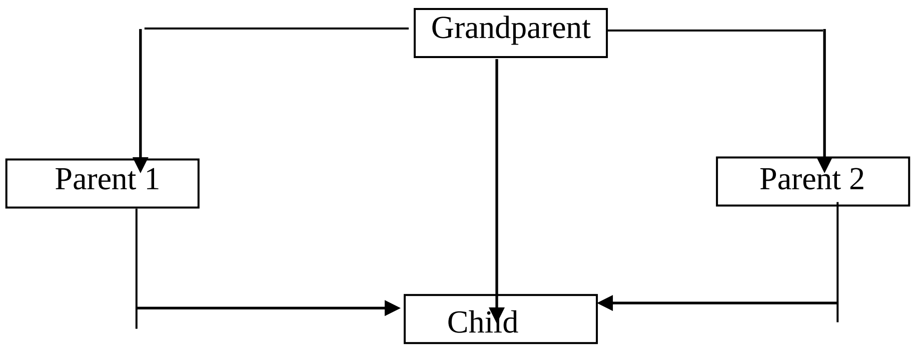
```
{
        float total;
  public
        void display (void);
};
void  result     :: display (void)
{
        total = part1 + part2 = score ;
        put _number ( );
        put_marks ( );
        put_score ( );
        cout << ”Total Score :”<< total << “\n”;
}
int  main ( )
{
        result student_1;
        student_1.get_number(1234);
        student_1.get_marks(27.5, 33.0);
        student_1.get_score(6.0);
        student_1.display( );
        return 0;
}
```

Here is the output of Program 8.5

```
Roll No: 1234
Marks Obtained :
Part1 = 27.5
Part2 = 33
Sports wt :6
Total Score : 66.5
```

4.2.9 VIRTUAL BASE CLASSES

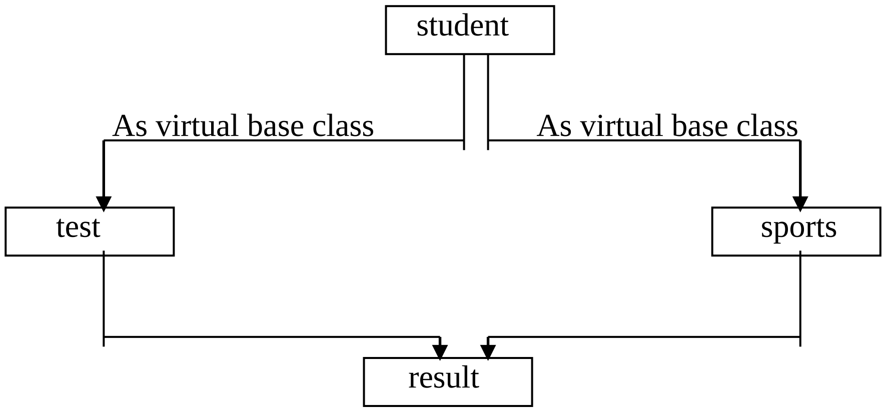
Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated the child has two direct base classes ‘parent1’ and ‘ parent 2’ which themselves hava a common base class ‘ grandparent’. The child inherits the traits of ‘grandparent ‘ vin two separate paths. It can also Inherit directly as shown by the broken line. The grandparent is sometimes referred to as indirect base class.



The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below :

```
Class A                                          // grandparent
{
      ………..
      ……….
};
Class B1      : virtual public A                 // parent 1
{
      …………
      ………...
};
Class B2 : public virtual A                      // parent 2
{
      ………..
      ……….
};
class C : public B1, public B2                 // child
{
      ……..                                 // only one copy of A
      …….                                  // will be inherited
};
```

When a class is made a Virtual base class, C++ takes necessary care to see that only see copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.



A program to implement the concept of virtual base class is illustrated in program

Virtual Base Class

```
# include <iostream>
using namespace std;
class student
{
        protected;
                int  roll_number;
        public :
                void get_number ( int a)
                {
                        roll_number = a;
                }
                void put_number(void)
                {
                        cout << “Roll No : << roll_number <<”/n”;
                }
};
class test : virtual public student
{
```

```
          protected :
                    float part, part2;
          Public :
                    void get_marks (float x, float y)
                    {
                              part1 = x; part2=y;
                    }

                    void put_marks(void)
                    {
                              cout<<"Marks obtained  :"<<"\n'"'
                              << "part1="<<part1<<"\n"
                              <<"part2="<<oart2<<"\n";
                    }
};
 class sports : public virtual student
{
          protected:
                    float score :
          public :
                    void get_score (float s)
                    {
                              Score = s;
                    }
                    void put_score(void)
                    {
                              cout<< "Sports wt: "<score<<"\n\n";
                    }
};
class result :public test, public sports
{
                    float total;
          public :
                    void display (void);
```

```
};
void result ::display (void)
{
        total = part1 + part2 +score;
        put_number ();
        put_marks();
        put_score();
        cout<< “total score : “ << total <<”\n”
}
int main ()
{
        result student _1;
        student_1.get_number(678);
        student_1.get_marks(30.5, 25.5);
        student_1.get_score (7.0);
        student_1 .display ();
        return 0:
}
```

The output of Program 8.6 would be

```
        Roll No : 678
        Marks obtained :
        Part1 = 30.5
        Part2 = 25.5
        Sport wt: 7
        Total Score : 63
```

4.2.10 ABSTRACT CLASSES

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built. In the previous example, the student class is an abstract class since it was not used to create any objects.

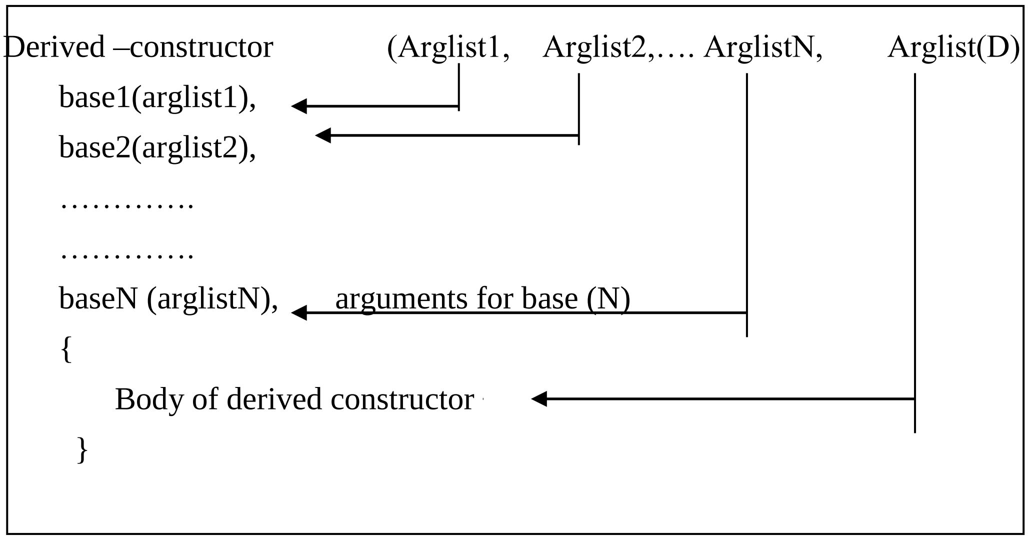
4.2.11 CONSTRUCTOS IN DERIVED CLASSES

The Constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor take any arguments , the derived class need not have a constructor function.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

The constructor of the derived class received the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is :


Derived –constructor (Arglist1, Arglist2,…. ArglistN, Arglist(D)
base1(arglist1),
base2(arglist2),
…………
…………
baseN (arglistN),
arguments for base (N)
{
Body of derived constructor
}

The header line of derived constructor function contains two parts separated by a colon(;). The first part provides the declaration of the arguments that are passed to the derived – constructor and the second part lists the function calls to the base constructors.

CONSTRUCTORS IN DERIVED CLASS

```
# include <iostream>
using namespace std;
class alpha
{
                int x;
        public :
                alpha ( int i)
                {
                        x = i;
                        cout << “alpha initialized \n”
                }
                void show x(void)
                {  cout <<”x=”<<x<<”\n”;
                }
};
Class beta
{
                float y;
        public ;
                beta (float j)
                {
                        y = j;
                        cout << ”beta initialized \n”;
                }
                void show_y (void)
                { cout << “y=”<<y<<”\n”;}
};
Class gamma : public beta, public alpha
{
                int m, n;
        public ;
                gama ( int a, float b, int c, int d);
                        alpha (a), beta(b);
                {
                        m = c;
                        n = d;
                        cout<< “gamma initialized \n “;
                }
                void show_mn(void)
                {
                    cout << “m=<<m<<”\n”
                          <<”n=”<<n<<”\n”;
            }
};
int main()
{
        gamma g(5,10.75, 20, 30);
        cout <<”\n”
        g.show_x();
        g.show_y();
        g.show_mn();
        return();
}
```

The output of Program 8.7 would be

```
        beta initialized
        alpha initialized
        gamma  initialized
        x = 5
        y = 10.75
        m = 20
        n = 30
```

Program8.8 illustrates the use of initialization lists in the base and derived constructors.

INITIALIZATION LIST IN CONSTRUCTORS

```
# include ( iostream>
using namespace std;
class alpha
{
                int x;
        public :
                alpha ( int i)
                {
                        x = i;
                        cout << “\n alpha constructor”;
                }
                void show_alpha(void)
                {
                        cout<<"x"<<x<<"\n":
                }
};
class  beta
{
                float p,q;
        public :
                beta (float a, float b ) : p(a), q(b+b)
                {
                        cout <<"\n beta constructed ';
                }
                void show _beta(void)
                {
                        Cout <<"p="<<p<<"\n";
                        Cout <<"q="<<q<<"\n";
                  }
};
Class gamma : public beta, public alpha
{
                int u,v;
        public:
                gamma ( int a, int b, float c);
                alpha (a*2), beta (c,c) u(a)
                { v=b; cout <<"\n gamma constructed";}
                void show_gamma(void)
                {
                        cout<<"u="<<u<<"\n";
                        cout<<"v="<<v<<"\n";

                }
};
```

```
int main ()
{
        gamma g( 2,4, 2.5);
        cout <<”\n\n Display member values “<<”\n\n”;
        g.show_alpha ();
        g.show_beta ();
        g.show_gamma ();
        return 0;
};
```

The output of Program 8.8 would be:

```
beta constructed
alpha constructed
gamma constructed
Display member values
x = 4
p = 2.5
q = 5
u = 2
v = 4
```

4.2.12 MEMBER CLASSES : NESTING OF CLASSES

C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below :

```
        class alpha ( ……);
        class beta (…….);
        class gamma
        {
                alpha a;                // a is an object of alpha class
                beta b;                 // b is an object of beta class
                ………
        };
```

All objects of Gamma class will contain the objects a and b. This kind of relationship a called containership or nesting.

Example :

```
Class gamma
{
        ……..
        alpha a:                    // a is object of alpha
        beta b;                     // b is object of beta
        public :
                gamma(arglist); a(arglist1), b(arglist2)
                {
                        // constructor body
                }
};
```

arglist is the first of arguments that is to be supplied when a gamma object is defined. These parameters are used for initializing the members of gamma. arglist 1 is the argument list for the constructor of a and arglist2 is the argument list for the constructor if b. arglist1 and agrlist2 may or may not use the arguments from arglist remember, a(arglist 1) and b(arglist 2) are function calls and therefore the arguments do not contain the date types. They are simply variables or constants.

Exmaple :

```
gamma ( int x, int y, float z) : a(x), b(x,z)
{
        Assignment section ( for ordinary other members)
}
```

4.3.0 POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

4.3.1 Introduction

Polymorphism is one of the crucial features of OOP. It simply means ‘one name, multiple forms’. We have already seen how the concept of polymorphism is implemented using the overloaded member functions are ‘selected’ for invoking be matching arguments, both type and number.

4.3.2 Pointers

Pointers is one of the key aspects of C++ language similar to that of C. As we know, Pointers offer a unique approach to handle data in C and C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable’s memory address rather than data. A pointer variable defined where to get the value of a specific data variable instead of defining actual data.

4.3.3 Pointers and Objects

We are already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

Where item is a class and x is an object defined to be of type item. Similarly we can define a pointer it_ptr of type item as follow:

```
        Item * it_ptr;
```

Object pointers are useful in creating objects at run time, We can also use an object pointer to access the public members of an object. Consider a class item defined as follows :

```
        class item
        {
                int code;
                float price;
        public:
                void getdata (int a, float b)
                {
                        code = a;
                        price = b;
                }
                void show (void)
                {
                        cout<<"Code:"<<code<<"\n";
                             <<"price:"<<price<<"\n\n";
                }
};
```

Let us declare an item variable x and a pointer ptr to x as follows:

```
            item x;
            item *ptr=&x;
```

The Pointer ptr is initialized with the address of x.

We can refer to the member functions of item in two ways, one by using the dot operator and the object, and another by using the arrow operator and the object pointer. The statements

```
x.getdata ( 100,75.50);
x.show();
```

are equivalent to

```
ptr -> getdata (100, 75.50);
ptr -> show ();
```

Since " ptr is an alias of x, we can also use the following method;

```
( *ptr.show();
```

The parentheses are necessary because the dot operator has higher procedure than the indirection operator ".

POINTERS TO OBJECTS

```
# include <iostream>
using namespace std;
class item
{
        int code;
        float price;
        public :
          void getdata (int a, float b)
         {
               code = a;
               price = b;
          }
          void show ( void )
          {
               cout<<"Code:"<<code<<"\n";
               cout<<"Price:"<<Price<<"\n";
          }
};
const int  size = 2;
int main ()
{
        item *p=new item (size};
        item *d = p;
        int x; i;
        float y;
        for ( i=0; i<size; i++)
        {
                cout << “Input code and price for item” << i+1;
                cin >> x >> y;
                p-> getdata (x,y);
                p++;
        }
        for (i=0; i<size; i++)
        {
                cout<< “Item:”<< i+1 << “\n”;
                d-> show ();
                d++;
        }
        return 0 ;
}
```

The output of Program 9.8 will be

```
Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code : 40
Price : 500
Item : 2
Code : 50
Price : 600
```

ARRAY OF POINTERS TO OBJECTS

```
# include <iostream.h>
#include <cstring>
using namespace std;
class city;
{
        protected:
                char *name;
                int len;
        public :
                city ()
                {
                        len = 0;
                        name = new char {len+1};
                }
                Void getname(void)
                {
                        char *s;
                        s = new char (30);
                        cout<<"Enter city name :";
                        cin>>s;
                        len = strlen(s);
                        name = new char (len +1);
                        strcpy (name, s);
                }
                void printname (void)
                {
                        cout<< name<<"\n";
                }
};
int  main ()
{
        city *cptr [10];                        //  array  of  10  pointers  to  cities
        int n=1;
        int option;
        do
        {
                cptr[n] = new city;             // create new city
                cptr[n]->getname();
                n++;
                cout<<"Do you want to enter one more name?\n";
                cout<<"(Enter 1 for yes 0 for no):";
                cin>> option;
        }
        while (option);
        cout<<"\n\n";
        for(int i = 1; i <=n; i++)
        {
                cptr[i] -> printname ();
        }
        return 0;
};
```

The output of program 9.9 would be:

```
        Enter city name : Hyderabad
        Do you want to enter one more name ?
        (Enter 1 for yes 0 for no);1
        Enter city name : Secunderabad
        Do you want to enter one more name ?
        (Enter 1 for yes 0 for no);1
        Enter city name : Malkajgiri
        Do you want to enter one more name ?
        (Enter 1 for yes 0 for no);0
        Hyderabad
        Secunderabad
        Malkajgiri
```

4.3.4 This Pointer

C++ uses a unique keyword called this to represent an object that invokes a member function, this is a pointer that points to the object for which this function was called. For example, the function call A.max() will set the pointer this to the address of the object.A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to the member function when it is called. The pointer this acts as an implicit argument to all the member functions. Consider the following simple example:

```
        class ABC
        {
                int  a;
                …….
                …….
        };
```

The private variable ‘a’ can be used directly inside a member function, like

```
        a=123  ;
```

We can also use the following statement to do the sane job:

```
        This->a 123;
```

Since C++ permits the use of shorthand form a=123, we have not been using the pointer this explicitly so far. However, we have been implicitly using the pointer this when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer this. One important application of the pointer this is to return the object it points to. For example, the statement

```
        return * this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a a member function and return the invoking as a result.

This Pointer

```
# include <iostream>
# include <cstring>
using namespace std;
class person
{
                char name [20];
                float age ;
        public :
                person (char *s, float a)
                {
                Strcpy(name, s);
                age= a;
                }
                person & person ::greater (person & x)
                {
                        If(x.age>=age)
                                return x;
                        else
                                return *this;
                }
                void display(void)
                {
                        cout << "Name:"<<name<< "\n"
                              <<Age :"<<age<<"\n";
                }
};
int main ()
{
        person P1("John", 37;.50),
                P2(:Ahmed:", 29.0),
                P3("Hebber",40.25);

        person P = P1.greater (P3);             // P3.greater (P1)
        cout<<"Elder person is :\n";
        p.dispaly();

        return 0;
}
```

The output of Program 9.10 would be:

```
        Elder person is:
        Name : Hebber
        Age : 40.25
        Elder person is:
        Name : John
        Age : 37.5
```

4.3.5 POINTERS TO DERIVED CLASSES

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable ca be made to point to objects belonging to different classes. For Example, if B is a base class and D is a

Derived class from B, then a pointer declared as a pointer to B can also be a pointer to D. Consider the following declarations:

```
    B * cptr;                 // pointer to class B type variable
    B b;                      // base object
    D d;                      // derived object
    Cptr = &b;                // cptr points to object b
```

We can make cptr to point to the object d as follows:

```
    Cptr = &d;                // cptr points to object d
```

This is perfectly valid with C++ because d is an object derived from the class B.

However, there is problem in using cptr to access the public members of ther derived class D. Using cptr, we can access only those members which are inherited from b and not the members that originally belong to D. In case a member of D has the same name as one of the members of B, then any reference to that member by cptr will always access the base class member.

POINTERS TO DERIVED OBJECTS

```
#include <iostream>
using namespace std;
class BC
{
        public
                int b;
                void show ()
                {  cout <<" b = << b << "\n"; }
};
Class DC : public BC
{
        public :
                ind d;
                void show ()
                { cout << "b="<<b<<"\n"
```

```
                              <<"d="<<d<<"\n";
                    }
};
int main ( )
{
          BC * bptr ;                              // base pointer
          BC base;
           bptr = & base;
          bptr->b=100;                             // access BC via base pointer
          cout<<"bptr points to base object \n";
          bptr ->show ();
          // derived class
          DC derived;
          Bptr = & derived                // address of derived object
          bptr-> b=200;                   // access DC via base pointer
          /* bptr ->d = 300;*/            // won't work
          cout<< "bptr now pointers to derived object \n";
          bptr ->show();                  // bptr now points to derived object
          /* accessing a using a pointer of type derived class DC */
          DC *dptr;
          dptr =&derived
          dptr->d=300;
          cout <<"dptr is derived type pointer\n";
          dptr ->show();
          (( DC *)bptr)->d=400;
          ((DC*)bptr ->show ();
          return 0;
}
```

Program 9.11 produces the following output :

```
          bptr pointers base object
          b=100
          bptr now points to derived object
          b=200
          bptr is derived type pointer
```

```
        b=200
        d=300
        using ((DC*)bptr)
        b=200
        d=400
```

4.3.6 VIRTUAL FUNCTIONS

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object to be the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function. Program 9.12 illustrates this point.

```
# include <iostrean>
using namespace std;
class Base
{
        public :
                void display () { cout <<"\n Display base "; }
                virtual void show () {cout <<"\n show base "; }
};
Class Derived : public Base
{
        public :
                void display () { cout <<"\n Display base "; }
                virtual void show () {cout <<"\n show base "; }
};
int main ()
{
                base B;
                Derived D;
                Base * bptr;

                cout <<"\n bptr points to base \n";
                bptr =&B;
        bptr-> display ();              // calls Base version
        bptr-> show ();                 // calls Base version

        cout <<"\n bptr points to Derived \n";
        bptr =&D;
        bptr-> display ();              // calls Base version
        bptr-> show ();                 // calls Derived version

        return 0;
}
```

The output of Program 9.12 would be:

```
bptr points to Base
Display base
Show base
bptr points to Derived
Display base
Show derived
```

RUNTIME POLYMOTPHISM

```
# include <iostream>
#include <cstring>
using namespace std;
class meida
{
    protected :
        char title [50];
        float price;
    public :
        media (char*s, float a)
        {
            Strcpy (title,s);
            price = a;
        }
        virtual void display() { } // empty virtual function
}:
```

```
class book; public meida
{
                int pages;
        public :
                book (char*s, float a, int p) :meida (s,a)
                {
                        pages = p;
                }
                void display ();
};
class tape : public media
{
                float time ;
        public :
                tape(char*s, float a, float t):media (s,a)
                {
                        time = t;
                }
                void display();
};
void  book ::display ()
{
        cout<< “\n Title :”<< title;
        cout<< “\n Pages :”<< pages;
        cout<< “\n Price :”<< price;
}
void tape ::display ()
{
        cout<< “\n Title :”<<title;
        cout<< “\n Play time :”<<title << “mins”;
        cout<< “\n Price :”<< price;
}
int main ()
{
        char * title = new char [30];
        float price, time;
        int pages;
        // Book details
        cout <<"\n Enter book details \n"
        cout<< "\n Title :"<< title; cin>>title;
        cout<< "\n Pages :"<< pages; cin>> pages;
        cout<< "\n Price :"<< price; cin >> price ;
        book book1 (title, price, pages );
        // Tape details
        cout <<"\n Enter Tape details \n"
        cout<< "\n Title :"<< title; cin>>title;
        cout<< "\n Price :"<< price; cin >> price ;
        cout<< "\n Play time (mins)::"; cin >> time ;
        tape tape1(title, price, time);
        media * list [ 2];
        list [0] =&book1;
        list [1] =&tape1;
        cout << "\n MEDIA DETAILS ";
        cout <<"\n ………Book ……..";
        list[0] -> display ( );                 // display book details
        cout <<"\n ………Tabe ……..";
        list[1] -> display ( );                 // display tape details
        result 0;
}
```

The output of Program 9.13 would be

```
ENTER BOOK DETAILS
Title : Programming _in_basic
Price : 88
Pages : 400
ENTER BOOK DETAILS
Title : Computing_Concepts
Price : 90
Play time (mins):55
```

```
MEDIA DETAILS
……. BOOK ……….
ENTER BOOK DETAILS
Title : Programming _in_ANSI_C
Pages : 400
Price : 88
……… TAPE……..
Title : Computing_Concepts
Play time : 55mins
Price : 90
```

RULES FOR VIRTUAL FUNCTION

When virtual functions are created for implementing late binding. We should observe some basic rules that satisfy the compiler requirements.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

4.3.7 PURE VIRTUAL FUNCTIONS

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class in seldom used for performing any task. It only serves as a placeholder.

State whether the following statements are TRUE or FALSE:

a) Using the operator overloading concept, we can change the meaning of an operator.

b) Operator overloading works when applied to class objects only.

c) Friend functions cannot be used to overload operators.

d) The overloaded operator must have at least one operand that is user-defined type.

e) Through operator overloading, a class type data can be converted to a basic type data.

e) A constructor can be used to convert a basic type to a class type data.

Questions:

1. What is operator overloading?
2. Why is it necessary to overload an operator?
3. What is an operator function? Describe the syntax of an operator function.
4. What is a conversion function? How is it created? Explain its syntax.
5. How many arguments are required in the definition of an overload unary operator?

NOTES

UNIT-V

5.4.0 EXCEPTION HANDLING

UNIT-V

5.1.0 MANAGING CONSOLE I/O OPERATION

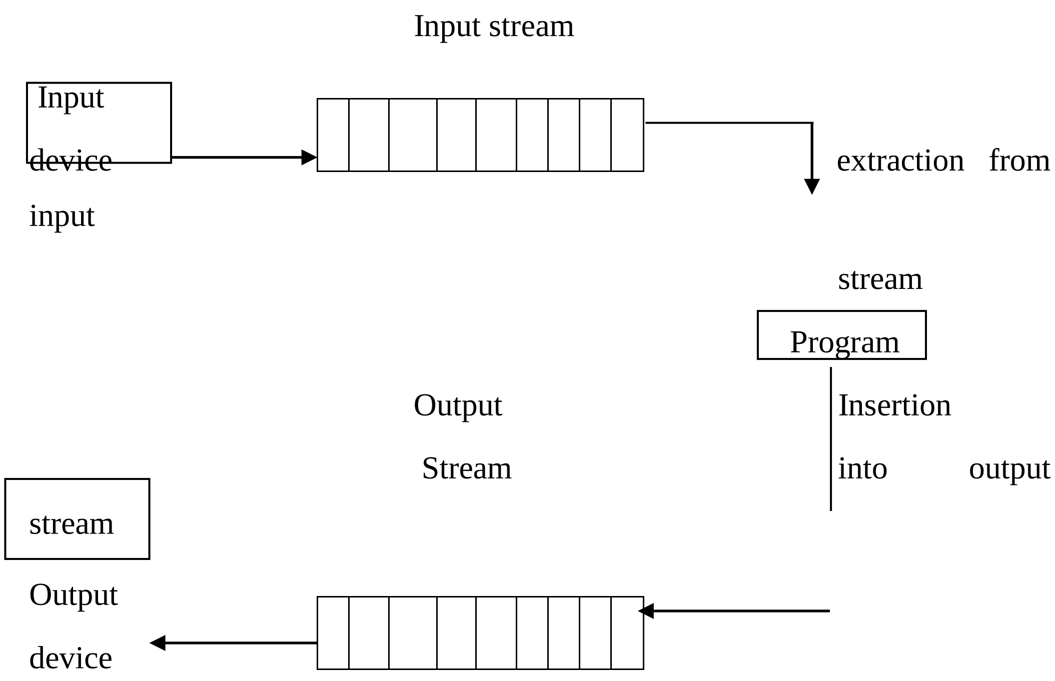
5.1.1 INTRODUCTION:

C++ uses the concept of stream and stream classes to implement i/o operations with console and disk files.

We will discus in this chapter how stream classes support the console oriented i/o operations.

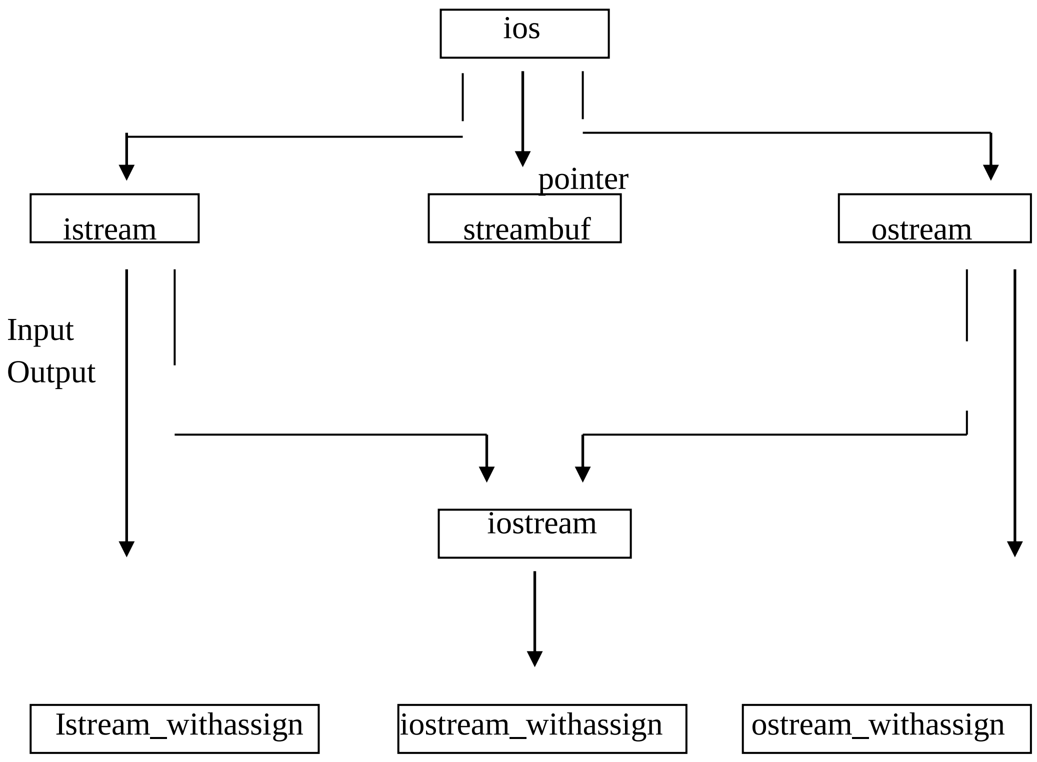
5.1.2 C++ STREAMS:

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. the source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream in other words a program extracts the bytes from an input stream and inserts bytes into an output stream.



5.1.3 C++ STREAM CLASSES:

The c++ i/o system contains a hierarchy of a classes that are used to define various streams to deal with both the console and disk files, these classes are called stream classes.



The class ios provides the basic support for formatted and unformatted i/o operations the class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance)provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams three classes, namely, isstream_withassign, ostream_withassign, and iostream_withassign add assignment operators to these classes.

5.1.4 UNFORMATTED I/O OPERATIONS:

Overloaded operators << and >>:

We have used the objects cin and cout for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic c++ types. The >> operators is overloaded in the istream class and << is overloaded in the ostream class. The following is the general format for reading data from the keyboard:

Syntax:

Cin >> variable > variable2 >>…..>>variableN

variable1,variable2,….. are valid c++ variable names that have been declared already this statement will cause the computer to stop the execution and look for input data from the keyboard.

The input data for this statement would be..

data1 data2 ……dataN

The input data are separated by white spaces and should match the type of variable in the cin list spaces, new lines and tabs will be skipped.

PUT() AND GET() FUNCTIONS:

The classes istream and ostream define two member function get() and put() respectively to handle the single character i/o operations. There are 2 types of get functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the new line character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character .since these functions are members of the i/o stream classes , we must invoke them using an appropriate object

EXAMPLE:

```
char c;
cin.get(c);
while(c!= ‘\n’)
{
  cout<<c;
cin.get(c);
}
```

This code reads and displays the line of text remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character.

GETLINE() AND WRITE():

We can read and display a line of text more efficiently using the line oriented i/o function getline() and write(). The getline() function reads a whole line of text that end with a newline character. This function can be invoked by using the objects cin as follows:

```
cin.getline(line, size);
```

This function call invokes the function getline() which reads character inputs into the variable line. The reading is terminated as soon as either the new line character ‘\n’ is encountered or size-1 character is read. The new line character is read but not saved. Instead it is replaced by the null character.

EXAMPLE:

```
 char name[20];
 cin.getline( name , 20);
```

5.1.5 FORMATTED CONSOLE I/O OPERATIONS

C++ supports a number of features that could be used for formatting the output., these features include:

- ios class functions and flags.
- Manipulators.
- User_defined output functions.
- The ios class contains a large number of member functions that would help us to format the output in a number of ways, the most important ones among them are listed in the table.

FUNCTION	TASK
width()	To specify the required size for displaying an output vale.
precision()	To specify the number of the number of digits to be displayed after the decimal point of a float value.
fill()	To specify a character that is used to fill the unused portion of a field.
setf()	To specify format flags that can control the form of output display.
unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the i/o statements to alter the format parameters of a stream.the table below show some important manipulator functions that are frequently used to access these manipulators, the file iomanip should be included in the program.

Manipulator	Equivalent ios function
setw()	width()
Setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
Resetiosflag()	unsetf()

In addition to these functions supported by the c++ library, we can create our own manipulator functions to provide any special output formats.

DEFINING FIELD WIDTH: Width ()

We can use the width() function to define the width of a field necessary for the output of a item. Since, it is a member function, we have used an object to invoke it, as shown below:

cout.width(w);

Where w is the field width. The output will be printed in a field of a character wide at the right end of the field. The width() function can specify the field width for only one item. After printing one item it will revert back to the default.

EXAMPLE:

cout. width(5);

cout << 543 << 12 <<'\n';

Will produce the output:

		5	4	3	1	2

SETTING PRECISION:PRECISION()

By default, the floating numbers are printed with six digits after the decimal point. However we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the precision() member function as follows:

cout.precision(3);

Where d is the number of digits to the right of the decimal point , for example the statements

cout.precision(3);

cout << sqrt(2) << "\n";

cout << 3.14159 << "\n";

cout <<2.50032 << "\n";

will produce the following output:

1.141 (truncated)

3.142 (rounded to the nearest cent)

2.5 (no training zeros)

Not that, unlike the function width(), precision() retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

FILLLING AND PADDING: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the fields are filled with white spaces, by default. However, we can use the fill() function to fill the unused positions by any desired character. It is used in the following form:

```
cout. fill (ch);
```

where ch represents the character which is used for filling the unused positions.

EXAMPLE:

```
cout.fill('*');
cout.width(10);
cout<< 5250 << "\n";
```

The output:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily like precision(), fill() stay in effect till we change it.

FORMATTING FLAGS, BIT-FIELDS AND SETF()

We have seen that when the function width() is used, the value is printed right-justified in the field width created. But it is a useful practice to print the text left-justified. The setf(), a member function of the ios class, can provide answers to these and many other formatting questions. The setf() function can be used as follows:

Syntax:

```
cout.setf (arg1, arg2)
```

The arg1 is one of the formatting flags defined in the class ios. the formatting flags specifies the format action required for the output. Another ios constant, arg2, know as bit field specifies the group to which the formatting flag belongs.

5.1.6 MANAGING OUTPUT WITH MANIPULATORS:

The header file iomanip provides the set of functions called manipulators which can be used to manipulate the output formats. They provide the same features as that of the ios member functions and flags. Some manipulators are more convenient to use then their counter parts in the ios. For example two or more manipulators can be used as a chain in one statement as shown below

EXAMPLE:

```
cout<<manip1<<manip2<<manip3<<item;
```

```
cout<<manip1<<item1<<manip2<<item2;
```

This kind of concatenation is useful when we born to display several column of output.

The tables also give their meaning and equivalents. To access these manipulators, we must include the file iomanip in the program.

Manipulator	Meaning
setw(int w)	
Setprecision(int d)	Set the field width to w.
	Set the floating point precision to d
setfill(int c)	Set the fill character to c
setiosflag(long f)	Set the format flag f.
Resetiosflag(long f)	Clear the flag specified by f.

DESIGNING OUR OWN MANIPULATORS:

We can design our manipulators for certain special purposes. The general form for creating a manipulator without any arguments is:

```
ostream & manipulator(ostream & output)
{
  …..
……(code)
…..
return output;
}
```

Here, the manipulator is the name of the manipulator under creation the following function defines a manipulator called unit that display “inches”:

```
ostream & unit (ostream & output)
{
 output<< “inches”;
return output;
}
```

The statement

```
cout<<36<<unit;
```

will produce the following output

```
      36 inches
```

We can also create manipulators that could represent a sequence of operations.

5.2.0 WORKING WITH FILES

5.2.1 INTRODUCTION:

We need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of file.

A file is a collection of related data stored in a particular area on the disk program can be designed to perform the read and write operations on these files.

5.2.2 Classes for File Stream Operations:

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.

5.2.3OPENING AND CLOSING A FILE:

If we want to use a disk file, we need to decide the following things about the file and its intended use:

- Suitable name for the file.
- Data type and structure.
- Purpose
- Opening method.

Class	Content
Filebuf	Its purpose is to set the file buffers to read and write.
Fstreambase	Provides operations common to the file streams.
Ifstream	Provide input operations
Ofstream	Provide output operations
Fstream	Provide support for simultaneous input and output operations.

OPENING FILES USING CONSTRUCTORS:

We know that a constructor is used to initialize an object while it is being created. Here a filename is used to initialize the file stream object. This involves the following steps:

create a file stream object to manage the stream using the appropriate class. That is to say, the class ofstream is used to create the output stream and the class.

- Initialize the file object with the desired filename.

Example:

```
ofstream outfile("results");
```

OPENING FILES USING OPEN()

As stated earlier, the function open() can be used to open multiple files that use the same stream object.

SYNTAX:

```
File-stream-class stream-object;
stream-object.open ("filename");
```

EXAMPLE:

```
ostream outfile;
outfile.open("data1");
……
……
outfile.close();
outfile.open("data2");
…..
……
outfile.close();
…
…..
```

The previous program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

5.2.4 DETECTING END-OF-FILE:

Detecting of the end-of-file condition is necessary for preventing any further attempt to read data from the file.

Example:

```
while (fin)
```

An ifstream object, such as fin, returns a value of 0 if any error occurs in the file operations including the end-of-file condition. Thus, the while loop terminates when file returns a value of zero on reaching the end of-file condition.

There is another approach to detect the end-of-file condition. Note that we have used the following statement:

```
if(fin1.eof() != 0)
{
exit;
}
```

eof() is a number function of ios class.

5.2.5 MORE ABOUT OPEN():FILEMODES

We have used ifstream and ofstream constructors and the function open() to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename, however, these functions can take two arguments, the second one for specifying the file mode. The general form of the function open() with two arguments is:

stream-object.open(“filename”,mode);

The second argument mode specifies the purpose for which the file is opened.

5.2.6 FILL POINTERS AND THEIR MANIPULATION:

Each file has associated pointer known as the file pointers. One of them is called the input pointer and the other is called the output pointer.we can use these pointers to move through the files while reading or writing. The inout pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output pointer is used for writing to given file location. Each time an input or output operation take place, the appropriate pointer is automatically advanced.

DEFAULT ACTIONS:

When we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to the end of the file.

H	E	L	L	O		H	A	I

FUNCTIONS FOR MANIPULATION OF FILE POINTERS:

All the actions on the file pointers take place automatically by default.we can control of the movement of the file pointers ourselves. The file stream classes support the following functions to manage such situations.

- seekg() moves get pointer to a specified location .
- seekp() moves put pointer to a specified location.
- tellg() give the current position of the get pointer.
- tellp() give the current position of the put pointer.

example:

infile.seekg(10);

It moves the file pointer to the byte number 10.

5.2.7 SPECIFYING THE OFFSET:

We have just now seen how to move a file pointer to a desired location using the seek functions.

'seek' functions seekg() and seekp() can also be used with two arguments as follows:

seekg (offset, reposition);

seekp (offset, reposition);

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter reposition.

The reposition takes one of the following three constants defined in the ios class:

* ios::beg	start of the file
* ios::cur	current position of the pointer
* ios::end	end of the file.

The seek function moves the associated file's 'get' pointer while the seekp() function moves the associated file's put pointer.

SEQUENTIAL INPUT AND OUTPUT OPERATIONS:

The file stream classes supports a number of member functions for performing the input and output operations on the files. One pair of functions write() and read are designed to write and read blocks of binary data.

put() and get() Functions:

The function put() writes a single character to the associated stream. Similarly, the functions get() read a single character from the associated stream.

WRITE() AND READ() FUNCTIONS:

The functions write() and read(), unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory.

READING AND WRITING A CLASS OBJECT:

The binary input and output functions read() and write() are designed to do exactly those job. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. The function write() copies a class object from memory byte by byte with no conversion. One important point to remember is that only data members are written to the disk file and the member functions are not.

5.2.8 UPDATING A FILE: RANDOM ACCESS:

Updating is a routine task in the maintenance of any data file. The updating would include one or more of the following tasks:

- displaying the contents of a file.
- modifying an existing item.
- adding a new item.
- deleting an existing item.

These actions require the file pointers to particular location that corresponds to the item/objects of equal lengths, in such a case, the size of each object can be obtained using the statement.

Example:

```
in object_length = sizeof(object);
```

That it receive the integer value of the given object.

Remember, we are opening an existing file for reading and updating the values. It is, therefore, essentially that the data members are of the same type and declared in the same order as in the existing file. Since, the member functions are not stored, they can be different.

5.2.9 ERROR HANDLING DURING FILE OPERATIONS:

So, far we have been opening and using the files for reading and writing on the assumption that every thing is fine with the files. This may not be true always for instance, one o the following things may happen when dealing with the files:

- a file which we are attempting to open for reading does not exist.
- The file name used for a new file may already exit.
- We may attempt an invalid operation such as reading past the end-of-file.
- There may not be any space in the disk for storing more data.
- We may use invalid file name.
- We may attempt to perform an operations when the file is not for that purpose.

The class ios supports several member functions that can be used to read the status recorded in a file stream. These functions that can be read the status recorded in a file stream. These functions along with their meanings are listed in table below:

FUNCTION	RETURN VALUE&MEANING
eof()	Returns true if end-of-file encountered while reading.
fail()	Returns true when an input or output operations has failed.
Good()	Returns true if any error is found
bad()	Returns true if an invalid operation is attempted

These functions may be used in the appropriate places in a program to locate the status of the file stream .

5.2.10 COMMAND-LINE ARGUMENTS:

The command-line arguments are typed by the user and delimited by a space. The first argument is always the filename and contains the program to be executed.

c> exam data results

here, exam is the name of the file containing the program to be executed and the data results are the filenames passed to the program as command-line arguments.

The main() functions which we have been using upto now without any arguments can take two arguments.

example:

main(int argc, char * argv[])

The first argument argc represent the number of arguments in the command line, .the second argument argc represents an array of char type pointers that points to the command line arguments.

State whether the following statements are TRUE or FALSE:

1) A C++ stream is a file.
2) C++ never truncates data.
3) The main advantage of **width()** function is that we can use one width specification for more than one item.
4) The **get(void)** function provides a single-character input that does not skip over the white spaces.
5) A programmer can define a manipulator that could represent a set of format functions.

6) The header file *iomanip.h* can be used in place of *iostream.h*
7) A stream may be connected to more than one file at a time.
8) A file pointer always contains the address of the file.
9) The **ios::ate** mode allows us to write data anywhere in the file.
10) We can add data to an existing file by opening in write mode.
11) The parameter **ios::app** can be used only with the files capable of output.
12) The data written to a file with **write()** function can be read with the **get()** function.
13) We can use the functions **tellp()** and **tellg()** interchangeably for any file.

Questions:

1) What is a stream?
2) Describe briefly the features of I/O system supported by C++.
3) Why it is necessary to include the file *iostream*.h in all our programs?
4) Discuss the various forms of **get()** function supported by the input stream. How are they used?
5) What is the role of **file()** function? When do we use this function?
6) Discuss the syntax of **set()** function?

NOTES

NOTES