

SUBJECT CODE : 210243

Strictly as per Revised Syllabus of
SAVITRIBAI PHULE PUNE UNIVERSITY
Choice Based Credit System (CBCS)
S.E. (Computer) Semester - I

OBJECT ORIENTED PROGRAMMING

(For IN SEM Exam - 30 Marks)

Anuradha A. Puntambekar

M.E. (Computer)

Formerly Assistant Professor in
P.E.S. Modern College of Engineering,
Pune

Dr. Gayatri M. Bhandari

Ph. D in Computer Engineering,

M. Tech (CE), BE (CE)

Professor (Computer Dept.)

JSPM's, Bhivarabai Sawant Institute of Technology & Research
Wagholi, Pune



OBJECT ORIENTED PROGRAMMING

(For IN SEM Exam - 30 Marks)

Subject Code : 210243

S.E. (Computer) Semester - I

First Edition : August 2020

© Copyright with A. A. Puntambekar

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :



Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :

Yogiraj Printers & Binders
Sr.No. 10/1A,
Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-93-90041-78-7



9789390041787

SPPU 19

PREFACE

The importance of **Object Oriented Programming** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Object Oriented Programming**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors

A. A. Puntambekar
Dr. Gayatri M. Bhandari

Dedicated to God.

SYLLABUS

Object Oriented Programming - (210243)

Credit	Examination Scheme and Marks
03	Mid - Semester (TH) : 30 Marks

Unit I Fundamentals of Object Oriented Programming

Introduction to object-oriented programming, Need of object-oriented programming. Fundamentals of object-oriented programming: Namespaces, objects, classes, data members, methods, messages, data encapsulation, data abstraction and information hiding, inheritance, polymorphism, Benefits of OOP, C++ as object oriented programming language.

C++ Programming - C++ programming Basics, Data Types, Structures, Enumerations, control structures, Arrays and Strings, Class, Object, class and data abstraction, Access specifiers, separating interface from implementation. **Functions** - Function, function prototype, accessing function and utility function, Constructors and destructors, Types of constructor, Objects and Memory requirements, Static members: variable and functions, inline function, friend function. (**Chapter - 1**)

Unit II Inheritance and Pointers

Inheritance - Base Class and derived Class, protected members, relationship between base Class and derived Class, Constructor and destructor in Derived Class, Overriding Member Functions, Class Hierarchies, Public and Private Inheritance, Types of Inheritance, Ambiguity in Multiple Inheritance, Virtual Base Class, Abstract class, Friend Class, Nested Class.

Pointers : declaring and initializing pointers, indirection Operators, Memory Management : new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, accessing Arrays using pointers, Arrays of Pointers, Function pointers, Pointers to Pointers, Pointers to Derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer. (**Chapter - 2**)

TABLE OF CONTENTS

Unit - I

Chapter - 1 Fundamentals of Object Oriented Programming (1 - 1) to (1 - 70)

Part I : Fundamentals

1.1 Introduction to Procedural, Modular, Generic and Object-Oriented Programming Techniques.....	1 - 3
1.1.1 Introduction to Procedural Programming Technique	1 - 3
1.1.2 Introduction to Modular Programming Technique	1 - 3
1.1.3 Introduction to Generic Programming Technique	1 - 4
1.2 Limitations of Procedural Programming.....	1 - 4
1.3 Need of Object-Oriented Programming	1 - 5
1.4 OOP Paradigm.....	1 - 5
1.5 Fundamentals of Object-Oriented Programming	1 - 7
1.5.1 Namespaces	1 - 7
1.5.2 Objects	1 - 7
1.5.3 Classes	1 - 7
1.5.4 Data Members	1 - 8
1.5.5 Methods and Messages	1 - 8
1.5.6 Data Encapsulation	1 - 9
1.5.7 Data Abstraction and Information Hiding.....	1 - 10
1.5.8 Inheritance	1 - 11
1.5.9 Polymorphism.....	1 - 11
1.6 Benefits of OOP	1 - 12
1.7 Drawbacks of OOP	1 - 12
Part II : Introduction to C++	
1.8 C++ as Object Oriented Programming Language.....	1 - 13

1.9 C++ Programming Basics.....	1 - 13
1.9.1 Comments in Program	1 - 15
1.9.2 Input and Output Operators	1 - 15
1.10 Data Types	1 - 16
1.11 Variable Declaration	1 - 17
1.12 Constant.....	1 - 18
1.13 Operator	1 - 19
1.14 Structures	1 - 19
1.14.1 Comparison between Arrays and Structure	1 - 20
1.14.2 Initializing Structure	1 - 21
1.15 Enumerations.....	1 - 23
1.16 Control Structures.....	1 - 24
1.17 Arrays	1 - 27
1.17.1 Characteristics of Arrays	1 - 28
1.17.2 Initialization of Arrays	1 - 28
1.18 Strings	1 - 31
1.18.1 String I/O Functions	1 - 32
1.18.2 Use of String Class	1 - 35
1.19 Class	1 - 37
1.19.1 Concept and Definition of Class.....	1 - 37
1.20 Object	1 - 38
1.21 Class and Data Abstraction	1 - 39
1.22 Class Scope and Accessing Class Members.....	1 - 39
1.22.1 Accessing Class Members that are Defined Inside the Class.....	1 - 40
1.22.2 Accessing Class Members that are Defined Outside the Class	1 - 44
1.23 Access Specifiers.....	1 - 45
1.24 Separating Interface from Implementation.....	1 - 46
1.25 Functions.....	1 - 47

Part III : Functions

1.25.1 Function Prototype	1 - 48
1.25.2 Argument Passing	1 - 49
1.26 Accessing Function and Utility Function	1 - 53
1.27 Constructors	1 - 54
1.27.1 Characteristics of Constructors	1 - 55
1.27.2 Default Constructor	1 - 56
1.27.3 Parameterized Constructor	1 - 56
1.27.4 Default Argument Constructor	1 - 59
1.27.5 Copy Constructor	1 - 60
1.28 Destructor	1 - 61
1.29 Objects and Memory Requirements.....	1 - 62
1.30 Static Members : Variable and Functions	1 - 63
1.31 Inline Function	1 - 66
1.32 Friend Function	1 - 69
1.32.1 Properties of Friend Functions	1 - 70

Unit - II

Chapter - 2 Inheritance and Pointers

(2 - 1) to (2 - 58)

Part I : Inheritance

2.1 Basic Concept of Inheritance	2 - 3
2.2 Base Class and Derived Class	2 - 3
2.3 Public and Private Inheritance	2 - 4
2.4 Protected Members	2 - 6
2.5 Relationship between Base Class and Derived Class	2 - 8
2.6 Constructor and Destructor in Derived Class.....	2 - 12
2.7 Overriding Member Functions.....	2 - 13
2.8 Class Hierarchies	2 - 14
2.9 Types of Inheritance	2 - 15
2.9.1 Single Inheritance	2 - 15

2.9.2 Multi - Level Inheritance	2 - 17
2.9.3 Multiple Inheritance.....	2 - 19
2.9.4 Hybrid Inheritance	2 - 21
2.9.5 Hierarchical Inheritance.....	2 - 23
2.10 Ambiguity in Multiple Inheritance.....	2 - 26
2.11 Virtual Base Class.....	2 - 28
2.12 Abstract Class.....	2 - 31
2.13 Friend Class	2 - 34
2.14 Nested Class.....	2 - 35

Part II : Pointers

2.15 Pointer - Indirection Operator	2 - 36
2.16 Declaring and Initializing Pointers	2 - 37
2.16.1 Accessing Variable through Pointers.	2 - 38
2.17 Memory Management : New and Delete	2 - 40
2.18 Pointers to Object.....	2 - 42
2.19 this Pointers.....	2 - 43
2.20 Pointers Vs Arrays.....	2 - 44
2.21 Accessing Arrays using Pointers.....	2 - 44
2.22 Pointer Arithmetic	2 - 46
2.23 Arrays of Pointers	2 - 49
2.24 Function Pointers.....	2 - 50
2.24.1 Passing Pointer to the Function.	2 - 51
2.24.2 Returning Pointer from Function	2 - 52
2.25 Pointers to Pointers	2 - 55
2.26 Pointers to Derived Classes	2 - 56
2.27 Null Pointer	2 - 57
2.28 void Pointer	2 - 57

Unit - I

1

Fundamentals of Object Oriented Programming

Syllabus

Introduction to object-oriented programming, Need of object-oriented programming, Fundamentals of object-oriented programming : Namespaces, objects, classes, data members, methods, messages, data encapsulation, data abstraction and information hiding, inheritance, polymorphism, Benefits of OOP, C++ as object oriented programming language.

C++ Programming - C++ programming Basics, Data Types, Structures, Enumerations, control structures, Arrays and Strings, Class, Object, class and data abstraction, Access specifiers, separating interface from implementation. **Functions -** Function, function prototype, accessing function and utility function, Constructors and destructors, Types of constructor, Objects and Memory requirements, Static members: variable and functions, inline function, friend function.

Contents

1.1	Introduction to Procedural, Modular, Generic and Object-Oriented Programming Techniques	
1.2	Limitations of Procedural Programming	
1.3	Need of Object-Oriented Programming	
1.4	OOP Paradigm Dec.-16, May-19, Marks 4	
1.5	Fundamentals of Object-Oriented Programming May-14, 17, 18, Dec.-19, Marks 6	
1.6	Benefits of OOP	
1.7	Drawbacks of OOP	
1.8	C++ as Object Oriented Programming Language	
1.9	C++ Programming Basics	
1.10	Data Types Dec.-18, Marks 3	
1.11	Variable Declaration	
1.12	Constant	
1.13	Operator	

1.14 Structures		
1.15 Enumerations		
1.16 Control Structures		
1.17 Arrays		
1.18 Strings		
1.19 Class		
1.20 Object	Dec.-17,	Marks 6
1.21 Class and Data Abstraction		
1.22 Class Scope and Accessing Class Members .		
1.23 Access Specifiers	Dec.-19,	Marks 2
1.24 Separating Interface from Implementation		
1.25 Functions	May-17,	Marks 8
1.26 Accessing Function and Utility Function		
1.27 Constructors	Dec.-17,	Marks 6
1.28 Destructor	May-19,	Marks 6
1.29 Objects and Memory Requirements		
1.30 Static Members : Variable and Functions . . .	May-17, Dec.-19,	Marks 4
1.31 Inline Function	Dec.-16, 18, 19, May-19,	Marks 6
1.32 Friend Function	Dec.-16, 18, May-18, 19,	Marks 6

Part I : Fundamentals**1.1 Introduction to Procedural, Modular, Generic and Object-Oriented Programming Techniques****1.1.1 Introduction to Procedural Programming Technique**

- This language is command driven or statement oriented language.
- The procedural programming is also called as imperative programming language.
- A program consists of sequence of statements. After execution of each statement the values are stored in the memory.
- The central features of this language are variables, assignment statements, and iterations.
- Examples of imperative programming are - C, Pascal, Ada, Fortran and so on.

Merits :

1. Simple to implement.
2. These languages have low memory utilization.

Demerits :

1. Large complex problems can not be implemented using this category of language.
2. Parallel programming is not possible in this language.
3. This language is less productive and at low level compared to other programming languages.

1.1.2 Introduction to Modular Programming Technique

- Modular programming is the process of subdividing a computer program into separate sub-programs.
- A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.
- Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.

Merits

- 1) Due to modular programming approach, the algorithm can be understood easily.
- 2) Many programmers can be employed one for each module.

- 3) Testing can be more thorough on each module.
- 4) The programs can be developed efficiently.
- 5) The modules can be reused.

Demerits

- 1) Due to multiple modules, it can lead problems to variable names.
- 2) Detail documentation is required for each module.
- 3) It can lead problems when modules are linked because links must be tested.

1.1.3 Introduction to Generic Programming Technique

Definition : Generic programming is an approach in which algorithms are written in terms of **type** to be specified later that are then instantiated when needed for specific types provided as parameters.

In the language like ADA, the generic programming approach is used.

Similarly, in C++ the Standard Template Library(STL) allows us to adopt the general programming approach.

Demerits

- 1) The abstract code can be written using this approach, which can be used to serve any data type element.
- 2) Generic programming paradigm is an approach to software decomposition.

Demerits

- 1) The syntax is complicated.
- 2) It is complex to implement.
- 3) The extra instantiations generated by templates can also cause the difficulty for the debuggers to debug the program.

1.2 Limitations of Procedural Programming

Following are some **limitations of procedure oriented programming** -

1. Global data is accessible by all the functions. Thus if some important data is declared as a global data then it will be accessed by all the functions. Any function can change it or destroy it. This will cause loss of an important information.

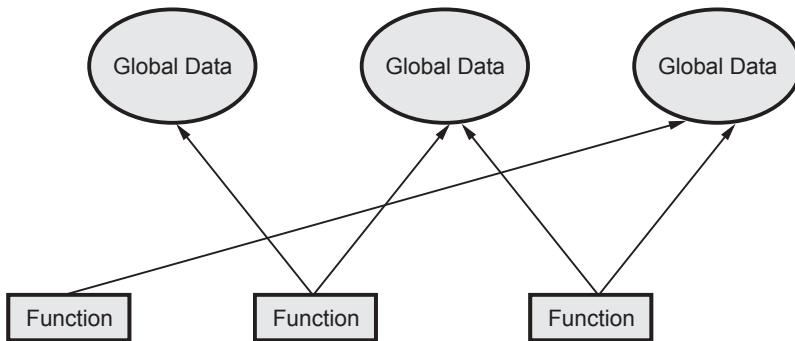


Fig. 1.2.1 Procedural paradigm

2. Sometimes many functions access the same set of data. Hence if we want to change that data then all the functions that are accessing them need to be modified. Similarly if a new data item is added then all the functions has to be modified so that they can access this new data item.
3. If we want to create a new data type then we can create the desired kind of data type, but then the programs become complex to write and maintain.

1.3 Need of Object-Oriented Programming

- Major motivation of object oriented programming is to **overcome the limitations** of procedural programming.
- The object oriented programming is **used in the applications** in which -
 1. Emphasis is on data rather than procedures.
 2. Programs can be divided into known objects.
 3. Data needs to be hidden from the outside functions.
 4. New data needs to be added frequently for maintaining the code.
 5. Objects need to communicate with each other.
 6. Some common properties are used by various functions.

1.4 OOP Paradigm

SPPU : Dec.-16, May-19, Marks 4

The basic idea behind object oriented programming is to combine into a single unit both the data and the functions that operate on the data. This unit is called as **object**. The functions in the object are called **member functions** and the data within it is called **instance variables**. The data can't be accessed directly, it can be accessed using function. Hence accidental use of data can be avoided in object oriented programming. This property is known as **data hiding**. The data and related functions are enclosed within an object. This is known as **data encapsulation**. Referring Fig. 1.4.1.

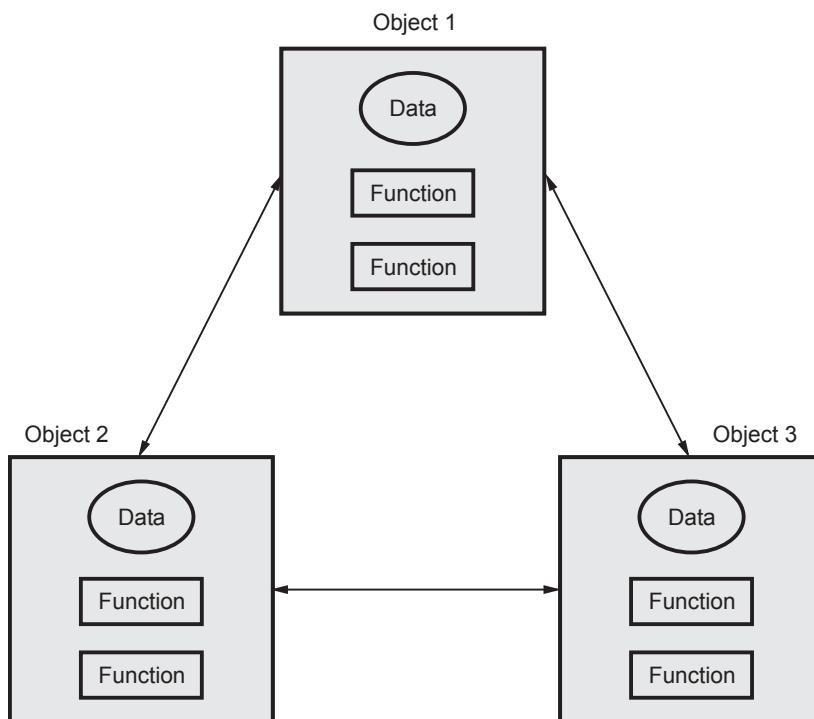


Fig. 1.4.1 Object oriented programming paradigm

Sr. No.	Procedural Programming Language	Object Oriented Programming Language (OOP)
1.	The procedural programming executes series of procedures sequentially.	In object oriented programming approach there is a collection of objects .
2.	This is a top down programming approach.	This is a bottom up programming approach.
3.	The major focus is on procedures or functions .	The main focus is on objects .
4.	Data reusability is not possible.	Data reusability is one of the important feature of OOP.
5.	Data hiding is not possible.	Data hiding can be done by making it private.
6.	It is simple to implement.	It is complex to implement.
7.	For example : C, Fortran, COBOL	For example : C++, JAVA

Review Question

1. Compare procedure oriented programming Vs. Object oriented programming.

SPPU : Dec.-16, May-19, Marks 4

1.5 Fundamentals of Object-Oriented Programming

SPPU : May-14, 17, 18, Dec.-19, Marks 6

1.5.1 Namespaces

- Namespaces are used to group the entities like class, variables, objects, and functions under some name. The namespaces help to divide global scope into sub-scopes where each sub-scope has its own name.
- In C++, the keyword **using** is used to introduce the namespace being used currently.
- All the files in the C++ standard library declare all its entities within the std namespace. That is why in the C++ program following line is written at beginning,

using namespace std

1.5.2 Objects

- Object is an instance of a class.
- Objects are basic run-time entities in object oriented programming.
- In C++ the class variables are called objects. Using objects we can access the member variable and member function of a class.
- Object represent a person, place or any item that a program handles.
- For example - If the class is **country** then the objects can be India, China, Japan, U.S.A and so on.
- A single class can create any number of objects.
- Declaring objects -**

The syntax for declaring object is -

Class_Name Object_Name;

- **Example**

Fruit f1;

For the class **Fruit** the object **f1** can be created.

1.5.3 Classes

- A class can be defined as an entity in which data and functions are put together.

- The concept of class is similar to the concept of **structure** in C.
- **Syntax of class** is as given below

```
class name_of_class
{
    private :
        variables declarations;
        function declarations;
    public :
        variable declarations;
        function declarations;
} ; ← do not forget semicolon
```

- **Example**

```
class rectangle
{
    private :
        int len, br;
    public :
        void get_data ( );
        void area( );
        void print_data ( );
};
```

- **Explanation**

- The class declared in above example is **rectangle**.
- The class name must be preceded by the keyword **class**.
- Inside the body of the class there are two keywords used **private** and **public**. These are called **access specifiers**.

1.5.4 Data Members

- The data members are the variables that are declared within the class.
- These members are declared along with the data types.
- The access specifier to these members can be public, private or protected.
- These data members can be accessible by the main() function using the object of a class.

1.5.5 Methods and Messages

- Object is an instance of a class. Every object consists of both data attributes and methods. The data attributes of every object are manipulated by the methods. These objects in the program communicate with each other by sending messages.

- **Message passing** is a mechanism by which the objects interact with each other. The process of interaction of objects is as given below -
 1. Define the class with data members and member functions.
 2. Create the objects belonging to the class.
 3. Establish the communication between these objects using the methods.
- Objects send information to each other using the methods. This process is called as message passing.
- A message for an object is nothing but the request for the execution of the procedure. Thus whenever the object wants to communicate it invokes a method. The procedure or method receives the information passed by an object and it generates the result. For example

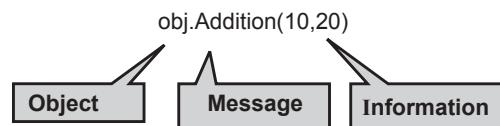


Fig. 1.5.1 Message

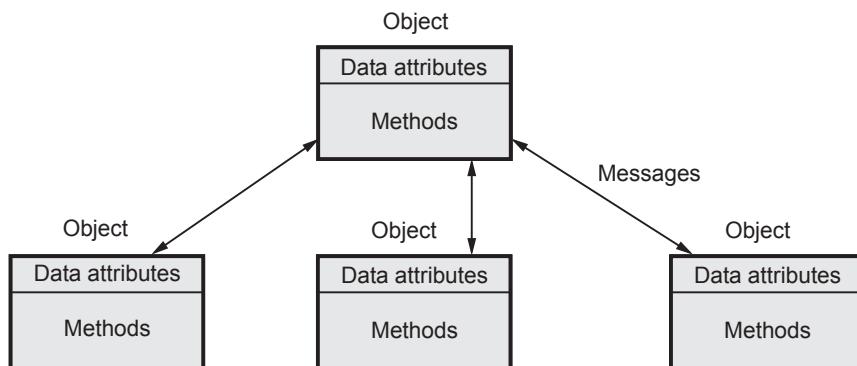


Fig. 1.5.2 Methods and messages

1.5.6 Data Encapsulation

- Encapsulation is for the detailed implementation of a component which can be hidden from rest of the system.
- In C++ the data is encapsulated.
- **Definition :** Encapsulation means binding of data and method together in a **single entity called class**.
- The data inside that class is accessible by the function in the same class. It is normally not accessible from the outside of the component.

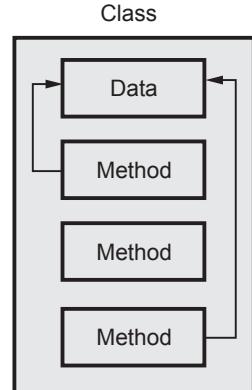


Fig. 1.5.3 Concept of encapsulation

1.5.7 Data Abstraction and Information Hiding

Abstraction

- **Definition :** Data abstraction means representing only essential features by hiding all the implementation details. In C++, class is an entity used for data abstraction purpose.
- **Example**

```
class Student
{
    int roll;
    char name [10];
public:
    void input ();
    void display ();
}
```

In main function we can access the functionalities using object. For instance

```
Student obj;
obj.input ();
obj.display ();
```

Thus only abstract representation can be presented, using class.

Information Hiding

The data member of member function of a class can be declared as **public** or **private**. If particular data attribute is declared as public then it is accessible to any other class. But if the data member is declared as **private** then only the member function of that class can access the data values. Another class cannot access these data values. This property is called **data hiding**.

Difference between Data Abstraction and Data Encapsulation

Sr. No.	Data encapsulation	Data abstraction
1.	It is a process of binding data members of a class to the member functions of that class.	It is the process of eliminating unimportant details of a class. In this process only important properties are highlighted.
2.	Data encapsulation depends upon object data type.	Data abstraction is independent upon object data type.
3.	It is used in software implementation phase.	It is used in software design phase.
4.	Data encapsulation can be achieved by inheritance.	Data abstraction is represented by using abstract classes.

1.5.8 Inheritance

- **Definition :** Inheritance is a property by which the new classes are created using the old classes. In other words the new classes can be developed using some of the properties of old classes.
- Inheritance support hierarchical structure.
- The old classes are referred as **base classes** and the new classes are referred as **derived classes**. That means the derived classes inherit the properties (data and functions) of base class.

- **Example :**

Here the **Shape** is a base class from which the **Circle**, **Line** and **Rectangle** are the derived classes. These classes inherit the functionality **draw()** and **resize()**. Similarly the **Rectangle** is a base class for the derived class **Square**. Along with the derived properties the derived class can have its own properties. For example the class **Circle** may have the function like **backgrcolor()** for defining the back ground color.

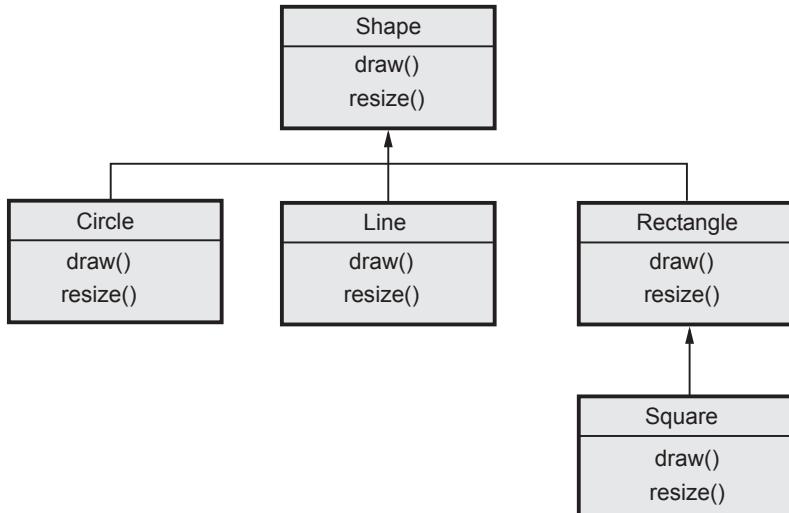


Fig. 1.5.4 Hierarchical structure of inheritance

1.5.9 Polymorphism

- Polymorphism means **many structures**.
- **Definition :** Polymorphism is the ability to take more than one form and refers to an operation exhibiting different behavior in different instances (situations).
- The behavior depends on the type of data used in the operation. It plays an important role in allowing objects with different internal structures to share the same external interface.
- Without polymorphism, one has to create separate module names for each method.
- For example the method **clean** is used to clean a **dish** object, one that cleans a **car** object, and one that cleans a **vegetable** object.

- With polymorphism, you create a single "clean" method and apply it for different objects.

Review Questions

1. Explain the features of object oriented programming.

SPPU : May-17, Marks 3, May-18, Marks 4, Dec.-19, Marks 6

2. Define the term - Class.

SPPU : May-14, Marks 2

1.6 Benefits of OOP

Following are some advantages of object oriented programming -

- Using **inheritance** the redundant code can be eliminated and the existing classes can be used.
- The standard working **modules** can be created using object oriented programming. These modules can then communicate to each other to accomplish certain task.
- Due to **data hiding** property, important data can be kept away from unauthorized access.
- It is possible to create **multiple objects** for a given class.
- For **upgrading the system** from small scale to large scale is possible due to object oriented feature.
- Due to **data centered nature** of object oriented programming most of the details of the application model can be captured.
- Message passing technique** in object oriented programming allows the objects to communicate to the external systems.
- Partitioning the code** for simplicity, understanding and debugging is possible due to object oriented and modular approach.

1.7 Drawbacks of OOP

Following are some drawbacks of OOP -

- The object oriented programming is **complex to implement**, because every entity in it is an object. We can access the methods and attributes of particular class using the object of that class.
- If some of the members are declared as **private** then those **members** are **not accessible** by the object of another class. In such a case you have to make use of inheritance property.
- In Object oriented programming, every thing must be arranged in the forms of **classes and modules**. For the lower level applications it is not desirable feature.

Part II : Introduction to C++

1.8 C++ as Object Oriented Programming Language

- The language C++ (pronounced as see plus plus) was developed by **Bjarne Strousstrup** in 1979 at Bell Labs.
- It is popularly known as a object oriented programming language. This language is compiled.
- C++ began as an enhancement to C, first by adding classes, virtual functions, inheritance and many other features.

Difference between C and C++

Sr. No.	C language	C++ language
1.	C is a procedure oriented language.	C++ is an object oriented programming language.
2.	C makes use of top down approach of problem solving.	C++ makes use of bottom up approach of problem solving.
3.	The input and output is done using <code>scanf</code> and <code>printf</code> statements.	The input and output is done using <code>cin</code> and <code>cout</code> statements.
4.	The I/O operations are supported by <code>stdio.h</code> header file.	The I/O operations are supported by <code>iostream.h</code> header file.
5.	C does not support inheritance, polymorphism, class and object concepts.	C++ supports inheritance, polymorphism, class and object concepts.
6.	The data type specifier or format specifier (%d, %f, %c) is required in <code>printf</code> and <code>scanf</code> functions.	The format specifier is not required in <code>cin</code> and <code>cout</code> functions.

1.9 C++ Programming Basics

Structure of C++

There are four sections in the structure of C++ program -

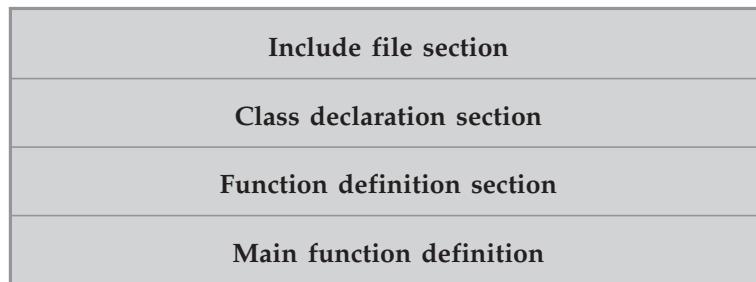


Fig. 1.9.1 Structure of C++ program

The sample C++ program is as follows-

C++ Program

FirstProg.cpp

```
//This is my first C++ program  
//This program simply prints the message Hello World  
#include<iostream>  
using namespace std;  
int main()  
{  
    cout<<"Hello World\n";  
}
```

Execute above program using the g++ command on Linux platform. Note that the gcc package must be installed on your machine to use this command. Following commands must be executed on terminal window.

- 1) g++ - O First Prog FirstProg.cpp
- 2) ./FirstProg

Program Explanation

The first two lines are the comments. It is a non executable portion.

Then the next line statement begins with the sign #. It is a pre-processor directive. It tells the pre-processor to include the header file **iostream.h**. This specific file iostream.h includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program. In Linux platform **# include<iostream>** is used.

Next line is

using namespace std;

This statement is used to define scope of the identifiers that are used in the program. **std** is a namespace where the standard class libraries are defined. By this statement we can bring all the identifiers in the current global scope. The keyword **namespace** is used to define the scope of identifiers. If namespace is not used then cout statement, must be written as std :: cout.

The **void main()** is a function from where the execution of C++ program starts. Then { indicates the beginning of the function definition.

The next line starts with **cout**. The **cout** represents the standard output stream in C++. By this statement we will get the "Hello world" message printed on the output screen. The **cout** is declared in standard **iostream** file. The cout should be followed by << and then by some message in double quotes.

Then the program ends by }.

1.9.1 Comments in Program

Comments are those statements in the program which are **non executable** in nature. They provide simply the information about the programming statements.

There are two ways by which we can give the comments in C++. The First way is similar to the comments in C. The start of the comment is /* and end of the comment is */. This comment statement is a multiline comment statement that means you can comment many lines together using /* and */. Another way is // at the start of the line which is to be commented. This is a single line comment statement. The comment statements are useful in documenting the program.

1.9.2 Input and Output Operators

In C++ the input operation is called **extraction** because data is extracted from keyboard and the operator `>>` is called **extractor**.

The C++ statement input operation will be -

```
cin >> a
```

This input statement on execution will wait for the user to type in a number using keyboard, the value entered by user will then be saved in a variable.

The C++ uses an output operation called **insertion** because data is inserted or sent from the variable. The operator `<<` is called **insertion operator**.

The C++ statement for output operation will be -

```
cout << a;
```

The data contained in variable a will be displayed on the screen.

Example 1.9.1 Write a C++ program to convert the polar co-ordinates into rectangular co-ordinates. (Hint : Polar co-ordinates(radius, angle) and rectangular co-ordinates(x, y) where $x = r\cos(\text{angle})$ and $y = r\sin(\text{angle})$).

Solution :

```
/*
Program to convert the Polar co-ordinates into rectangular co-ordinates
*/
#include<iostream>
#include<math.h>
#define PI 3.14159265
using namespace std;
int main()
{
    double r,angle;
    void PolarToRect(double,double);
    cout<<" Enter the value of radius: ";
```

```

    cin>>r;
    cout<<" Enter the value of angle(in degree): ";
    cin>>angle;
    PolarToRect(r,angle);
    return 0;
}
void PolarToRect(double r,double angle)
{
    double x,y;
    x=r*cos(angle*PI/180);
    y=r*sin(angle*PI/180);
    cout<<"\nx: "<<x;
    cout<<"\ny: "<<y;
    cout<<endl;
}

```

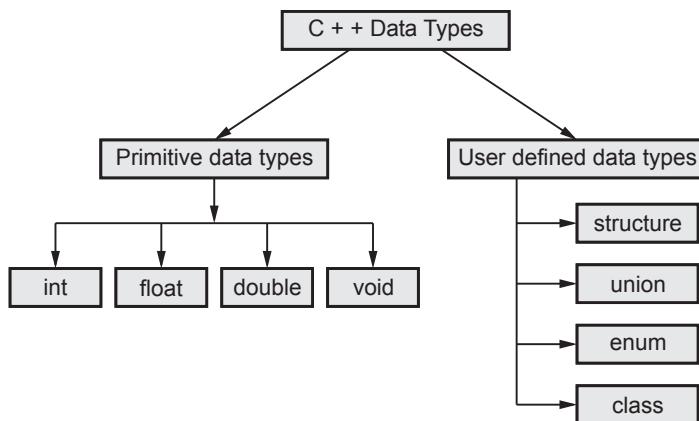
Output

Enter the value of radius: 5
 Enter the value of angle(in degree): 30
 x: 4.33013
 y: 2.5

1.10 Data Types**SPPU : Dec.-18, Marks 3**

The data types are specified by a standard keyword. The data types are used to define the type of data for particular variable. Various data types that are used in C++ are as shown by following Fig. 1.10.1.

Primitive data types are fundamental data types provided by C++. These are integer, float, double, char and void. User defined data types are structures, unions, enumerations and classes.

**Fig. 1.10.1**

Type	Size	Range
char	1 byte	- 127 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	- 127 to 127
int	4 bytes	- 2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	- 2147483648 to 2147483647
short int	2 bytes	- 32768 to 32767
unsigned short int	2 bytes	0 to 65,535
signed short int	2 bytes	- 32768 to 32767
long int	4 bytes	- 2,147,483,647 to 2,147,483,647
signed long int	4 bytes	same as long int
unsigned long int	4 bytes	0 to 4,294,967,295
float	4 bytes	+/- 3.4e +/- 38
double	8 bytes	+/- 1.7e +/- 308
long double	8 bytes	+/- 1.7e +/- 308

Review Question

1. What are primitive data types and user defined data types ?

SPPU : Dec.-18, Marks 3

1.11 Variable Declaration

Variable or identifier is an entity in a C++ program that stores some value. This value can be numerical, or characters.

Syntax for variable declaration:

Data_Type Variable_Name;

Example

```
int index;
char choice;
```

The variables is a sequence of one or more letters or alphanumeric characters.

Variables should follow following Rules -

1. It should not start with digit. It should always start with a letter.

2. No special character is allowed in variable name except underscore.
3. There should not be any blank space in variable name.
4. The variable name should not be a keyword.
5. The variable name should be meaningful, so that its purpose can be easily understood.

Keyword

Keywords are special reserved words associated with some meaning. The keywords are -

asm	enum	private	throw
auto	explicit	protected	true
bool	export	public	try
break	extern	register	typedef
case	false	reinterpret_cast	typeid
catch	float	return	typename
char	for	short	union
class	friend	signed	unsigned
const	goto	sizeof	using
const_cast	if	static	virtual
continue	inline	static_cast	void
default	int	struct	volatile
delete	long	switch	wchar_t
do	mutable	template	while
double	namespace	this	
dynamic_cast	new		
else	operator		

1.12 Constant

Constants are used to define the fixed values in C++.

For example

100 ←———— Decimal number

78.66 ←———— Real number with decimal point

“Hello” ← String constant

‘T’ ← Character constant.

1.13 Operator

Various operators used in C++ are enlisted below -

Type	Operator	Meaning	For example
Arithmetic	+	Addition or unary plus	c = a + b
	-	Subtraction or unary minus	d = - a c = a - b
	*	Multiplication	c = a * b
	/	Division	c = a/b
	%	Mod	a%b
Relational	<	Less than	a < 4
	>	Greater than	a > 4
	<=	Less than equal to	a <= 4
	>=	Greater than equal to	a >= 4
	==	Equal to	a == 4
Logical	!=	Not equal to	a != 4
	&&	And	0 && 1
		Or	0 1
Assignment	=	Is assigned to	a = 5
Increment	++	Increment by one	+i or i++
Decrement	--	Decrement by one	--x or x --

1.14 Structures

Definition : A structure is a group of items in which each item is defined by some identifier. These items are called **members** of the structure. Thus structure is a collection of various data items which can be of **different data types**.

The Syntax of declaring structure in C++ is as follows -

```
struct name {
    member 1;
    member 2;
    ...
    ...
    ...
    member n;
};
```

Example :

```
struct stud {
    int roll_no;
    char name[10];
    float marks;
};
struct stud stud1,stud2;
```

The stud 1 and stud 2 will look like this

roll_no	name [10]	marks
---------	-----------	-------

Using typedef

An alternative to using a structure tag is to use the structure tag is to use the **typedef** definition. For example

```
typedef struct {
    int roll_no;
    char name[10];
    float marks;
} stud;
```

The word **stud** represents the complete structure now. So whenever the word **stud** will appear there ever you can assume the complete structure. The stud will act like a data type which will represent the above mentioned structure. So we can declare the various structure variables using the tag stud like this-

```
stud stud1,stud2;
```

1.14.1 Comparison between Arrays and Structure

Sr. No.	Array	Structure
1.	Array is a collection of similar type of elements.	Structure is a collection of variety of elements which can be of different data types.
2.	Array elements can be accessed by the index placed within [].	Structure elements can be accessed with the help of . (dot) operator.

3.	To represent an array, array name is followed by [].	To represent structure a keyword struct has to be used.
4.	Example : int a [20];	Example : Struct student { int roll_no; char name [20]; }

1.14.2 Initializing Structure

The initialization of the structure should be within the brackets, as shown below

```
struct stud
{
    int roll_no;
    char name[10];
    float marks;
};

struct stud stud1={1,"ABC",99.99};
struct stud stud2={2,"XYZ",80};
```

Usually the structure declaration should be done **before the main function** i.e. at the top of the source code file, before the variable or the function declaration.

The C++ Program that uses structure for defining the collection of the members of different data types is as follows -

```
*****
This Program is for assigning the values to the structure
variable. Also for retrieving the values.
*****
#include<iostream>
using namespace std;
struct student {
    int roll_no;
    char name[10];
    float marks;
}stud1;           Structure tag
void main(void)
{
    cout<<"\n Enter the roll number: ";
    cin>>stud1.roll_no;           Using dot. we are accessing
                                member roll_no
    cout << "\n Enter the name: ";
    cin>>stud1.name;
    cout<<"\n Enter the marks: ";
    cin>>stud1.marks;
```

```

cout<<"\n The record of the student is ";
cout<<"\n\n Roll_no    Name    Marks";
cout<<"\n-----\n";
cout<<"    "<<stud1.roll_no<<"\t"<<stud1.name<<"\t"<<stud1.marks;
}

```

Output

Using dot operator each member is printed

Enter the roll number: 10

Enter the name: Anuja

Enter the marks: 98

The record of the student is

Roll_no Name Marks

10 Anuja 98

Example 1.14.1 Write a C program to represent a complex number using structure and add two complex numbers.

Solution :

```

#include<iostream>
using namespace std;
typedef struct Complex
{
    float real;
    float img;
}C;
void main()
{
    C x, y, z;
    cout<<"\n Enter the real part of first complex number: ";
    cin>>x.real;
    cout<<"\n Enter the imaginary part of first complex number: ";
    cin>>x.img;
    cout<<"\n Enter the real part of second complex number: ";
    cin>>y.real;
    cout<<"\n Enter the imaginary part of second complex number: ";
    cin>>y.img;
    cout<<"\n\t The First complex number is "<<x.real<<"+"<<x.img<<"i";
    cout<<"\n\t The second complex number is "<<y.real<<"+"<<y.img << "i";
    z.real = x.real + y.real;
    z.img = x.img + y.img;
    cout<<"\n The Addition is: "<<z.real<<"+"<<z.img << "i";
}

```

Output

```

Enter the real part of first complex number: 3
Enter the imaginary part of first complex number: 6
Enter the real part of second complex number: 4
Enter the imaginary part of second complex number: 8
The First complex number is 3+6i
The second complex number is 4+8i
The Addition is: 7+14i

```

1.15 Enumerations

To create user defined type names the keyword **enum** is used. The syntax of using enum is

```
enum name{list of names} variables;
```

For example :

```
enum day{Monday,Tuesday,Wednesday} d;
d=Tuesday;
```

By default the value of first name is 0, second name has value 1 and so on. But we can give some other specific value to the list element. For example

```
enum day{Monday=10,Tuesday=20,Wednesday=30};
```

Following is a simple C++ program that illustrates the use of enum

```
*****
Program for using enum
*****
#include<iostream>
using namespace std;
int main()
{
    enum mylist{small,middle=5,large} a,b;
    a=middle;
    b=large;
    cout<<"a= "<<a<<" b= "<<b;
    return 0;
}
```

Output

```
a= 5 b= 6
```

1.16 Control Structures

Various control structures are -

- 1. if statement
- 2. while statement
- 3. do-while statement
- 4. switch case statement
- 5. for loop

Let us discuss these control structures with the help of simple examples.

1. If statement

There are two types of if statements - simple if statement and compound if statement. The **simple if** statement is a kind of if statement which is followed by single statement. The **compound if** statement is a kind of if statement for which a group of statements are followed. These group of statements are enclosed within the curly brackets.

If statement can also be accompanied by the **else** part.

Following table illustrates various forms of **if statement**

Type of Statement	Syntax	Example
simple if	if(condition) statement	if(a<b) cout<<"a is smaller than b";
compound if	if(condition){ statement 1; }	if(a<b){ cout<<"a is smaller than b"; cout<<"b is larger than a"; }
if...else	if(condition) statement; else statement;	if(a<b) cout<<"a is smaller than b"; else cout<<"a is larger than b";
compound if...else	if(condition){ statement 1; }	if(a<b){ cout<<"a is smaller than b"; cout<<"b is larger than a"; } else

	<pre> } else { statement 1; } </pre>	<pre> { cout<<"a is larger than b"; cout<<"b is smaller than a"; } </pre>
if...else if	<pre> if(condition) { statement 1; } else if(condition) { statement 1; } else { statement 1; } </pre>	<pre> if(a<b) { cout<<"a is smaller than b"; } else if(a<c) { cout<<"a is smaller than b" } else { cout<<"a is larger than b and c"; } </pre>

2. while statement

The while statement is executing repeatedly until the condition is false. The while statement can be simple while or compound while. Following table illustrates the forms of while statements -

Type of statement	Syntax	Example
simple while	while(condition) statement	while(a<10) cout<<" a is smaller than 10";
compound while	while(condition) { statement 1; }	while(a<10) { cout<<"a is less than b"; a++; }

3. do..while

The do...while statement is used for repeated execution. The difference between do while and while statement is that, in while statement the condition is checked before executing any statement whereas in do while statements the statement is executed first and then the condition is tested. There is at least one execution of statements in case of do...while. Following table shows the use of do while statement.

Type of Statement	Syntax	Example
do...while	do { statement 1; } }while(condition);	do { cout<<"a is less than b"; a++; } } while(a<10);

Note that the while condition is terminated by a semicolon.

4. switch case statement

From multiple cases if only one case is to be executed at a time then switch case statement is executed. Following table shows the use of switch case statements

Type of statement	Syntax	Example
switch ...case	switch(condition) { case caseno:statements break; default: statements	cout<<"\n Enter choice"; cin>>choice; switch(choice) { case 1: cout<<"You have selected 1"; break;

```

        }
    }

    case 2: cout<<"You have selected 2";
              break;

    case 3: cout<<"You have selected 3";
              break;

    default: cout<<"good bye";
    }
}

```

5. for Loop

The for loop is a not statement it is a loop, using which the repeated execution of statements occurs. Following table illustrates the use of for loop -

loop	Syntax	Example
simple for loop	for(initialization; termination; step count) statement	for(i=0;i<10;i++) c[i]=a[i]+b[i];
compound for loop	for(initialization; termination; step count) { statement 1; statement 2; ... }	for(i=0;i<10;i++) { for(j=0;j<10;j++) c[i][j]=a[i][j]+b[i][j]; }

1.17 Arrays

- Array is a collection of similar data type elements.
- Syntax of array is

Datatype Name[size]

- Example

int a[10];

- This type of array is called **one dimensional array**.

By default the array index starts at 0. Following Fig. 1.17.1 represents the elements stored in array -

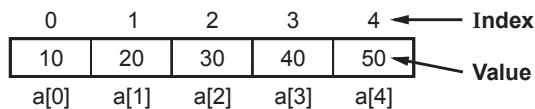


Fig. 1.17.1

1.17.1 Characteristics of Arrays

Following are some important characteristics of arrays -

1. The array contains all the elements of same data type.
2. All the elements of array share the same name and then can be distinguished from one another by index.
3. All the elements in an array are occupied at continuous memory locations.
4. The array size must be mentioned at the time of declaration. The size of the array must be constant expression and not the variable.
5. The name of the array represents the address of the first element of an array.

1.17.2 Initialization of Arrays

The process of storing the elements in an array is called as **initialization of arrays**. There are various ways by which the arrays can be initialized -

Method 1 :

Syntax

```
Data type name[size]={value 0, value 1, value 2, ..., value n-1};
```

Example

```
int a[5]={10,20,30,40,50};
```

Method 2 :

```
cout<<"\n Enter the elements";
for(int i=0;i<n;i++)
    cin>>a[i];
```

Following simple C++ program represents the second method of storing the elements in an array. Then we are reading the array element by element and displaying the contents.

```
*****
Program to store the elements in an array and then retrieve them
*****
#include<iostream.h>
void main()
{
    int a[10],i,n;
    cout<<"\n How many elements are there in the array?";
    cin>>n;
    cout<<"\n Enter the elements\n";
    for(i=0;i<n;i++)
```

```

    cin>>a[i];
    cout<<"\n The elements of an array are ...";
    for(i=0;i<n;i++)
        cout<<" "<<a[i];
}

```

Output

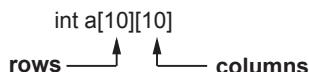
How many elements are there in the array?5

Enter the elements
10 20 30 40 50

The elements of an array are ...
10 20 30 40 50

Two dimensional arrays

The arrays in which the elements are arranged in the form of rows and columns are called **two dimensional arrays**.

For example :

The representation of two dimensional array is

	0	1	2
0	10	20	30
1	40	50	60

→ a[0][1]

The initialization of two dimensional arrays is -

Method 1 :

```

int a[2][3]={
    {10,20,30},
    {40,50,60}
};

```

Method 2 :

```

cout<<"\n Enter the elements";
for(int i=0;i<n;i++)
{
    for(int j=0;j<m;j++)
    {
        cin>>a[i][j];
    }
}

```

Following is a C++ program in which the addition of two matrices is performed using two dimensional array.

```
*****
Program to addition of two matrices
*****
#include<iostream.h>
#define SIZE 5
void main()
{
    int a[SIZE][SIZE],b[SIZE][SIZE],c[SIZE][SIZE];
    int i,j,n;
    cout<<"\n\t ADDITION OF TWO MATRICES ";
    cout<<"\n Enter the order for the matrix: ";
    cin>>n;
    cout<<"\n\t Enter the matrix a";
    cout<<"\n Enter the elements\n";
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            cin>>a[i][j];
    cout<<"\n\t Enter the matrix b";
    cout<<"\n Enter the elements\n";
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            cin>>b[i][j];
//Performing addition of two matrices
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        c[i][j]=a[i][j]+b[i][j];
//Displaying the matrix c
cout<<"\n The addition of two matrices is ...";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        cout<< " " <<c[i][j];
    }
    cout<<"\n";
}
}
```

Output

ADDITION OF TWO MATRICES
Enter the order for the matrix: 3

Enter the matrix a
Enter the elements

```

1 2 3
4 5 6
7 8 9

Enter the matrix b
Enter the elements
1 1 1
2 2 2
3 3 3

The addition of two matrices is ...
2 3 4
6 7 8
10 11 12

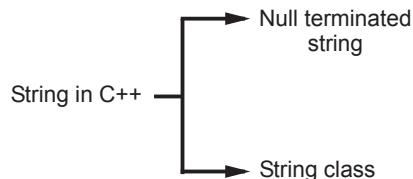
```

1.18 Strings

Definition : String is a sequence of characters which is represented within double quotes.

Example : "I love India"

C++ support two types of strings



Let us discuss the null terminated strings first

The string is stored in the memory with the terminating character '\0' with ASCII code. Each character is associated with its corresponding ASCII value.

For example -

The string is stored in the memory with the terminating character '\0' with ASCII code. Each character is associated with its corresponding ASCII value.

- **For example-**

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Characters	I		L	o	v	e		I	n	d	i	a	\0
ASCII code	73	32	76	111	118	101	32	73	110	100	105	97	0

Each character requires 1 byte of memory space. Successive characters are stored in successive bytes.

- **C/C++ representation :** In C/C++, the string is represented as array of characters.
For example the string **str** can be represented as
`char str[10];`
- **Initialization of string :** Initializing of string can be done as follows -
`char str[6] = "INDIA";`

Or

`char str[] = {'I', 'N', 'D', 'I', 'A'};`

The string will be stored in the array **str** as

I	N	D	I	A	\0
str[0]	str[1]	str[2]	str[3]	str[4]	str[5]

We need not have to explicitly mention '\0' at the end of the string. C/C++ inserts '\0' automatically at the end of the string.

1.18.1 String I/O Functions

Reading the strings

1. **Using cin** - The string can be read with the help of **cin**.

For example

```
char name[20];
cin >> name;
```

But the **cin** statement has some drawbacks. When we read some blank character using **cin** statement, then as soon as it finds the blank character it terminates. For instance :

If the string is "My Computer" then using **cin** we could read only "My" because after "My" blank character is encountered and the **cin** terminates there. This is illustrated by following simple C++ program.

C++ Program

```
#include<iostream>
using namespace std;
void main()
{
    char name[20];
    cout << "\n Enter the string : ";
    cin > name;
    cout << "\n You have entered : " << name;
}
```

Output

```
Enter the string : My Computer
```

```
You have entered : My
```

2. Using gets_s() - The drawback of **cin** function is removed in **gets_s** function. The **gets** is a function used to read the string of any length including space or tab character. The syntax of **gets_s** is -

```
gets_s(string);
```

Following program makes use of **get** function to read the string.

```
#include<iostream>
using namespace std;
void main()
{
    char name[20];
    cout<<"\n Enter the string : ";
    gets_s(name);
    cout<<"\n You have entered :" <<name;
}
```

Output

```
Enter the string : My computer
```

```
You have entered: My computer
```

3. Reading character by character

We can read the entire string character by character as follows

```
char str[10];
for(i=0;i]!='\0';i++)
{
    cin>>str[i];
}
```

Writing the strings

There are various functions for displaying the string on console (output screen).

1. Using cout

We can use **cout** statement to display the string.

For example

```
char name[20];
cout<<"\n Enter the string";
cin>>name;
cout<<"\n The entered string is ...";
cin>>name;
```

2. Using puts

The puts is a special function used to display the string.

The syntax of puts is

```
puts(string);
```

Following simple C++ program illustrates the use of puts for printing the string

C++ Program

```
#include<iostream>
using namespace std;
void main()
{
    char name[20];
    cout<<"\n Enter the string : ";
    gets_s(name);
    cout<<"\nYou have entered :";
    puts(name);
}
```

Output

```
Enter the string : My Computer
You have entered :My Computer
```

3. Printing character by character

Again while printing the string character by character we will use **cout** statement.

C++ Program

```
#include<iostream>
using namespace std;
void main()
{
    char name[10];
    int i;
    cout<<"\n Enter the string : ";
    gets_s(name);
    cout<<"\n You have entered : ";
    for (i = 0; name[i] != '\0'; i++)
        cout<<name[i];
}
```

Output

```
Enter the string : Computer
```

```
You have entered : Computer
```

In above program, before the last printf statement we have written one for loop. This for loop is for visiting each character in the string one by one. We have given the terminating condition as name[i]!='\0'. That means when we press the enter key after entering the string **name** then '\0' will be stored in name[i].

getchar() and putchar()

The **getchar()** is a macro which is used for reading a single character. The **putchar()** is a macro for outputting the character. The syntax is -

```
int getchar();
int putchar(int c);
```

The use of getchar and putchar is illustrated by following C++ program -

```
#include<iostream>
using namespace std;
void main()
{
    int ch;
    cout<<"\n When you want to terminate the string then only press Enter : ";
    while ((ch = getchar()) != '\n')
        putchar(ch);
}
```

Output

When you want to terminate the string then only press Enter : India
India

Library Functions used for handling string

Sr. No.	String Function	Purpose
1	strcpy(s1, s2);	Copies string s2 into string s1.
2	strcat(s1, s2);	Concatenates string s2 onto the end of string s1.
3	strlen(s1);	Returns the length of string s1.
4	strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.

1.18.2 Use of String Class

The **string** class is used to define the strings. The syntax of using the **string** class is as follows -

```

string()
string(char *str);
string(string &str);

```

Various operators can be used for performing the operations on strings are described by following table

Operator	Purpose
+	Used for concatenating two strings
=	Used for assigning the values to another string
==	Used for checking the equality of two strings
!=	Used for not equal to condition
<	for less than value
>	greater than value
<=	less than or equal to value
>=	The greater than or equal to value
[]	For denoting the subscript value

Following C++ program that illustrates the use of **string** class.

C++ Program

```

#include<iostream>
#include<string>
using namespace std;
void main()
{
    string s1, s2;
    s1 = "Hello";
    cout << "\n The string : ";
    cout << s1;
    s2 = s1;
    cout << "\n The copied string is: ";
    cout << s2;
    s2 = "Friends";
    cout << "\n The new string is: " << s2;
    string s3;
    s3 = s1 + s2;
    cout << "\n The concatenated string is: ";
    cout << s3;
}

```

Output

```
The string : Hello  
The copied string is: Hello  
The new string is: Friends  
The concatenated string is: HelloFriends
```

Note that it is essential to include header file **string** in order to use string class.

1.19 Class

1.19.1 Concept and Definition of Class

- Each class is a collection of data and functions that manipulate the data.
- Placing the **data and functions together** into a single entity is the **central idea** in object oriented programming.
- The **nouns** in the system specification help the C++ programmer to determine the set of **classes**. The objects for these classes are created. These objects work together to implement the system.
- Classes in C++ is the natural evolution of C notion of **struct**.

For example :

```
class rectangle  
{  
private:  
    int len,br;  
public:  
    void get_data();  
    void area();  
    void print_data();  
};
```

- **Rules for declaring class name -**

1. The class name must begin with the letters, it may be followed by letters, digits or underscore.
2. The name of the class must not be same as keyword or reserved word.
3. The class name must not contain any special character such as ~,!,@,#,\$,%,&,*,(,),{},[],+, -,|,/, \

Difference between Structure and Class

Sr. No.	Structure	Class
1.	By default the members of structure are public .	By default the members of class are private .
2.	The structure can not be inherited .	The class can be inherited .
3.	The structures do not require constructors .	The classes require constructors for initializing the objects.
4.	A structure contains only data members.	A class contains the data as well as the function members.

1.20 Object

SPPU : Dec.-17, Marks 6

The object is an instance of a class. Hence object can be created using the class name. The object interacts with the help of procedures or the methods defined within the class.

In order to instantiate an object we need to declare an object along with the class name.

For example -

```
int a; //an instance of type integer
double marks;//an instance of type double
Student xyz;//an instance of object xyz
```

Difference between Class and Object

Following are some differences between the class and the object -

Sr. No.	Class	Object
1.	For a single class there can be any number of objects. For example - If we define the class as River then Ganga, Yamuna, Narmada can be the objects of the class River.	There are many objects that can be created from one class. These objects make use of the methods and attributes defined by the belonging class.
2.	The scope of the class is persistent throughout the program.	The objects can be created and destroyed as per the requirements.
3.	The class can not be initialized with some property values.	We can assign some property values to the objects.
4.	A class has unique name.	Various objects having different names can be created for the same class.

Review Question

1. What is class and object ? Differentiate between class and object. **SPPU : Dec.-17, Marks 6**

1.21 Class and Data Abstraction

- Data abstraction is one of the most important feature of object oriented programming paradigm. It allows us to create user defined data type using **class** construct.
- With data abstraction we can think about **what operations** can be performed on particular type of data and not how it does.
- Data abstraction is used for increase the modularity of the program.

For example

```
class Student
{
    Private:
        int roll;
        char name[10];
    public:
        void input();
        void display();
};
```

In the main function, we can access the functionalities using the **object**. For instance -

```
Student obj;
obj.input();
obj.display();
```

- From above code the implementation details are hidden and to perform some task only the required functionalities are invoked.

1.22 Class Scope and Accessing Class Members

- The data members and member function defined within a class belong to that class's scope.
- The non-member functions belong to **global namespace**
- Within the class's scope, the data members of that class are immediately accessible by the member functions of that class.
- Outside the class the, only the public members of that class are accessible by handle. Following C++ program illustrates this concept

C++ Program

```
#include <iostream>
using namespace std;
class Test
{
private:
    int a,b,c;
public:
    int Addition(int a, int b)
    {
        c= a+b;
        return c;
    }
    void Display( )
    { cout << "The sum is:" << c << "\n";}
};

int main()
{
    Test obj;
    obj.Addition(10,20);
    obj.Display();
    return 0;
}
```

} Public member functions of class text

Accessing them outside the class
using **object of that class**

- The class members are accessed using the handle called **object** with the help of dot operator. The `->` is used to access the pointer variable.

1.22.1 Accessing Class Members that are Defined Inside the Class

- The function declared within the class is usually called as **member function** in C++ and it is called **method** in object oriented programming.
- The data declared along with some data type is called as the **data member** of that class.
- The **data members** denote the **property** of the class and the **member functions** denote the **operations** on that data.
- Normally in C++ the data within the class is declared as **private** and member functions are declared as **public**. This avoids any manipulation of data by the functions outside the class. As these functions are declared as **public** they can be accessible from outside the class.
- Some times you may require the **data** to be **private** and **member functions** to be **public**. Thus **access mode** helps the C++ programmer to hide the data whenever required.
- Following C++ program illustrates the concept of class, access specifiers, function and data members.

```
/*
Program to demonstrate the class, access specifiers and data and member functions
*/
#include <iostream>
using namespace std;
class Test
{
private:
    int a,b,c;           Data members are declared as private.
public:
    int Addition(int a, int b)
    {
        c = a+b;
        return c;
    }
    void Display( )
    { cout << "The sum is:" << c << "\n";}
};
int main()
{
    Test obj;
    obj.Addition(10,20);
    obj.Display();
    return 0;
}
```

Member functions are declared as public.
Note that: Only within the function the data members can be manipulated.

Only member functions are accessible outside the class because they are public.

Output

The sum is : 30

Example 1.22.1 Write a C++ program that inputs two numbers and outputs the largest number using class.

Solution :

```
/*
Program to check the largest number
*/
#include<iostream>
using namespace std;
class Test
{
    int a,b;
public:
    void Get_num()
    {
        cout << "Please two numbers: "<< endl;
        cin >>a>>b;
    }
}
```

```

void Check_largest()
{
    if(a>b)
        cout<<a<<" is largest number"<<endl;
    else
        cout<<b<<" is largest number"<<endl;
}
int main()
{
    Test obj;
    obj.Get_num();
    obj.Check_largest();
    return 0;
}

```

Example 1.22.2 Write a C++ program to check whether an integer is a prime or a composite number.

Solution :

```

/*********************************************************************
Program to check if the number is prime or not
*****
#include<iostream>
using namespace std;
class Test
{
    int num;
public:
    void Get_num()
    {
        cout << "Please enter a positive integer" << endl;
        cin >> num;
    }
    void Check_prime()
    {
        int flag = 1;
        for(int n = 2; n <= num - 1; n++)
        {
            if(num % n == 0)
            {
                flag=0;
            }
        }
        if(flag==1)
            cout<<num<<" is a prime number"<<endl;
    }
}

```

```

        else
            cout<<num<<" is a composite number"<<endl;
    }
};

int main()
{
    Test obj;
    obj.Get_num();
    obj.Check_prime();
    retrun 0;
}

```

Output(Run 1)

Please enter a positive integer
5
5 is a Prime number

Output(Run 2)

Please enter a positive integer
10
10 is a composite number

Example 1.22.3 Consider a Bank Account class with Acc No and balance as data members. Write a C++ program to implement the member functions get_Account_Details() and display_Account_Details(). Also write suitable main function.

Solution :

```
*****
Program to implement Bank Account class
*****
#include <iostream>
using namespace std;
class BankAccount
{
    int AccNo;
    double balance;
public:
    void getAccDetails()
    {
        cout<<"\n Enter the Account Number: ";
        cin>>AccNo;
        cout<<"\n Enter the Balance Amount: ";
        cin>>balance;
    }
    void DisplayAccDetails()
    {
```

```

cout<<"\nAccount Number: "<<AccNo;
cout<<"\nBalance Amount: "<<balance;
cout<<endl;

}

};

int main()
{
    BankAccount obj;
    obj.getAccDetails();
    cout<<"\n The account details are ..."<<endl;
    obj.DisplayAccDetails();
    return 0;
}

```

Output

Enter the Account Number: 101

Enter the Balance Amount: 10000

The account details are ...

Account Number: 101

Balance Amount: 10000

1.22.2 Accessing Class Members that are Defined Outside the Class

The members functions that needs to be defined **outside the class** are defined using the **scope resolution** operator. Following program illustrates it

```
*****
Program that uses a class where the member functions are defined outside a class
*****/




#include <iostream>
using namespace std;
class Test
{
private:
    int a,b,c;
public:
    int Addition(int,int);
    void Display();
};

int Test::Addition(int a, int b)      // Definition of function
{                                     // Outside the class
    c= a+b;
    return c;
}
```

```

}

void Test::Display()
{ cout < "The sum is:" < c < "\n";}

int main()
{
    Test obj;
    obj.Addition(10,20);
    obj.Display();
    return 0;
}

```

Output

The sum is:30

1.23 Access Specifiers

SPPU : Dec.-19, Marks 2

- Inside the body of the class the data members and functions are declared using the keywords like **private**, **public** or **protected**. These are called **access specifiers**.
- The access specifier specifies the manner in which the data can be accessed.
- By default the access specifier is of **private** type. When the data and functions are declared as private then only members of same class can access them. This achieves the **data hiding** property.

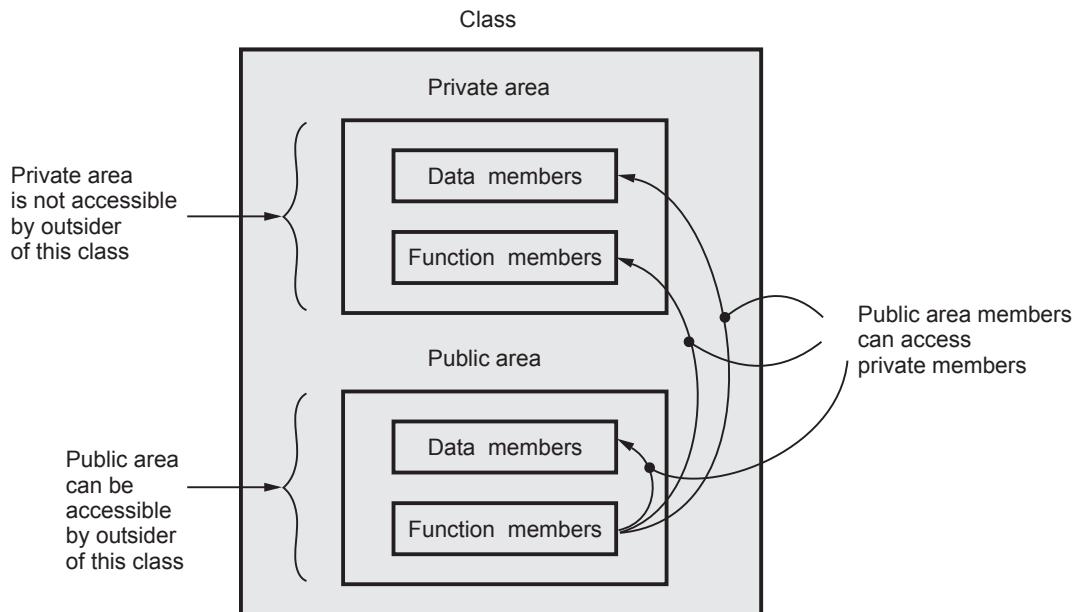


Fig. 1.23.1 Accessibility for members of class

- The **public** access specifier allows the function declared outside the class to access the data and functions declared under it.
- The **protected** access specifier allows the functions and data declared under it be accessible by the belonging class and the immediate derived class. Outside the belonging class and derived class these members are not accessible.
- The accessibility of members of class is as shown in following Fig. 1.23.1. (See Fig. 1.23.1 on previous page.)

Review Question

1. Explain visibility modes in inheritance.

SPPU : Dec.-19, Marks 2

1.24 Separating Interface from Implementation

The **interface file** is a file that contains the declaration of the class. The **implementation files** are the files that define the actual functionalities and invoke those functionalities.

Thus the C++ program can be split into

- **Header files** - contains class definitions and function prototypes
- **Source-code files** - contains member function definitions

The **advantage** of this arrangement is that the modification of the program becomes easy.

Following example illustrate this separation process.

Step 1 : Create an header file named **Test.h**. It is as follows

Test.h

```
class Test
{
private:
    double a, b, c;
public:
    void Get_data(double x, double y);
    double Addition();
};
```

Step 2 : Create an implementation file in which the functions declared in header file are defined.

Implementation.cpp

```
#include<iostream.h>
#include "D:\test.h"
```

```
using namespace std;
void Test::Get_data(double x, double y)
{
    a = x;
    b = y;
}
double Test::Addition()
{
    c = a + b;
    return c;
}
```

Step 3 : Create a driver program which will invoke the functionalities defined in the implementation file.

Test.cpp

```
#include <iostream.h>
#include "D:\test.h"
using namespace std;
int main()
{
    Test obj;
    obj.Get_data(10, 20);
    cout << "\n The addition is: " << obj.Addition();
    return 0;
}
```

Step 4 : In order to get the output compile the test.cpp file and implementation .cpp file

Output

The addition is: 30

Part III : Functions

1.25 Functions

SPPU : May-17, Marks 8

Programmer handles the C++ functions using following methods -

1. Definition of function.
2. Call to the function.

In C++ normally we write all the function definitions just before the void main() function.

Let us take some example to understand this concept.

Suppose, I want to perform addition of any two numbers and want to write a function for performing such addition. Then I will write a function sum as follows -

```

void sum() /*definition of the function*/
{
    int a,b,c;
    cout<<"\n Enter The two numbers";
    cin>>a;
    cin>>b;
    c=a+b;
    cout<<"\n The Add ition Of two numbers is "<<c;
}
int main()
{
    sum();/*call to the function*/
    return 0;
}

```

The Definition of the Function

The syntax for the function definition is

```

data_type name_of_function (data_type parameter1,data_type parameter2, ... data_type
parameterm)
{
body for the logic of function
.
.
.
}
```

The definition actually gives the task to be done by the function. In the above program we have given the definition of sum function in which we have accepted the two numbers a and b, performed their addition and printed the result which is stored in variable c.

1.25.1 Function Prototype

A **function prototype** is declaration of function without specifying the function body. That is the function prototype specifies name of function, argument type and return type.

The Call to the Function

The call to the function is given by simply giving the name of the function. The syntax for this is

Name_of_function(parameter₁,parameter₂,...parameter_n);

In above program we invoke the sum function in the main.

There are various types of the function -

1. Passing nothing and returning nothing.

2. Passing the parameters and returning nothing.
3. Passing parameters and returning something.

1.25.2 Argument Passing

There are various ways by which the function can be handled. We can define the functions by passing arguments to them or not passing any argument at all. Following are various approaches of the function handling.

Type 1. Passing Nothing and Returning Nothing

C++ Program

```
#include<iostream>
using namespace std;
void sum() /*definition of the function*/
{
    int a,b,c;
    cout<<"\n Enter The two numbers: ";
    cin>>a;
    cin>>b;
    c=a+b;
    cout<<"\n The Addition Of two numbers is "<<c;
}
int main()
{
    sum();/*call to the function*/
    return 0;
}
```

Output

```
Enter The two numbers: 2 3
The Addition Of two numbers is 5
```

Here, the same above example is repeated. Notice here that no parameter is passed in the function. Similarly no return statement is written in the function sum. Hence we have given the data type for the function sum as void. The void means returning nothing or NULL. It is always good to give the data type for the function main as void, because it is returning nothing. We can pass the argument to main as void to indicate that there is no parameter passed in the function main.

For example :

```
void main(void) can be written instead of main()
```

Type 2. Passing the Parameter and Returning Nothing.

Here we will see a sample C++ code in which the function is written with some parameter.

C++ Program

```
#include<iostream>
using namespace std;
void sum(int x,int y)/*definition*/
{
    int c;
    c=x+y;
    cout<<"\n The Addition is: "<<c;
}
int main()
{
    int a,b;
    cout<<"\n Enter The two numbers: ";
    cin>>a;
    cin>>b;
    sum(a,b);/*call*/
    return 0;
}
```

Output

Enter The two numbers: 5 7

The Addition is: 12

The parameters a and b are passed. In the definition we have interpreted them as x and y. You can take them as a, b respectively or x, y or any other names of your own choice. It makes no difference. It takes them as a and b only. There are actually two methods of parameter passing.

1. Call by Value.

The above example which we have discussed is of parameter passing by call by value. This is called by value because the values are passed.

2. Call by Reference.

In call by reference the parameters are taken by reference. Pointer variables are passed as, parameters.

For example :

C++ Program

```
#include<iostream>
using namespace std;
void sum(int *x,int *y)//function definition
{
    int c;
    c=*x+*y;
```

```

        cout<<"The addition of two numbers is: "<<c;
}
int main()
{
    int a,b;
    void sum(int *,int *); //function declaration
    cout<<"\n Enter The Two Numbers: ";
    cin>>a;
    cin>>b;
    sum(&a,&b); //call to the function
    return 0;
}

```

Output

Enter The Two Numbers: 6 5
The addition of two numbers is: 11

Type 3. Passing the Parameters and Returning from the Function.

In this method the parameters are passed to the function. And function also returns something. Depending upon that something the data type of the function gets decided. That means, if the function is returning an integer value the data type of the function becomes the int, similarly the float, double or char can be data types of the function if that function is returning the float, double or character type variables. For returning any value the keyword return is used. If the function is returning nothing then its data type is supposed to be void.

Let us understand this method by sum example.

C++ Program

```

#include<iostream>
using namespace std;
int sum(int a,int b)
{
    int c=a+b;
    return c; //returning c which is of int type, so data type of sum is int
}
int main()
{
    int a,b,c;
    int sum(int,int); /* Only mentioning of data type is allowed for the parameters*/
    cout<<"\n Enter The Two Numbers: ";
    cin>>a;
    cin>>b;
    c=sum(a,b);
    cout<<"\n The Addition Is = "<<c;
}

```

```

    return 0 ;
}

```

Output

Enter The Two Numbers: 4 5

The Addition Is = 9

Example 1.25.1 Consider the following declaration :

```

class TRAIN
{
    int trainno;
    char dest[20];
    float distance;
public:
    void get(); //To read an object from the keyboard
    void put(); //To write an object into a file
    void show(); //To display the file contents on the monitor
};

```

Complete the member functions definitions

SPPU : May-17, Marks 8

Solution :

```

#include <iostream>
#include <fstream>
using namespace std;
class TRAIN
{
protected:
    int trainno;
    char dest[20];
    float distance;

public:
    void get()
    {
        cout << "\n Enter trainno: ";
        cin >> trainno;
        cout << "\n Enter destination: ";
        cin >> dest;
        cout << "\n Enter distance: ";
        cin >> distance;
    }
    void show()

```

```

{
    ifstream in_obj;
    in_obj.open("test.dat", ios::binary);
    in_obj.read((char*)this, sizeof(TRAIN));

    cout << "\n Train No : " << trainno;
    cout << "\n Destination : " << dest;
    cout << "\n Distance :" << distance;
}
void put() // Member function for write file
{
    ofstream out_obj;
    out_obj.open("test.dat", ios::app | ios::binary);
    out_obj.write((char*)this, sizeof(TRAIN)); //writes current record to file
}
};

int main()
{
    TRAIN obj;
    cout << "\nEnter the Record:";
    obj.get();
    cout << "\tWriting the Record....";
    obj.put();
    cout << "\nThe Record is:";
    obj.show();
    return 0;
}

```

1.26 Accessing Function and Utility Function

- **Access Function**
 - It is a function that can read or display data.
 - Another use of access function is to check the truth or false conditions. Such functions are also called as predicate functions.
- **Utility Function**
 - It is a helper or supporting function that can be used by access function to perform some common activities.
 - It is a private function because it is not intended to use outside the class.
- **C++ Program**

```
#include<iostream>
using namespace std;
class Test
{
private:
    double a, b, c;
```

```

double division(double a, double b)//Utility Function
{
    c = a / b;
    return c;
}
public:
    void Get_data(double x, double y)//Access Function
{
    a = x;
    b = y;
}
int CheckZero(double b)//Access Function
{
    if (b == 0)
        return 1;
    else
        return 0;
}
void Display()//Access Function
{
    if (CheckZero(b)==1)
        cout << "\n Division is not possible!!!";
    else
    {
        c = division(a, b);
        cout << "The division is:" << c << "\n";
    }
}
};

int main()
{
    Test obj;
    obj.Get_data(20, 0);
    obj.Display();
    return 0;
}

```

Output

Division is not possible!!!

Note that in above program, **Get_data** and **Display**, **CheckZero** are the access functions while **division** is an utility function.

1.27 Constructors

SPPU : Dec.-17, Marks 6

Objects need to initialize the variables. Such initialized variables can then be used for processing. If the variables are not been initialized then they hold some **garbage value**. And if such variables are taken for operation then unexpected results may occur. In

order to avoid that the class can include a special function called constructor. The constructor can **automatically be called** whenever a **new object** of this class is created. The constructor will have the **same name** as the **class name**. It should not have any return type.

In C++ there are various ways of using constructors. We will understand with the help of programming examples.

1.27.1 Characteristics of Constructors

Following are some rules that must be followed while making use of constructors -

1. Name of the constructor must be **same as the name of the class** for which it is being used.
2. The constructor must be declared in the **public mode**.
3. The constructor gets invoked automatically when an object gets created.
4. The constructor should not have any return type. Even a **void** type should not be written for the constructor.
5. The constructor can not be used as a member of union or structure.
6. The constructors can have default arguments.
7. The constructors can **not be inherited**. But the derive class can invoke the constructor of base class.
8. Constructors can make use of **new** or **delete** operators for allocating or releasing the memory.
9. Constructors can not be **virtual**.
10. Multiple constructors can be used by the same class.
11. When we declare the constructor explicitly then we must declare the object of that class.

Types of Constructors

Various types of constructors used in C++ are -

1. Default constructor
2. Parameterized constructor
3. Default argument constructor
4. Copy constructor.

Let us discuss them in detail.

1.27.2 Default Constructor

This is the simplest way of defining the constructor. We simply define the constructor without passing any argument to it.

C++ Program

```
#include<iostream>
using namespace std;
class image
{
private:
    int height,width;
public:
    image()
    {
        height=0;
        width=0;
    }
    int area()
    {
        cout<<"Enter the value of height"<<"\n";
        cin>>height;
        cout<<"Enter The value of width"<<"\n";
        cin>>width;
        return (height*width);
    }
};
int main()
{
    image obj1;
    cout<<"The area is :"<<obj1.area()<<endl;
    return 0;
}
```

Output

```
Enter the value of height
10
Enter The value of width
20
The area is : 200
```

1.27.3 Parameterized Constructor

Another way of defining the constructor is by passing the parameters.

We can call the parameterised constructor using

1. Implicit call
2. Explicit call

For example :

Here an object **obj1** gets created by passing the parameters 5 and 3 for the class **image**.

```
image obj1(5,3); <----- Implicit call
image obj1=image(5,3); <----- Explicit call
```

Most commonly use of implicit call is preferred in parameterised constructor. Following program makes use of parameterised constructor.

C++ Program

```
#include<iostream>
using namespace std;
class image
{
    private:
        int height,width;
    public:
        image(int x,int y) //constructor
        {
            height=x;
            width=y;
        }
        int area()
        {
            return (height*width);
        }
};
int main()
{
    image obj1(5,3);
    cout<<"The area is :"<<obj1.area()<<endl;
    return 0;
}
```

Output

The area is :15

Example 1.27.1 Write a class called “arithmetic” having two integer and one character data members. It performs the operation on its integer members indicated by character member(+,-,*,/). For example * indicates multiplication on data members as $d1*d2$. Write a class with all necessary constructors and methods to perform the operation and print the operation performed in format $Ans= d1 \text{ op } d2$. Test your class using `main()`.

Solution :

```
#include<iostream>
using namespace std;
class Arithmetic
```

```
{  
    int d1,d2;  
    char op;  
public:  
  
    Arithmetic(int x,char c,int y)  
    {  
        d1=x;  
        d2=y;  
        op=c;  
    }  
    int operation()  
    {  
        int c;  
        switch(op)  
        {  
            case '+':c=d1+d2;  
                break;  
            case '-':c=d1-d2;  
                break;  
            case '*':c=d1*d2;  
                break;  
            case '/':c=d1/d2;  
                break;  
        }  
        return c;  
    }  
};  
int main()  
{  
    Arithmetic obj1(10,'+',20);  
    Arithmetic obj2(20,'-',10);  
    Arithmetic obj3(10,'/',5);  
    Arithmetic obj4(10,'*',20);  
    cout<<"\n Addition of 10+20= "<<obj1.operation();  
    cout<<"\n Subtraction of 20-10= "<<obj2.operation();  
    cout<<"\n Division of 10/5= "<<obj3.operation();  
    cout<<"\n Multiplication of 10*20= "<<obj4.operation();  
    return 0;  
}
```

Output

Addition of 10+20= 30
Subtraction of 20-10= 10
Division of 10/5= 2
Multiplication of 10*20= 200

1.27.4 Default Argument Constructor

The constructor can be defined by passing the default arguments to it. Consider following example

```
#include <iostream>
using namespace std;

class Test
{
private:
    int a;
    int b;
    int c;
public :
    Test(int x=10,int y=20); // declaration of constructor with default arguments
    void display()
    {
        cout<<"\n a= "<<a;
        cout<<"\n b= "<<b;
        cout<<"\n c= "<<c;
    }
};

Test::Test(int x,int y)
{
    a=x;
    b=y;
    c=a+b;
}

int main()
{
```

Output

```
a= 10
b= 20
c= 30
```

```
a= 100
b= 20
c= 120
```

1.27.5 Copy Constructor

The copy constructor is called whenever a new variable is created from an object.

In C++ the copy constructor is created when the copy of existing object needs to be created. Usually the compiler creates a copy constructor for each class when no copy constructor is defined. Such a constructor is called **implicit constructor** and when the copy constructor is explicitly created in the program then it is called **explicit constructor**.

Definition : Copy constructor is a special type of constructor in which new object is created as a copy of existing object.

In other words in copy constructor one object is initialized by the other object. The general form of copy constructor is -

```
classname (classname &object)
{
    //body of the constructor
}
```

While invoking the copy constructor we will use following syntax

```
classname new_object_name(old_object_name);
```

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

C++ Program

```
#include<iostream>
using namespace std;
class test
{
int x;
public:
//default constructor
test();
//parameterized constructor
test(int val)
{
    x=val;
}
//copy constructor
test(test &obj)
{
    x=obj.x;//entered the value in obj.x
}
void show()
{
```

```

        cout<<x;
    }
};

int main()
{
    int val;
    cout<<"Enter some number"<<endl;
    cin>>val;
    test Old(val);
    //call for copy constructor
    test New(Old); <---- object 'Old' is passed as argument to object 'New'
    cout<<"\n The original value is: ";
    Old.show();
    cout<<"\n The New copied value is: ";
    New.show();
    cout<<endl;
    return 0;
}

```

Output

Enter some number
500

The original value is : 500
The New copied value is : 500

In above program there are three constructors first one is the simple constructor and second constructor is a constructor in which parameter is passed. The copy constructor is always declared by passing reference parameter to it. And the reference variable is given by '&'. We can not pass the parameter by value to copy constructor.

Review Question

1. Write a program which uses default constructor, parameterized constructor, and destructor.

SPPU : Dec.-17, Marks 6

1.28 Destructor

SPPU : May-19, Marks 6

The destructor is called when the object is destroyed. The object can be destroyed automatically when the scope of the objects end (i.e. when the function ends) or explicitly by using operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

We normally use the destructor when the object assigns the dynamic memory in its lifetime.

C++ Program

```
#include<iostream>
using namespace std;
class image
{
    private:
        int *height,*width;
    public:
        image(int,int);//constructor
        ~image();//destructor
        int area();//regular function
};
image::image(int x,int y)
{
    height=new int;//assigns memory dynamically using 'new'
    width=new int;
    *height=x;
    *width=y;
}
image::~image()
{
    delete height;//destroys the memory using 'delete'
    delete width;
}
int image::area()
{
    return (*height* *width);
}
int main()
{
    image obj1(10,20);
    cout<<"The area is :"<<obj1.area()<<endl;
    return 0;
}
```

Output

The area is : 200

Review Question

- What do you mean by constructor and destructor? Write appropriate C++ program which uses copy constructor

SPPU : May-19, Marks 6

1.29 Objects and Memory Requirements

- The memory space objects is allocated when they are declared.
- The members functions are created and placed in the memory only when they are defined as a part of a class specification.

- As the objects are using the same **member functions** of the belonging class, **no separate memory space** is created for these function when objects are created.
- Only for the **member variables** the **separate memory space** is created for **each object**.

Fig. 1.29.1 illustrates this concept

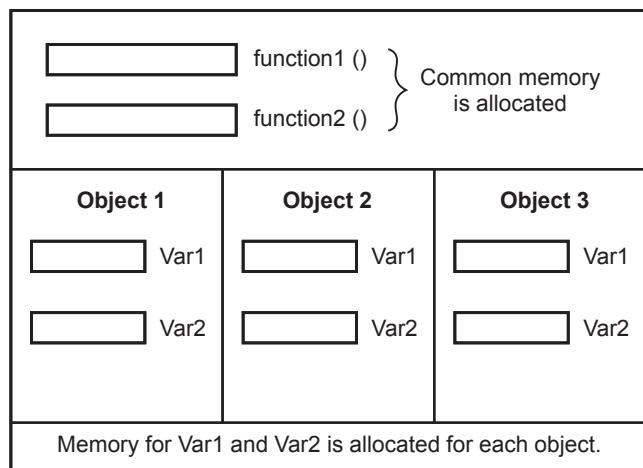


Fig. 1.29.1 Memory allocation for objects

The memory required by any object is dependent upon following factors

- size of non static data members of the class.
- order of data members.
- Byte padding.

Consider following example -

```
class Test
{
    float a;
    int b;
    static int c;
    char d;
};
```

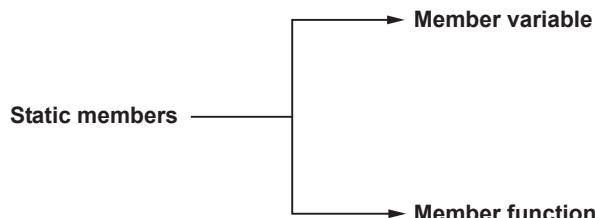
Test **obj**;

For **obj** size will be `sizeof(a)+sizeof(b)+sizeof(d)=4+2+1=7 bytes`. Note that the size of static members is not included in this calculation. Static members are really not part of the class object. They won't be included in object's layout.

1.30 Static Members : Variable and Functions

SPPU : May-17, Dec.-19, Marks 4

- A class contains two types of static members -



- The members of the class must be declared with the keyword **static**.
- The static data members have the same properties as that of **global variables**.
- When we declare regular variables then multiple copies are created for accessing them by each corresponding object but when the keyword static is associated with any class variable then that means only single copy of that variable must be created and all other objects must share that copy.
- The static data members can be accessed without any instantiation of class as an object.
- The static member functions are not used very frequently in programs. All static variables are **initialized to zero** before first object is created.
- The **syntax of static data declaration** and definition is as given below -

```
class test {
    private:
        static int count; //declaration
        ...
};

int test::count=100; //definition without instantiation of class as object
```

C++ program using static keyword

```
#include<iostream>
using namespace std;
void fun()
{
    static int cnt=0;
    cout << "\ncount = " << cnt;
    cnt++;
}
int main()
{
    for (int i = 0; i < 5; i++)
        fun();
    return 0;
}
```

Output

```
count = 0
count = 1
count = 2
count = 3
count = 4
```

C++ Program without using static

```
#include<iostream>
using namespace std;
void fun()
{
    int cnt=0;
    cout << "\ncount = " << cnt;
    cnt++;
}
int main()
{
    for (int i = 0; i < 5; i++)
        fun();
    return 0;
}
```

Output

```
count = 0
```

In above program, we can clearly understand that the single copy of variable is created and it is accessed each time.

understand that the single copy of

- The static function can not refer to any non static member data in its class. The static function can access only static and class-specific data. The syntax of static function is as given below -

```
class test()
{
    private:
        ...
    public:
        static int fun(); //static function definition
    {

    }
};

int main()
{
    ...
    test::fun(); //function call
    ...
}
```

The program having static class members is as given below -

C++ Program

```
#include <iostream>
using namespace std;
class count
{
    private:
        int number;
        static int total;
                    // declaration:static data
    public:
        count();           // initialize
        int get_number(); // get a number
        static int get_total(); // get total count
    };
count::count()           //initialize one count
{
    number = 100 + total++;
}
int count::get_number() // get number
{
    return number;
}
```

```

int count::get_total()    // get total counts
{
    return total;
}
int count::total = 0;    // definition of static data
int main()
{
    //will print initial count
    cout << "Total count = " << count::get_total() << endl;
    count a, b, c;
    //will increment the count 3 times
    cout << " a=" << a.get_number() << endl;
    cout << " b=" << b.get_number() << endl;
    cout << " c=" << c.get_number() << endl;
    //will print the final incremented count
    cout << "Total count = " << count::get_total() << endl;
    return 0;
}

```

Output

```

Total count = 0
a=100
b=101
c=102
Total count = 3

```

Review Questions

1. What is static member function ?

SPPU : Dec.-19, Marks 3

2. Explain the significance of the keyword static in programming.

SPPU : May-17, Marks 4

1.31 Inline Function

SPPU : Dec.-16, 18, 19, May-19, Marks 6

When we define a function normally the compiler makes a copy of that definition in the memory. And when a call to that function is made the compiler jumps to those copied instructions and when the function returns, the execution resumes from the next line in the calling function. Hence if we make a call to that function for 5 times then each time the copied block of function in the memory will be referred by the compiler. This also means that there is only one copy of the function definition in the memory and on each call to that function the same copy is referred.

Now if there are many calls to that function and function contains very few lines of code then such a jumping to memory becomes a performance overhead for the compiler. It ultimately slows down the execution of the program. Hence the solution is to make the function inline.

Definition : The inline function is a function whose code is copied in place of each function call. The inline specifies that the compiler should insert the complete body of function in every context where the function is used.

Programming example :

```
#include<iostream.h>
inline largest(int x,int y,int z)
{
    if(x>y&&x>z)
    {
        cout<<" First number is greatest and it is: "<<x;
    }
    else if(y>x&&y>z)
    {
        cout<<" Second number is greatest and it is: "<<y;
    }
    else
    {
        cout<<" Third number is greatest and it is: "<<z;
    }
}
int main()
{
    int a,b,c;
    cout<<" Enter first number: ";
    cin>>a;
    cout<<" Enter second number: ";
    cin>>b;
    cout<<" Enter third number: ";
    cin>>c;
    largest(a,b,c);
    cout<<endl;
    return 0;
}
```

Output

```
Enter first number: 20
Enter second number: 10
Enter third number: 30
Third number is greatest and it is: 30
```

Situation in which Inline function may not work :

Following are the situations in which the inline functions may not work

1. For the functions that contain the static variables.
2. For the functions returning values if - for loop, switch or go to exists.

3. For the functions not returning a function and if return statement exists.

4. If the inline functions are recursive function.

Inline Function Vs. Macro :

- The major difference between inline functions and macros is the way they are handled. Inline functions are analyzed by the compiler, whereas macros are expanded by the C++ preprocessor.
- Macro invocation do not perform type checking or do not check that arguments are well-formed whereas the inline function can do these tasks.
- Macro can not return any value whereas the inline function can return some value.
- Error correction is simpler in case of inline function as compared to Macro function.

Advantages of Inline Functions

- 1) It does not require function calling overhead.
- 2) It also saves overhead of return call from a function.
- 3) It enhances the compile time performance.

Example 1.31.1 Define class number which has inline function mult () and cube () for calculating the multiplication of 2 double numbers given and cube of the integer number given.

SPPU : Dec.-16, Marks 4

Solution :

```
#include <iostream>
using namespace std;
    inline double Mult(double x, double y)
    {
        return (x*y);
    }
    inline int cube(int n)
    {
        return n*n*n;
    }
};

int main()
{
    number obj;
    cout << "Mult (20,10): " << obj.Mult(20, 10) << endl;
    cout << "Cube (10): " << obj(cube(10)) << endl;
    return 0;
}
```

Output

Mult (20,10): 200
Cube (10): 1000

Review Questions

1. What are inline functions ? What are their advantages ? Give an example.

SPPU : Dec.-18, Marks 3, May-19, Marks 6

2. What is Inline function ? Explain with suitable program.

SPPU : Dec.-19, Marks 4

1.32 Friend Function

SPPU : Dec.-16, 18, May-18, 19, Marks 6

The friend function is a function that is **not a member function** of the class but it can **access the private and protected members** of the class.

The friend function is given by a keyword *friend*.

These are special functions which are declared anywhere in the class but have given **special permission to access the private members** of the class.

C++ Program

```
#include<iostream>
using namespace std;
class test
{
    int data;
    friend int fun(int x); //declaration of friend function
public:
    test()//constructor
    {
        data = 5;
    }
    int fun(int x)
    {
        test obj;
        //accessing private data by friend function
        return obj.data + x;
    }
    int main()
    {
        cout << "Result is = "<< fun(4)<< endl;
        return 0;
    }
}
```

Output

Result is = 9

1.32.1 Properties of Friend Functions

Following are some **properties of friend functions** -

1. The friend function is **not defined within the scope of the class**.
2. It **cannot be invoked** by the **object** of particular class.
3. It can be invoked **like a normal function**.
4. This function **can access the private members** of the class.
5. Usually the objects of some class are passed as an argument to the friend function.
6. It must be declared with the keyword **friend**.

Review Questions

1. *What is friend function ? Explain with suitable example.*

SPPU : Dec.-16, Marks 2, May-19, Marks 4

2. *What are friend functions and static functions ?*

SPPU : May-18, Marks 6, Dec.-18, Marks 4



Unit - II

2

Inheritance and Pointers

Syllabus

Inheritance - Base Class and derived Class, protected members, relationship between base Class and derived Class, Constructor and destructor in Derived Class, Overriding Member Functions, Class Hierarchies, Public and Private Inheritance, Types of Inheritance, Ambiguity in Multiple Inheritance, Virtual Base Class, Abstract class, Friend Class, Nested Class.

Pointers : declaring and initializing pointers, indirection Operators, Memory Management : new and delete, Pointers to Objects, this pointer, Pointers Vs Arrays, accessing Arrays using pointers, Arrays of Pointers, Function pointers, Pointers to Pointers, Pointers to Derived classes, Passing pointers to functions, Return pointers from functions, Null pointer, void pointer.

Contents

2.1	Basic Concept of Inheritance	
2.2	Base Class and Derived Class	
2.3	Public and Private Inheritance	
2.4	Protected Members	Dec.-17, Marks 6
2.5	Relationship between Base Class and Derived Class	
2.6	Constructor and Destructor in Derived Class	
2.7	Overriding Member Functions	
2.8	Class Hierarchies	
2.9	Types of Inheritance	Dec.-16, 19, Marks 6
2.10	Ambiguity in Multiple Inheritance	May-19, Marks 6
2.11	Virtual Base Class	
2.12	Abstract Class	May-19, Marks 2
2.13	Friend Class	
2.14	Nested Class	
2.15	Pointer - Indirection Operator	
2.16	Declaring and Initializing Pointers	
2.17	Memory Management : New and Delete	Dec.-18, May-19, Marks 6

2.18	Pointers to Object
2.19	this Pointers Dec.-16, 19, Marks 2
2.20	Pointers Vs Arrays
2.21	Accessing Arrays using Pointers
2.22	Pointer Arithmetic Dec.-18, Marks 4
2.23	Arrays of Pointers
2.24	Function Pointers Dec.-17, May-18, 19, Marks 5
2.25	Pointers to Pointers
2.26	Pointers to Derived Classes
2.27	Null Pointer
2.28	void Pointer

Part I : Inheritance

2.1 Basic Concept of Inheritance

Definition : Inheritance is a property in which data members and member functions of some class are used by some other class.

Inheritance allows the reusability of the code in C++.

2.2 Base Class and Derived Class

- The class from which the data members and member functions are used by another class is called the **base class**.
- The class which uses the properties of base class and at the same time can add its own properties is called **derived class**.
- There are three types of access specifier or qualifier using which the members of the class are accessed by the other class -
 1. Private
 2. Public
 3. Protected
- If a base class has **private** members then those members are not accessible to derived class.
- **Protected members** are public to derived classes but **private** to rest of the program.
- **Public members** are accessible to all.
- The derived class can inherit base class publicly or privately. The notation used for inheritance is :

For example -

```
class d1:public b1  
class d2:private b2
```

The first line indicates that there are two classes **d1** and **b1**. It means "the derived class **d1** inherits the base class **b1** publicly".

The second line indicates that there are two classes **d2** and **b2**. It means "the derived class **d2** inherits the base class **b2** privately."

- The base class and derived class can generate their own objects. These objects differ from each other.

2.3 Public and Private Inheritance

The **access specifier** such as **public, private or protected** determines how elements of base class are inherited by the derived class.

When the access specifier for the inherited base class is public then all public members of the base class become public members of the derived class.

If the access specifier is private then all public members of the base class become private members of the derived class.

The private members of base class are inaccessible to derived class. The public members of the base class become private members to derived class but those are accessible to derived class. If the **access specifier is not present** then it is taken as **private** by default.

Inheriting Base Class in Public Mode

Example :

```
#include <iostream>
using namespace std;
class Base
{
    int x;
public:
    void set_x(int n)
    {
        x = n;
    }
    void show_x( )
    {
        cout << "\n x= " << x;
    }
};

// Inherit as public
class derived : public Base
{
    int y;
public:
    void set_y(int n)
    {
        y = n;
    }
    void show_y()
    {
        cout << "\n y= " << y;
    }
};
```

```

int main()
{
    derived obj;//object of derived class
    int x, y;
    cout<<"\n Enter the value of x";
    cin>>x;
    cout<<"\n Enter the value of y";
    cin>>y;
    //using obj of derived class base class member is accessed
    obj.set_x(x);
    obj.set_y(y); // access member of derived class
    obj.show_x(); // access member of base class
    obj.show_y(); // access member of derived class
    return 0;
}

```

Output

Enter the value of x30

Enter the value of y70

```

x= 30
y= 70

```

In above program the *obj* is an object of derived class. Using *obj* we are accessing the member function of base class. The derived class inherits base class using an access specifier public. Now following program will contain error.

Inheriting Base Class in Private Mode**Example :**

```

#include <iostream>
using namespace std;
class Base
{
    int x;
public:
    void set_x(int n)
    {
        x = n;
    }
    void show_x( )
    {
        cout <<"\n x= "<<x;
    }
};

```

```
// Inherit as private
class derived : private Base
{
    int y;
public:
    void set_y(int n)
    {
        y = n;
    }
    void show_y()
    {
        cout << "\n y= " << y;
    }
};

int main()
{
    derived obj;//object of derived class
    int x, y;
    cout << "\n Enter the value of x";
    cin >> x;
    cout << "\n Enter the value of y";
    cin >> y;
    obj.set_x(x); // error: not accessible
    obj.set_y(y);
    obj.show_x(); // access member of base class
    obj.show_y(); // error: not accessible
    return 0;
}
```

As indicated by the comments the above program will generate error messages “*not accessible*”. This is because the derived class inherits the base class privately. Hence the public members of base class become private to derived class.

2.4 Protected Members

SPPU : Dec.-17, Marks 6

The *protected* access specifier is equivalent to the *private* specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible. Following program illustrates the same.

```
#include <iostream>
using namespace std;
class Base
{
protected:
    int x;
```

```
public:  
void set_x(int n)  
{  
    x = n;  
}  
void show_x()  
{  
    cout << "\n x= " << x;  
}  
};  
class derived : public Base  
{  
int y;  
public:  
void set_y(int n)  
{  
    y = n;  
}  
void show_xy()  
{  
    //can access protected member in derived class  
    cout << "\nderived::x = " << x;  
    cout << "\n y= " << y;  
}  
};  
  
int main()  
{  
derived obj;  
int x, y;  
cout << "\nEnter the value of x";  
cin >> x;  
cout << "\nEnter the value of y";  
cin >> y;  
obj.set_x(x);  
obj.set_y(y); // access member of derived class  
obj.show_x();  
obj.show_xy(); // access member of derived class  
cout << "\n Setting another value to x" << endl;  
//protected members become private to outside base and derived class  
obj.x=100; //error: not accessible  
return 0;  
}
```

Review Question

1. Explain public, private and protected keywords using program

SPPU : Dec.-17, Marks 6

2.5 Relationship between Base Class and Derived Class

Inheritance is an important feature in object oriented programming that allows the reusability of the code.

The fundamental idea behind the inheritance is that - make use of data members and member functions of base class in derived class along with some additional functionality present in derived class. Following example illustrates the relationship between base class and derived class. In the following **program inheritance is not used**.

```
#include<iostream>
using namespace std;
class Base
{
public:
    char str1[10], str2[10];
    void Input_Data()
    {
        cout << "\n Enter String1 : ";
        cin >> str1;
        cout << "\n Enter String2 : ";
        cin >> str2;
    }
    void display()
    {
        cout << " \n String1 : " << str1;
        cout << " \n String2 : " << str2;
    }
};
class Derived
{
public:
    char str1[10], str2[10];
    void Input_Data()
    {
        cout << "\n Enter String1 : ";
        cin >> str1;
        cout << "\n Enter String2 : ";
        cin >> str2;
    }
    void display()
    {
        cout << " \n String1 : " << str1;
        cout << " \n String2 : " << str2;
    }
}
```

This block of code is repeated for the derived class if the inheritance is not used

```

int getlength(char s[10])
{
    int i;
    for (i = 0; s[i] != '\0'; i++);
    return i - 1;
}
void compare()
{
    int i, j, flag = 0;
    int n1 = getlength(str1);
    int n2 = getlength(str2);
    for (i = 0, j = 0; i<=n1, j<=n2; i++, j++)
    {
        if (str1[i] != str2[j])
            flag = 1;
    }
    if (flag == 1)
        cout << "\n Two strings are not equal";
    else
        cout << "\n Two strings are equal";
}
int main()
{
    Derived d_obj;
    d_obj.Input_Data();
    d_obj.display();
    d_obj.compare();
    return 0;
}

```

Output

Enter String1 : hello
 Enter String2 : hello
 String1 : hello
 String2 : hello

Two strings are equal

Program explanation : In above code, we have created two independent classes. The purpose of this code is to compare two strings. In Derived class, we need to repeat the code same for input and display functions in order to make the function *compare* working.

Secondly if there are multiple derived classes, and if there is a need for some modifications in the code, then those modifications need to be carried out in all the derived classes.

If we use inheritance then this kind of repetition can be avoided. Following program illustrates this idea.

```
#include<iostream>
using namespace std;
class Base
{
public:
    char str1[10], str2[10];
    void Input_Data()
    {
        cout << "\n Enter String1 : ";
        cin >> str1;
        cout << "\n Enter String2 : ";
        cin >> str2;
    }
    void display()
    {
        cout << "\n String1 : " << str1;
        cout << "\n String2 : " << str2;
    }
};

class Derived :public Base
{
public:
    int getlength(char s[10])
    {
        int i;
        for (i = 0; s[i] != '\0'; i++);
        return i - 1;
    }
    void compare()
    {
        int i, j, flag = 0;
        int n1 = getlength(str1);
        int n2 = getlength(str2);
        for (i = 0, j = 0; i <= n1, j <= n2; i++, j++)
        {
            if (str1[i] != str2[j])
                flag = 1;
        }
        if (flag == 1)
            cout << "\n Two strings are not equal";
        else
            cout << "\n Two strings are equal";
    }
};
```

```
int main()
{
    Derived d_obj;
    d_obj.Input_Data();
    d_obj.display();
    d_obj.compare();
    return 0;
}
```

Output

Enter String1 : hello

Enter String2 : hello

String1 : hello

String2 : hello

Two strings are equal

Program Explanation : We have inherited methods Input_Data() and display() from the base class and derived class simply contains the method for comparing two strings.

Thus all the commonly used functionalities are defined in base class and only the additional functionalities that are required for corresponding derived classes are defined in respective derived class.

Advantages of Inheritance

One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses.

1. **Reusability** : The base class code can be used by derived class without any need to rewrite the code.
2. **Extensibility** : The base class logic can be extended in the derived classes
3. **Data hiding** : Base class can decide to keep some data private so that it cannot be altered by the derived class.
4. **Overriding** : With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class

Review Questions

1. Explain the relationship between base class and derived class.
2. What are the advantages of inheritance ?

2.6 Constructor and Destructor in Derived Class

- When we create an object for derived class then first of all the Base class constructor is called and after that the Derived class constructor is called.
- The reason behind this is that the Derived class inherits from the Base class, both the Base class and Derived class constructors will be called when a Derived class object is created.
- When the main function finishes running, the derived class's destructor will get called first and after that the Base class destructor will be called.
- This is also called as **chain of constructor calls**.

Following code illustrates this execution order.

```
#include<iostream.h>
class Base
{
public:
    Base()
    {
        cout << "Base constructor" << endl;
    }
    ~Base()
    {
        cout << "Base destructor" << endl;
    }
};

class Derived:public Base
{
public:
    Derived()
    {
        cout << "Derived constructor" << endl;
    }

    ~Derived()
    {
        cout << "Derived destructor" << endl;
    }
};

void main()
{
    Derived obj;
```

Output

Base constructor
Derived constructor
Derived destructor
Base destructor

Review Question

1. Explain the constructor and destructor execution in derived class.

2.7 Overriding Member Functions

Definition : Redefining a function in a derived class is called function overriding.

Function overloading means within the class we can declare same function name, but arguments and return types are different and function overriding means the function name is same but the task carried out with it is different. For example -

C++ Program

```
#include <iostream.h>
class A
{
    private:
        int a,b;
    public:
        void get_msg()
        {
            a=10;
            b=20;
        }
        void print_msg()
        {
            int c;
            c=a+b;//performing addition
            cout<<"\n C(10+20)= "<<c;
            cout<<"\n I'm print_msg() in class A";
        }
};
class B : public A
{
    private:
        int a,b;
    public:
        void set_msg()
        {
```

```

        a=100;
        b=10;
    }
    void print_msg()
    {
        int c;
        c=a-b;//performing subtraction in this function
        cout<<"\n\n C(100-10) = "<<c;
        cout<<"\n I'm print_msg() in class B ";
    }
};

void main()
{
    A obj_base;
    B obj_derived;
    obj_base.get_msg();
    obj_base.print_msg();//same function name
    obj_derived.set_msg();
    obj_derived.print_msg();//but different tasks
}

```

Output

C(10+20)= 30
I'm print_msg() in class A

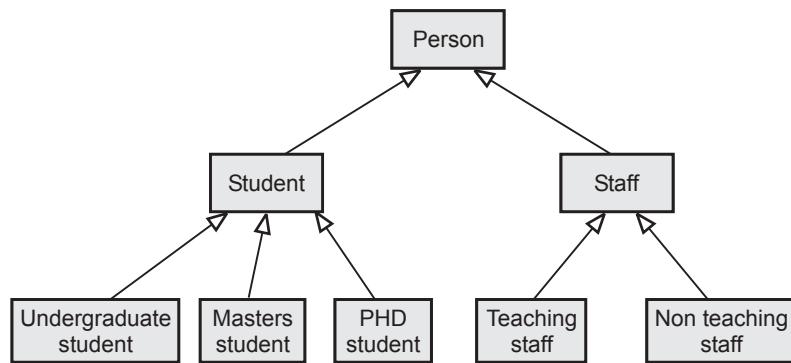
C(100-10) = 90
I'm print_msg() in class B

Program Explanation : In above program we have written two functions **print_msg()** by the same name but performing different operation. The **print_msg()** function in **class A** (i.e. base class) is performing addition of two numbers and the **print_msg()** function in **class B** is performing the subtraction of two numbers. Both the functions have **same return type, same name and no parameters** but their role is changing. This mechanism of changing the task of some function in derived class is called **function overriding**.

2.8 Class Hierarchies

Inheritance is a mechanism in which using base class, various classes can be derived. Following diagram represents the class hierarchy. The class hierarchy is normally represented by **class diagram**

Example :



The implementation of class hierarchy is possible using **hierarchical inheritance**.

2.9 Types of Inheritance

SPPU : Dec.-16, 19, Marks 6

Various types of inheritance are -

- 1) Single inheritance
- 2) Multilevel inheritance
- 3) Multiple inheritance
- 4) Hybrid inheritance
- 5) Hierarchical inheritance

2.9.1 Single Inheritance

In single inheritance there is **one parent per derived class**. This is the most common form of inheritance.

The simple program for such inheritance is -

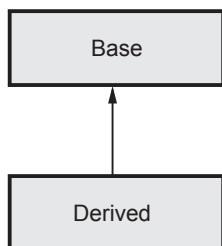


Fig. 2.9.1 Single inheritance

C++ Program

```
#include <iostream>
using namespace std;
class Base
{
public:
    int x;
void set_x(int n)
{
    x = n;
}
void show_x( )
{
    cout<<"\n\t Base class ... ";
    cout <<"\n\t x= "<<x;
}
};

class derived : public Base
{
int y;
public:
void set_y(int n)
{
    y = n;
}
void show_xy()
{
    cout<<"\n\n\t Derived class ... ";
    cout<<"\n\t x = "<<x;
    cout <<"\n\t y = "<<y;
}
};

int main()
{
derived obj;
int x, y;
cout<<"\n Enter the value of x";
cin>>x;
cout<<"\n Enter the value of y";
cin>>y;
obj.set_x(x);//inherits base class
obj.set_y(y); // access member of derived class
obj.show_x();//inherits base class
obj.show_xy(); // access member of derived class
return 0;
}
```

Output

```

Enter the value of x 10
Enter the value of y 20
    Base class ...
        x= 10

    Derived class ...
        x = 10
        y = 20

```

2.9.2 Multi - Level Inheritance

When a derived class is derived from a base class which itself is a derived class then that type of inheritance is called multilevel inheritance.

For example - If class A is a base class and class B is another class which is derived from A, similarly there is another class C being derived from class B then such a derivation leads to multilevel inheritance.

The implementation of multilevel inheritance is as given below -

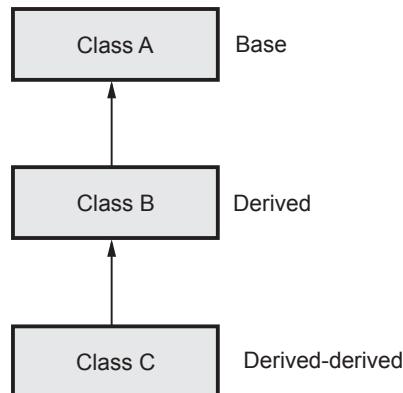


Fig. 2.9.2 Multilevel inheritance

C++ Program

```

#include<iostream>
using namespace std;
class A
{
protected:
    int x;
public:
    void get_a(int);
    void put_a();
};
void A::get_a(int a)
{

```

```
        x=a;
    }
void A::put_a()
{
    cout<<"\n The value of x is "<<x;
}
class B:public A
{
protected:
    int y;
public:
    void get_b(int);
    void put_b();
};
void B::get_b(int b)
{
    y=b;
}
void B::put_b()
{
    cout<<"\n The value of y is "<<y;
}
class C:public B
{
    int z;
public:
    void display();
};
void C::display()
{
    z=y+10;
    put_a();//member of class A
    put_b();//member of class B
    cout<<"\n The value of z is "<<z;
}
int main()
{
    C obj;//object of class C
    //accessing class A member via object of class C
    obj.get_a(10);
    //accessing class B member via object of class C
    obj.get_b(20);
    ///accessing class C member via object of class C
    obj.display();
    cout<<endl;
    return 0;
}
```

Output

```
The value of x is 10
The value of y is 20
The value of z is 30
```

In above program we have declared 3 classes namely A, B and C. In these classes values to variables x, y and z are assigned.

In class C, which is actually derived from a derived class B(derived from A) a display() function is written. Note that the members of class A and B are accessible in class C as it is a derived class.

```
z=y+10;
put_a();//member of class A
put_b();//member of class B
cout<<"\n The value of z is "<<z;
```

Similarly in *main()* function we have created an object of class C.

```
C obj;
```

And now using this *obj* we can access the member of any class.

Thus the multilevel inheritance is achieved.

2.9.3 Multiple Inheritance

In multiple inheritance the derived class is derived from more than one base class.

The implementation of multiple inheritance is as shown in Fig. 2.9.3.

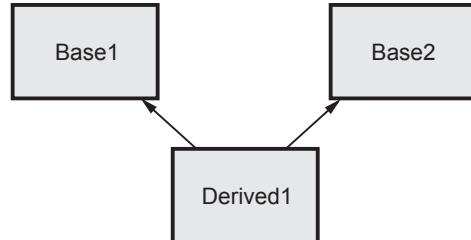


Fig. 2.9.3 Multiple inheritance

```
#include <iostream>
using namespace std;
class Operation
{
protected:
    int x, y;
public:
    void set_values (int a, int b)
    {
        x=a;
        y=b;
    }
};
```

```
class Coutput
{
public:
    void display (int i);
};

void Coutput::display (int i)
{
    cout << i << endl;
}
//product class inherits two base classes -
//Operation and Coutput
class product: public Operation, public Coutput
{
public:
    int function ()
    {
        return (x * y);
    }
};
//sum class inherits two base classes -
//Operation and Coutput
class sum: public Operation, public Coutput
{
public:
    int function ()
    {
        return (x + y);
    }
};

int main ()
{
    product obj_pr;//object of product class
    sum obj_sum;//object of sum class
    obj_pr.set_values (10,20);
    obj_sum.set_values (10,20);
    cout<<"\n The product of 10 and 20 is "<<endl;
    obj_pr.output (obj_pr.function());
    cout<<"\n The sum of 10 and 20 is "<<endl;
    obj_sum.output (obj_sum.function());
    return 0;
}
```

Output

The product of 10 and 20 is

200

The sum of 10 and 20 is
30

In above program there are two classes *Operation* and *Coutput*. The derived class *product* is derived from both *Operation* and *Coutput* classes. Similarly the derived class *sum* is derived from two classes : *Operation* and *Coutput*.

Then in main function *obj_pr* is an object created for class *product* and *obj_sum* is an object created for class *sum*. Thus multiple inheritance is achieved.

2.9.4 Hybrid Inheritance

When two or more types of inheritances are combined together then it forms the hybrid inheritance. The following Fig. 2.9.4 represents the typical scenario of hybrid inheritance.

The following implementation shows that multiple and multilevel inheritance is combined together to form a hybrid inheritance.

C++ Program

```
#include<iostream>
using namespace std;
class A
{
protected:
    int x;
public:
    void get_a(int);
    void put_a();
};
void A::get_a(int a)
{
    x=a;
}
void A::put_a()
{
    cout<<"\n The value of x is "<<x;
}
```

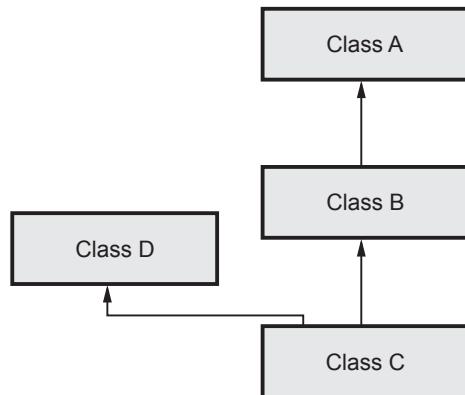


Fig. 2.9.4 Hybrid inheritance

```
class B:public A
{
protected:
    int y;
public:
    void get_b(int);
    void put_b();
};

void B::get_b(int b)
{
    y=b;
}
void B::put_b()
{
    cout<<"\n The value of y is "<<y;
}

class D
{
protected:
    int t;
public:
    void get_d(int);
    void put_d();
};

void D::get_d(int d)
{
    t=d;
}
void D::put_d()
{
    cout<<"\n The value of t is "<<t;
}

//multiple inheritance added in the multilevel inheritance
class C:public B,public D
{
    int z;
public:
    void display();
};

void C::display()
{
    z=y+t+10;
    put_a();//member of class A
    put_b();//member of class B
    put_d();//member of class D
    cout<<"\n The value of z is "<<z;
```

```

}

int main()
{
    C obj;//object of class C
    //accessing class A member via object of class C
    obj.get_a(10);
    //accessing class B member via object of class C
    obj.get_b(20);
    ///accessing class C member via object of class C
    obj.get_d(30);
    obj.display();
    cout<<endl;
    return 0;
}

```

Output

The value of x is 10
The value of y is 20
The value of t is 30
The value of z is 60

2.9.5 Hierarchical Inheritance

Hierarchical inheritance is a kind of inheritance in which one or more classes are derived from the common base class. For example -

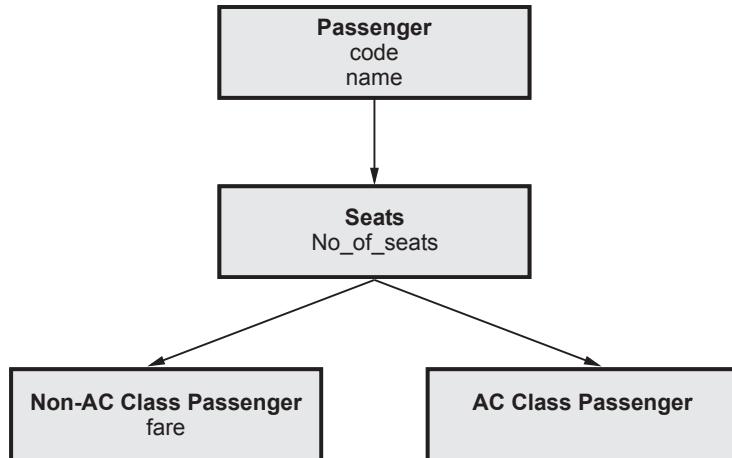


Fig. 2.9.5 Hierarchical inheritance

In this type of inheritance the subclass can inherit the properties of its parent classes and at the same time it can add its new features. The subclass can serve as a base class for the lower level classes.

The C++ program demonstrating this type of inheritance is as shown below -

```
#include<iostream>
using namespace std;
class Passenger
{
    int code;
    char name[20];
public:
    void getPassenger()
    {
        cout<<"\nEnter the code and name ";
        cin>>code>>name;
    }
    void ShowDetails()
    {
        cout<<"\nCode: "<<code;
        cout<<"\nName: "<<name;
    }
};
class Seats:public Passenger
{
    int NoOfSeats;
public:
    void getSeats()
    {
        cout<<"\nEnter the Number of Seats";
        cin>>NoOfSeats;
    }
    void Display_Reservation()
    {
        cout<<"\nNumber of Seats: "<<NoOfSeats;
    }
};
class AC_Class:public Seats
{
    double fare;
public:
    void getFare()
    {
        cout<<"\nEnter the fare";
        cin>>fare;
    }
    void DisplayFare()
    {
        cout<<"\nType: AC Class Reservation";
        cout<<"\nFare Amount: "<<fare;
```

```
    }
};

class NonAC_Class:public Seats
{
public:
    void DisplayClass()
    {
        cout<<"\nType: Non AC Class Reservation";
    }
};

using namespace std;
int main()
{
    int i,m,n,choice;
    AC_Class a[10];
    NonAC_Class na[10];
    cout<<"\nEnter the number of AC Class Passengers ";
    cin>>m;
    for(i=0;i<m;i++)
    {
        cout<<"\nEnter Details of Passenger "<<i+1;
        a[i].getPassenger();
        a[i].getSeats();
        a[i].getFare();
        return 0;
    }
    cout<<"\nEnter the number of Non-AC Class Passengers ";
    cin>>n;
    for(i=0;i<n;i++)
    {
        cout<<"\nEnter the Details of Passenger "<<i+1;
        na[i].getPassenger();
        na[i].getSeats();
    }
    while(1)
    {
        cout<<"\n Displaying Details";
        cout<<"\n 1. AC Class \n 2. Non AC Class\n 3. Exit\n";
        cout<<"\nEnter Choice";
        cin>>choice;
        switch(choice)
        {
            case 1:for(i=0;i<m;i++)
            {
                a[i].ShowDetails();
                a[i].Display_Reservation();
                a[i].DisplayFare();
            }
        }
    }
}
```

```

        }
        break;

case 2:for(i=0;i<n;i++)
{
    na[i].ShowDetails();
    na[i].DisplayClass();
    na[i].Display_Reservation();
}
break;
case 3:exit(0);
}
}
}

```

Review Questions

1. Write short notes on types of inheritance with respect to :

(i) Single (ii) Multiple (iii) Hierarchical

SPPU : Dec.-19, Marks 6

2. Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors

SPPU : Dec.-16, Marks 4

2.10 Ambiguity in Multiple Inheritance

SPPU : May-19, Marks 6

- Ambiguity is the problem that arise in multiple inheritance. It is also called as **diamond problem**.

Consider the following Fig. 2.10.1.

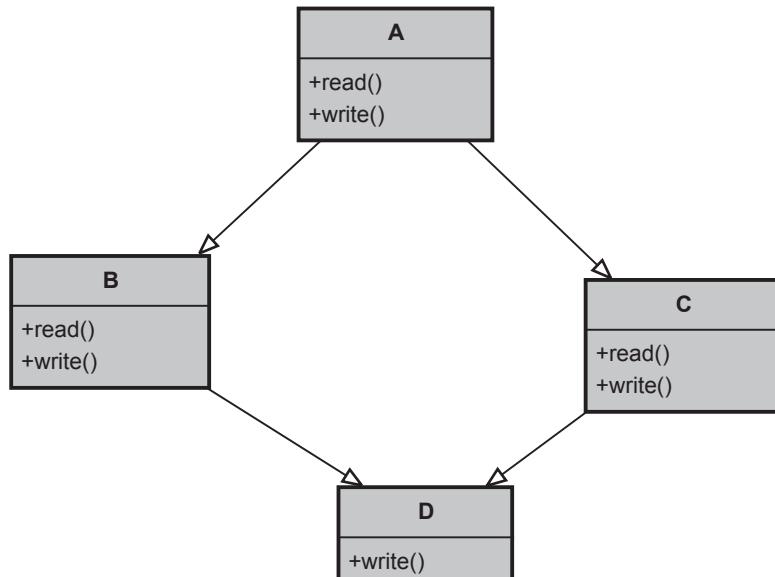


Fig. 2.10.1 Ambiguity in inheritance

- Now the code for the above design can be written as

```
class A
{
public:
    read();
    write();
};

class B:public A
{
public:
    read();
    write();
};

class C:public A
{
public:
    write();
};

class D:public B,public C
{
public:
    write();
};
```

- We try to inherit the **write()** function of base class A in class B and C, which will be alright. But if try to use the **write()** function in class D then the compiler will generate error, because it is ambiguous to know which **write()** function to choose whether of class B or class C. This ambiguity occurs because the compiler understands that the class D is derived from both class B and class C and both of these classes have the versions of **write()** function. So the A class gets duplicated inside the class D object.
- The compiler will complain when compiling the code: error: 'request for member "write" is ambiguous', because it can't figure out whether to call the method **write()** from **A::B::D** or from **A::C::D**.
- That means the programming language does not allow us to represent the concept as given in the design.
- C++ allows the solving of this problem by using **virtual inheritance**. This process is also called as **disinheritance**.
- In order to prevent the compiler from giving error due to multipath inheritance we use the keyword **virtual**. That means base class is made virtual.

Review Question

1. What is multiple inheritance ? What is ambiguity in multiple inheritance ? Give suitable example to demonstrate multiple inheritance.

SPPU : May-19, Marks 6

2.11 Virtual Base Class

In order to prevent the compiler from giving an error due to ambiguity in multiple path or multiple inheritance, we use the keyword virtual. That means when we inherit from the base class in both derived classes, the base class is made virtual. The code that illustrates the concept of virtual base class is as given below -

C++ Program

```
#include<iostream.h>
class base {
    public:
        int i;
};
class derived1:virtual public base
{
public:
    int j;
};
class derived2:virtual public base
{
public:
    int k;
};
//derived3 is inherited from derived1 and derived2
//but only one copy of base class is inherited.
class derived3:public derived1,public derived2
{
public:
    int sum()
    {
        return i+j+k;
    }
};
void main()
{
    derived3 obj;
    obj.i=10;
    obj.j=20;
    obj.k=30;
    cout<" The sum is = "<obj.sum();
}
```

Output

The sum is = 60

Program Explanation

In above program if we do not write the keyword virtual while deriving the classes derived1 and derived2 then compiler would have generated error messages stating the ambiguity in accessing the member of base class.

The sum is a function which can access the variables i, j and k of parent classes. In main function we have created an object obj of derived3 class and using this object i, j and k can be accessed.

Example 2.11.1 Develop an object oriented program in C++ to create a payroll system for an organization where one base class consist of employee name, code, designation and another base class consist of a account no and date of joining. The derived class consists of the data members such as basic pay, DA, HRA, CCA and deductions PF, LIC, IT. (Program must use the concept of virtual base class)

Solution :

```
*****
Program to Payroll System for organization
*****
#include<iostream.h>
class Employee
{
public:
    char name[10];
    int code;
    char designation[15];

};
class Accounts:virtual public Employee
{
    int accno;
    char doj[10];
};

class Pay:virtual public Employee
{
public:
    double BasicPay;
    double DA;
    double HRA;
    double CCA;
    double PF,LIC,IT;
};

class Derived:public Accounts,public Pay
{
```

```

public:
    void get_details()
    {
        cout<<"\n Enter the name of Employee: ";
        cin>>name;
        cout<<"\n Enter the Employee Code: ";
        cin>>code;
        cout<<"\n Enter the designation of Employee: ";
        cin>>designation;
        cout<<"\n Enter the Basic Payment: ";
        cin>>BasicPay;
        cout<<"\n Enter amount of DA : ";
        cin>>DA;
        cout<<"\n Enter the amount of HRA: ";
        cin>>HRA;
        cout<<"\n Enter the amount of CCA: ";
        cin>>CCA;
        cout<<"\n Enter the amount of PF: ";
        cin>>PF;
        cout<<"\n Enter the amount of LIC: ";
        cin>>LIC;
        cout<<"\n Enter the amount of IT: ";
        cin>>IT;
    }
    double NetPayment()
    {
        double amount,deductions;
        deductions=PF+LIC+IT;
        amount=(BasicPay+DA+HRA+CCA)-deductions;
        return amount;
    }
};

void main()
{
    Derived obj;
    obj.get_details();
    cout<<obj.NetPayment();
}

```

Output

Enter the name of Employee: ABC
 Enter the Employee Code: 100
 Enter the designation of Employee: Manager
 Enter the Basic Payment: 12000
 Enter amount of DA : 3000
 Enter the amount of HRA: 2000
 Enter the amount of CCA: 700
 Enter the amount of PF: 1200

```
Enter the amount of LIC: 1000
Enter the amount of IT: 3000
12500
```

2.12 Abstract Class

SPPU : May-19, Marks 2

- Abstract class is a class which is mostly used as a base class. It contains at least one **pure virtual function**. Abstract classes can be used to specify an interface that must be implemented by all subclasses.
- The virtual function is function having nobody but specified **by = 0**. This tells the compiler that nobody exists for this function relative to the base class. When a *virtual* function is made **pure**, it forces any derived class to override it. If a derived class does not, an error occurs. Thus, making a *virtual* function **pure** is a way to guarantee that a **derived class will provide its own redefinition**.

For example

```
#include <iostream>
using namespace std;

class area
{
    double dim1, dim2;
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0; // pure virtual function
};

class square : public area
{
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
    }
}
```

```
};

class triangle : public area
{
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    square s;
    triangle t;
    int num1,num2;
    cout<<"\n Enter The two dimensions for calculating area of
square";
    cin>>num1>>num2;
    s.setarea(num1,num2);
    p = &s;
    cout << "Area of square is : " << p->getarea() << '\n';
    cout<<"\n Enter The two dimensions for calculating area of
triangle";
    cin>>num1>>num2;
    t.setarea(num1,num2);
    p = &t;
    cout << "Area of Triangle is: " << p->getarea() << '\n';
    return 0;
}
```

Output

```
Enter The two dimensions for calculating area of square10 20
Area of square is : 200
```

```
Enter The two dimensions for calculating area of triangle6
8
Area of Triangle is: 24
```

In above code the *getarea()* is a function which is defined as pure virtual function in base class. But it is redefined in derived class *square* and *triangle*. In derived class *square* the *getarea()* function calculates the area of square and in derived class *triangle* the *getarea()* function calculates the area of triangle. The definition of *getarea* in base class is

overridden by the definitions of functions in derived class. The class area acts as an abstract class because -

- It specifies an interface which is used by all the derived classes. Thus it is never used directly it simply gives skeleton to other derived classes.
- It contains one pure virtual function `getarea()`.

Example 2.12.1 What are abstract classes ? Write a program having student as an abstract class and create many derived classes such as engineering, science, medical etc. from the student class. Create their object and process them.

Solution :

```
#include<iostream>
#include<cstring>
using namespace std;
class Student
{
    char name[10];
public:
    void SetName(char n[10])
    {
        strcpy(name,n);
    }
    void GetName(char n[10])
    {
        strcpy(n,name);
    }
    virtual void qualification()=0;
};

class Engg:public Student
{
public:
    void qualification()
    {
        char n[10];
        GetName(n);
        cout<<n<<" is a an engineering student"<<endl;
    }
};
class Medical:public Student
{
public:
    void qualification()
    {
        char n[10];
        GetName(n);
    }
};
```

```

        cout<<n<<" is a medical student"<<endl;
    }
};

int main()
{
    Student *s;
    Engg e;
    Medical m;
    char nm[10];
    cout<<"\n Enter the name: "<<endl;
    cin>>nm;
    e.SetName(nm);
    s=&e;
    s->qualification();
    cout<<"\n Enter the name: "<<endl;
    cin>>nm;
    m.SetName(nm);
    s=&m;
    s->qualification();
    return 0;
}

```

Output

Enter the name:
Ramesh
Ramesh is a an engineering student

Enter the name:
Suresh
Suresh is a medical student

Review Question

1. What is abstract class ? Give suitable example

SPPU : May-19, Marks 2

2.13 Friend Class

Similar to a friend function one can declare a class as a friend to another class. This allows the friend class to access the private data members of the another class. In the following program class B is declared as friend of class A. Therefore class B can access the variable *data*, and variable is private member of class A.

C++ Program

```
#include<iostream>
using namespace std;
class A
{
private:
    int data;
    friend class B;//class B is friend of class A
public:
    A()//constructor
    {
        data = 5;
    }
};
class B
{
public:
    int sub(int x)
    {
        A obj1; //object of class A
        //the private data of class A is accessed in class B
        // data contains 5 and x contains 2
        return obj1.data - x;
    }
};
int main()
{
    B obj2;
    cout << "Result is = "<< obj2.sub(2);
    getch();
    return 0;
}
```

Output

Result is = 3

For certain specific application one can declare either a friend function or a friend class. But if all the functions or classes are declared as friend then it will lose the purpose of data encapsulation and data hiding.

2.14 Nested Class

When one class is defined inside the other class then it is called the nested class. The nested class can access the data member of the outside class. Similarly the data member of the nested can be accessed from the main. Following is a simple C++ program that illustrates the use of nested class.

```
/*
The program for demonstration of nested class
*/
#include<iostream.h>
class outer
{
public:
    int a; // Note that this member is public
    class inner
    {
    public:
        void fun(outer *o,int val)
        {
            o->a = val;
            cout<<"a= "<<o->a;
        }
    }; //end of inner class
}; //end of outer class
void main()
{
    outer obj1;
    outer::inner obj2;
    obj2.fun(&obj1,10); //invoking the function of inner class
}
```

Output

a= 10

Part II : Pointers**2.15 Pointer - Indirection Operator**

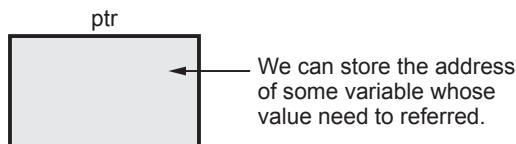
Definition : A pointer is a variable that represents the memory location of some other variable. The purpose of pointer is to hold the memory location and not the actual value.

Consider the variable declaration

```
int *ptr;
```

ptr is the name of our variable. The ***** informs the compiler that we want a pointer variable, the **int** says that we are using our pointer variable which will actually store the address of an integer. Such a pointer is said to be **integer pointer**.

Thus **ptr** is now ready to store an address of the value which is of integer type.

**Fig. 2.15.1 Pointer variable**

2.16 Declaring and Initializing Pointers

The dynamic memory allocation is done using an operator **new**. The syntax of dynamic memory allocation using **new** is

`new data type;`

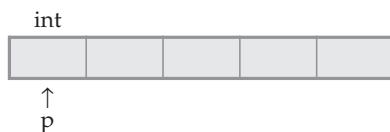
For example :

```
int *p;
p=new int;
```

We can allocate the memory for more than one element. For instance if we want to allocate memory of size in for 5 elements we can declare.

```
int *p;
p=new int[5];
```

In this case, the system dynamically assigns space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to *p*. Therefore, now, *p* points to a valid block of memory with space for five elements of type *int*.



The program given below allocates the memory for any number of elements and the memory for those many number of elements get deleted at the end of the program.

The memory can be deallocated using the **delete** operator. The syntax is

`delete variable_name;`

For example

```
delete p;
```

Let us discuss following C++ program which makes use of new and delete operators

C++ Program

```
#include <iostream>
using namespace std;
int main ()
{
    int i,n;
```

```

int *p;
cout << "How many numbers would you like to type? ";
cin >> i;
p = new int[i];//dynamic memory allocation
if (p == 0)
    cout << "Error: memory could not be allocated";
else
{
    for (n=0; n<i; n++)
    {
        cout << "Enter The Number: ";
        cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
        cout << " " <<p[n];
    delete[] p;//dynamic memory deallocation
}
return 0;
}

```

Output

How many numbers would you like to type? 7
 Enter The Number: 10
 Enter The Number: 20
 Enter The Number: 30
 Enter The Number: 40
 Enter The Number: 50
 Enter The Number: 60
 Enter The Number: 70
 You have entered: 10 20 30 40 50 60 70

The use of dynamic memory allocation avoids the wastage of memory. Because the programmer can allocate the memory as per his need using *new* operator and when that memory block is not needed it is deallocated.

2.16.1 Accessing Variable through Pointers

To understand how to access the variables through pointer let us understand the program given below -

```
#include<iostream>
using namespace std;
void main()
{
    int *ptr;
    int a, b;
```

```
a = 10; /*storing some value in a*/
ptr = &a; /*storing address of a in ptr*/
b = *ptr; /*getting value at ptr in b*/
cout << "\n a = " << a; /*printing value stored at a*/
cout << "\n ptr = " << ptr; /*printing address stored at ptr*/
cout << "\n ptr = " << *ptr; /*Level of indirection at ptr*/
cout << "\n b = " << b; /*printing the value stored at b*/
}
```

Output

```
a = 10
ptr = 0020FE64
ptr = 10
b = 10
```

Program Explanation : In above program, firstly we have stored 10 in variable **a**, then we have stored an address of variable **a** in variable **ptr**. For that we have declared **ptr** as pointer type. Then on the next line value at the address stored in **ptr** is 10 which is been transferred to variable **b**. Finally we have printed all these values by some **printf** statements.

The following program illustrates the concept of **&** and ***** in more detail-

```
#include<iostream>
using namespace std;
int main()
{
    int *ptr;
    int a, b;

    a = 10;
    b = 20;
    cout << "\n Originally a =" << a << " b = " << b;
    ptr = &a;
    b = *ptr;
    cout << "\n\n Now the changed values are\n\t a = " << a << " b = " << b;
    cout << "\n ptr = " << ptr;
    cout << "\n &ptr = " << &ptr;
    cout << "\n *ptr = " << *ptr;
    cout << "\n *(&ptr) = " << *(&ptr);
    cout << "\n Address of a is &a = " << &a;
    cout << "\n Address of b is &b = " << &b;
    return 0;
}
```

Output

```
Originally a = 10 b = 20
Now the changed values are
a = 10, b = 10
```

```

ptr = 65522
& ptr = 65524
* ptr = 10
* (&ptr) = 65522
Address of a is &a = 65522
Address of b is &b = 65520

```

In above program there are 2 variables and 1 pointer variable. The values stored in these variables are -

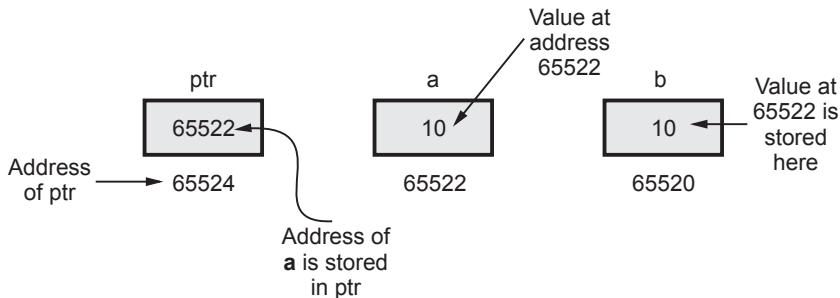


Fig. 2.16.1

Key Point Whenever we want to store address of some variable into another variable, then another variable should be of pointer type.

2.17 Memory Management : New and Delete

SPPU : Dec.-18, May-19, Marks 6

The dynamic memory allocation is done using an operator **new**. The syntax of dynamic memory allocation using **new** is

new data type;

For example :

```

int *p;
p=new int;

```

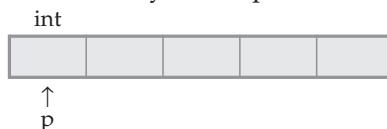
We can allocate the memory for more than one element. For instance if we want to allocate memory of size in for 5 elements we can declare.

```

int *p;
p=new int[5];

```

In this case, the system dynamically assigns space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to p. Therefore, now, p points to a valid block of memory with space for five elements of type *int*.



The program given below allocates the memory for any number of elements and the memory for those many number of elements get deleted at the end of the program.

The memory can be deallocated using the **delete** operator. The syntax is
delete variable_name;

For example

```
delete p;
```

Let us discuss following C++ program which makes use of new and delete operators

C++ Program

```
#include <iostream>
using namespace std;
int main ()
{
    int i,n;
    int *p;
    cout << "How many numbers would you like to type? ";
    cin >> i;
    p= new int[i];//dynamic memory allocation
    if (p == 0)
        cout << "Error: memory could not be allocated";
    else
    {
        for (n=0; n<i; n++)
        {
            cout << "Enter The Number: ";
            cin >> p[n];
        }
        cout << "You have entered: ";
        for (n=0; n<i; n++)
            cout << " " <<p[n];
        delete[] p;//dynamic memory deallocation
    }
    return 0;
}
```

Output

```
How many numbers would you like to type? 7
Enter The Number: 10
Enter The Number: 20
Enter The Number: 30
Enter The Number: 40
Enter The Number: 50
Enter The Number: 60
Enter The Number: 70
You have entered: 10 20 30 40 50 60 70
```

The use of dynamic memory allocation avoids the wastage of memory. Because the programmer can allocate the memory as per his need using *new* operator and when that memory block is not needed it is deallocated.

Review Questions

1. Compare and contrast memory allocation and deallocation using *new* *delete*.

SPPU : May-19, Marks 4

2. What is dynamic memory allocation ? Explain its use in C++ with suitable example.

SPPU : Dec.-18, Marks 6

2.18 Pointers to Object

Following is a simple C++ program in which a pointer to object variable is used to access the value of the class.

```
*****
Program to display the contents using the pointer to object.
*****/
```

```
#include <iostream>
using namespace std;
class Test
{
    int a;
public:
    Test(int b)
    {
        a = b;
    }
    int getVal()
    {
        return a;
    }
};
int main()
{
    Test obj(100), *ptr_obj;
    ptr_obj = &obj;

    cout <<"Value obtained using pointer to object is ..."<<endl;
    cout<<ptr_obj->getVal()<<endl;
    return 0;
}
```

Address of **obj** is stored in
pointer variable.

Output

100

TECHNICAL PUBLICATIONS® - An up thrust for knowledge

Program Explanation

In above program, the object to the class **Test** is **obj**. One pointer variable is created named **ptr_obj**. This is actually a pointer to the object. This pointer can access the public function of the class **Test**. Hence we are calling the function **getVal** of the class **Test** using the pointer to the object. Thus member function can be accessed using pointer.

2.19 this Pointers

SPPU : Dec.-16, 19, Marks 2

The keyword *this* identifies a special type of pointer. Suppose that you create an object named **x** of **class A** and **class A** has a **nonstatic member function f()**. If you call the function **x.f()**, the keyword **this** in the body of **f()** stores the address of **x**. You cannot declare this pointer or make assignments to it.

A static member function does not have a **this** pointer.

The **this** pointer is passed as a hidden parameter to the member function call and it is available in the function definition as a local variable. Below is an example in which this pointer is used to refer the **num** variable.

C++ Program

```
#include<iostream>
using namespace std;
class test
{
private:
    int num;
public:
    void get_val(int num)
    {
        //this pointer retrieves the value of obj.num
        //this pointer is hidden by automatic variable num
        this->num=num;
    }
    void print_val()
    {
        cout<<"\n The value is "<<num;
    }
};
int main()
{
    test obj;
    int num;
```

```

cout<<"\n Enter Some Value ";
cin>>num;
obj.get_val(num);
obj.print_val();
return 0;
}

```

Output

Enter Some Value 10
The value is 10

The **this** pointer points to the object for which the member function is called. Hence from above code *obj.get_val(num)* can be interpreted as *get_val(&obj,num)*; That means **this** pointer is passed as a hidden argument to the called function by some automatic variable like *num*.

Review Question

1. What is the use of this pointer ?

SPPU : Dec.-16, 19, Marks 2

2.20 Pointers Vs Arrays

Sr.No.	Array	Pointer
1	Array is a collection of similar data type elements.	Pointer is a variable that can store an address of another variable.
2	Arrays are static in nature that means once the size of array is declared we can not resize it.	Pointers are dynamic in nature, that means the memory allocation and deallocation can be done using new and delete operators .
3	Arrays are allocated at compile time	Pointers are allocated at run time.
4	Syntax: type var_name[size];	Syntax: type *var_name;

2.21 Accessing Arrays using Pointers

Pointers are meant for storing the address of the variable. The pointer can point to any cell of array. For example -

```

int *ptr;
int a[10];
ptr=&a[5]; //The address of 6th element of array a is stored in pointer variable.

```

It is possible to store the base address of array to pointer variable and entire array can be scanned using this pointer.

```
#include<iostream>
using namespace std;
int main()
{
    int a[3], *ptr;
    int i;
    ptr = &a[0];//storing base address of an array
    cout << "\n Address using array";
    for (i = 0; i < 3;i++)
        cout << "\n The a[" << i << "] is " << &a[i];
    cout << "\n Address using pointer";
    for (i = 0; i < 3; i++)
        cout << "\n The ptr#" << i << " is " <<ptr+i;
    return 0;
}
```

Output

```
Address using array
The a[0] is 003CF83C
The a[1] is 003CF840
The a[2] is 003CF844
Address using pointer
The ptr#0 is 003CF83C
The ptr#1 is 003CF840
The ptr#2 is 003CF844
```

Example 2.21.1 Write a C program using pointer for searching the desired element from the array.

Solution :

```
#include<iostream>
using namespace std;
void main()
{
    int a[10], i, n, *ptr, key;
    cout << "\n How Many elements are there in an array ? ";
    cin >> n;
    cout << "\n Enter the elements in an array ";
    for (i = 0; i < n; i++)
        cin >> a[i];
    ptr = &a[0];/*copying the base address in ptr */
    cout << "\n Enter the Key element ";
    cin >> key;
    for (i = 0; i < n; i++)
```

```

{
    if (*ptr == key)
    {
        cout<<"\n The element is present ";
        break;
    }
    else
        ptr++; /*pointing to next element in the array*/
        /*Or write ptr=ptr+i*/
}

```

Output

How Many elements are there in an array ? 5

Enter the elements in an array

10
20
30
40
50

Enter the Key element 40

The element is present

2.22 Pointer Arithmetic

SPPU : Dec.-18, Marks 4

Various operations can be performed using pointer variables as follows -

Let

```
int *ptr1,*ptr2;
int x;
```

Operation	Meaning
x=ptr1 * ptr2	Multiplication of two pointer variables is possible in this way.
x=ptr1-ptr2	The subtraction of two pointer variables.
ptr1-- or ptr1 ++	The pointer variable can be incremented or decremented .
x=ptr1+10	We can add some constant to pointer variable.
x=ptr1-20	We can subtract a constant value from the pointer.

ptr1<ptr2	
ptr1>ptr2	
ptr1==ptr2	
ptr1<=ptr2	The relational operations are possible on pointer variable while comparing two pointers.
ptr1>=ptr2	
ptr1!=ptr2	

But here is a list of some **invalid operations**. These operations are not allowed in any program.

Operation	Meaning
ptr1+ptr2	Addition of two pointers is not allowed
ptr1*ptr2	Multiplication of two pointers is not allowed
ptr1/ptr2	Division of two pointers is not allowed
ptr1*2	Multiplying by some constant to a pointer variable is not allowed
ptr2/2	Division by some constant to a pointer variable is not allowed
&ptr1=100	Address of variable can not be altered directly.

Now consider

```
int *ptr;
```

This is a pointer variable which is of integer type. This allocates the memory of 2 bytes for each such variable. If there are 10 integer pointers then total 20 bytes of memory block will be reserved. Consider a block in memory consisting of ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer **ptr** at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write:

```
ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer it adds 2 to **ptr** instead of 1, so the pointer "points to" the next integer, at memory location 102. The same goes for other data types such as floats, doubles, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic".

The `ptr++` or `++ptr` is equivalent to `ptr+1`

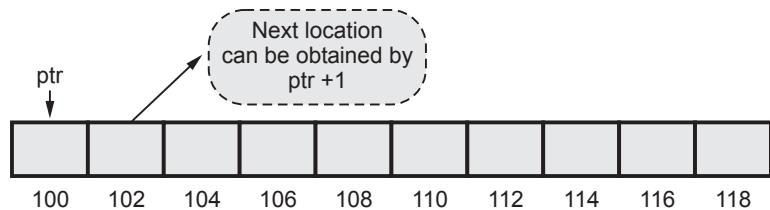


Fig. 2.22.1

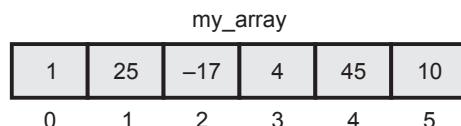
As a block of 10 integers in a contiguous fashion is similar to the concept of array we will now discuss an interesting relationship between arrays and pointers.

Consider the following :

```
int my_array[ ] = {1,23,17,4,-5,100};
```

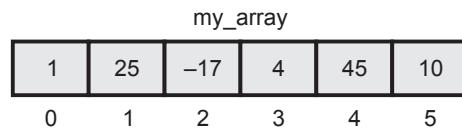
Here we have an array containing 6 integers. We refer to each of these integers with the help of subscript of `my_array`, i.e. using `my_array[0]` through `my_array[5]`. Alternatively, we can also access them via a pointer as follows :

```
int *ptr;
ptr = &my_array[0];/* point our pointer at the first integer in our array */
```



`my_array [0]` is 1 or it is `(my_array + 0)`
`my_array [1]` is 25 or it is `(my_array + 1)`
`my_array [2]` is -17 \Rightarrow `(my_array + 2)`
`my_array [3]` is 4 \Rightarrow `(my_array + 3)`
`my_array [4]` is 45 \Rightarrow `(my_array + 4)`
`my_array [5]` is 10 \Rightarrow `(my_array + 5)`

Fig. 2.22.2 Method 1 for accessing elements of an array



$\text{ptr} = \& \text{my_array}[0]$ i.e. location 0th in array
 then $*\text{ptr}$ will contain 1 i.e. my_array [0] value
 $*(\text{ptr} + 1) \Rightarrow a[1]$ i.e. 25
 $*(\text{ptr} + 2) \Rightarrow a[2]$ i.e. -17
 $*(\text{ptr} + 3) \Rightarrow a[3]$ i.e. 4
 $*(\text{ptr} + 4) \Rightarrow a[4]$ i.e. 45
 $*(\text{ptr} + 5) \Rightarrow a[5]$ i.e. 10

Fig. 2.22.3 Method 2 for accessing array elements (using pointers)

Review Question

1. Explain various arithmetic pointer operations

SPPU : Dec.-18, Marks 4

2.23 Arrays of Pointers

The array of pointers means the array locations are containing the address of another variable which is holding some value. For example,

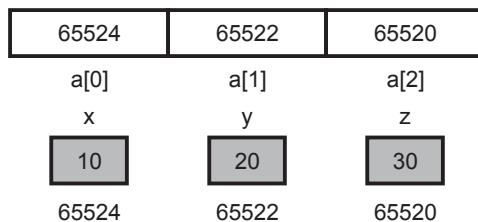


Fig. 2.23.1 Array of pointers

The array of pointers is the concept which is mainly used when we want to store the multiple strings in an array. Here we have simply taken the integer values in three different variables x, y and z. The addresses of x, y and z are stored in the array a. This concept is implemented by following simple C++ program.

```
#include<iostream>
using namespace std;
void main()
{
    int *a[10];/*array is declared as of pointer type*/
    int i, x, y, z;

    cout<<"\n Enter The Array Elements ";
    cin > x >> y >> z;
```

```

a[0] = &x; /*storing the address of each variable in array location */
a[1] = &y;
a[2] = &z;

for (i = 0; i<3; i++)
{
    cout<<"\nThe element "<<a[i]<<" is at location "<<a[i];
}
}

```

Output

Enter The Array Elements

10
20
30

The element 10 is at location 65524
The element 20 is at location 65522
The element 30 is at location 65520

2.24 Function Pointers

SPPU : Dec.-17, May-18, 19, Marks 5

- The pointer to the function means a pointer variable that stores the address of function.
- The function has an address in the memory same like variable. As address of function name is a memory location, we can have a pointer variable which will hold the address of function.
- The data type of pointer will be same as the return type of the function. For instance : if the return type of the function is **int** then the integer pointer variable should store the address of that function.

- **Syntax**

Return_Type *pointer_variable (data_type);

- **For example,**

float (*fptr)(float);

Here **fptr** is a pointer to the function which has float parameter and returns the value float. Note that the parenthesis around **fptr**; otherwise the meaning will be different.

For example,

float *fptr(float);

This means **fptr** is a function with a float parameter and returns a pointer to float.

float fun(float);
float (*fptr) (float);
fptr=&fun;

Thus

```
/*function returning pointer to float */
float *fptr (float a);
/*pointer to function returning float */
float (*fptr) (float a);
```

The following program illustrates the use of pointer to the function

```
#include<iostream>
using namespace std;
void main()
{
    void display(float(*)(int), int);
    float area(int);
    int r;
    cout<<"\n Enter the radius ";
    cin>>r;
    display(area, r);/*function is passed as a parameter to another function*/
}
void display(float(*fptr)(int), int r)
{
    /*call to pointer to function*/
    cout<<"\n The area of circle is "<(*fptr)(r);
}
float area(int r)
{
    return (3.14*r*r);
}
```

Output

Enter the radius 10

The area of circle is 314.000000

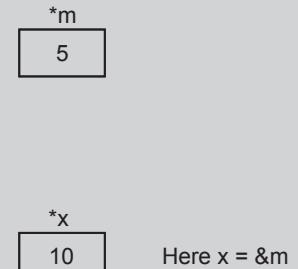
The function **display** calls the function **area** through pointer variable **fptr**. Thus **fptr** is actually a pointer to the function **area**. As function **area** returns the float value we have the pointer as of float type.

2.24.1 Passing Pointer to the Function

A pointer variable can be passed as an argument to the function. This method of parameter passing is called **call by reference**. Following C++ illustrates how to pass pointer as an argument to the function

```
*****
Passing a pointer variable to the function
*****
#include<iostream>
using namespace std;
void main()
```

```
{
int *m;
void fun(int *);
int n = 5;
m = &n;
cout<<"\n Following value is just before function call \n";
cout<<*m;
cout<<"\n Following value is obtained from the function \n";
fun(m); // Note how the pointer variable is passed
cout<<*m;
}
void fun(int *x)
{
    *x = 10;
}
```

**Output**

Following value is just before function call
5
Following value is obtained from the function
10

2.24.2 Returning Pointer from Function

C++ allows to return a pointer i.e. address of a local variable from the function. The function which returns a pointer is can be declared as follows

```
Data_type *Function_Name
{
    Function body
}

Following program illustrates how to return a pointer from function
*****
Demonstration of function Returning Pointer
*****
#include<iostream>
using namespace std;
int* sum(int*, int*);
int main()
{
    int a, b;
    int *c;
    cout<<"\n Enter the value of a and b ";
    cin>>a>>b;
    c = sum(&a, &b);
    cout<<"Sum of "<<a<<" and "<<b<<" is "<<*c;
    return 0;
}
```

```
int* sum(int *x, int *y)
{
    int z;
    z = *x + *y;
    return &z;
}
```

Output

Enter the value of a and b 10 20
Sum of 10 and 20 is 30

Example 2.24.1 Write a program to find the sum of an array Arr by passing an array to a function using pointer.

SPPU : Dec.-17, Marks 4

Solution :

```
#include <iostream>
using namespace std;
int fun(const int *arr, int size)
{
    int sum = *arr;
    for (int i = 1; i < size; ++i)
    {
        sum = sum + *(arr+i);
    }
    return sum;
}

int main()
{
    const int SIZE = 5;
    int numbers[SIZE] = {10, 20, 90, 76, 22};
    cout << "The sum of array is: " << fun(numbers, SIZE) << endl;
    return 0;
}
```

Example 2.24.2 Explain pointer to a variable and pointer to a function. Use suitable example.

SPPU : May-18, Marks 4

Solution :

- **Pointer to variable :** A pointer is a variable that represents the memory location of some other variable. The purpose of pointer is to hold the memory location and not the actual value.
- Consider the variable declaration

```
int *ptr;
```

ptr is the name of our variable. The * informs the compiler that we want a pointer variable, the int says that we are using our pointer variable which will actually store the address of an integer. Such a pointer is said to be **integer pointer**.

- **Pointer to function :** The pointer to the function means a pointer variable that stores the address of function.
- **Syntax**

```
Return_Type *pointer_variable (data_type);
```

For example

```
float (*fptr)(float);
```

Here **fptr** is a pointer to the function which has float parameter and returns the value float. Note that the parenthesis around fptr; otherwise the meaning will be different.

The following program illustrates the use of pointer to the function

```
#include<iostream>
using namespace std;
void main()
{
    void display(float(*)(int), int);
    float area(int);
    int r;
    cout<<"\n Enter the radius ";
    cin>>r;
    display(area, r);/*function is passed as a
                      parameter to another function*/
}
void display(float(*fptr)(int), int r)
{
    /*call to pointer to function*/
    cout<<"\n The area of circle is "<(*fptr)(r);
}
float area(int r)
{
    return(3.14*r*r);
}
```

Review Question

1. What is the concept of function pointers ? Give suitable example in C++.

SPPU : May-19, Marks 5

2.25 Pointers to Pointers

A pointer can point to other pointer variables which brings the multiple level of indirection. We can point to any number of pointer variables. But as the level of indirection increases the complexity of program gets increased.

```
*****
Demonstration of Pointer to pointer
*****  

#include<iostream>
using namespace std;
void main()
{
    int a;
    int *ptr1, **ptr2;
    a = 10;
    ptr1 = &a;
    ptr2 = &ptr1;
    cout<<"\n a = "<<a;
    cout<<"\n *ptr1 = "<<*ptr1; /*value at address in ptr1*/
    cout<<"\n ptr1 = "<<ptr1; /*storing address of a*/
    cout<<"\n *ptr2 = "<<*ptr2; /*storing address of ptr1*/
    cout<<"\n ptr2 = "<<ptr2; /*address of ptr2*/
}
```

Output

```
a= 10
*ptr1 = 10
ptr1 = 65524
*ptr2= 65524
ptr2 = 65522
```

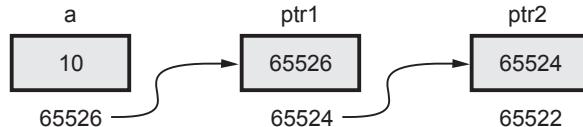


Fig. 2.25.1 Pointer to Pointer

In above program address of variable **a** is stored in a pointer variable **ptr1**. Similarly address of pointer variable **ptr1** is stored in variable **ptr2**. Thus **ptr2** is a pointer to pointer because it stores address of (65524) a variable which is already holding some address (65526). Hence we have declared **ptr2** as :

```
int **ptr2;
```

2.26 Pointers to Derived Classes

- It is possible to declare a pointer that points to the derived class.
- Using this pointer, one can access the data attributes as well as member functions of derived class.

Example Program

```
#include<iostream>
using namespace std;
class Base
{
public:
    int a;
};
class Derived :public Base
{
public:
    int b;
    void display()
    {
        cout << "\n a= " << a << "\n b= " << b;
    }
};
int main()
{
    Derived obj;
    Derived *ptr; //Pointer to derived class
    ptr = &obj;
    ptr->a = 100;
    ptr->b = 200;
    ptr->display();
}
```

Output

```
a= 100
b= 200
```

Program Explanation :

In above program

- We have declared base class Base and derived class Derived.
- The Base class declares simply a variable a and derived class defined one data member and one member function display.
- Inside the main, the pointer to the Derived class is declared which is ptr.
- Using pointer variable the variable of base class, variable of derived class and functionality of derived class can be accessed.

2.27 Null Pointer

- Variables are not initialized automatically by a valid address. If they are not initialized explicitly, then they might be assigned with garbage value.
- Pointer that are not initialized with valid address may cause some substantial damage. For this reason it is important to initialize them. The standard initialization is to the constant NULL.
- Using the NULL value as a pointer will cause an error on almost all systems.
- Literal meaning of NULL pointer is a pointer which is pointing to nothing.
- NULL pointer points the base address of segment.

Following is a simple program that illustrates the problem of Null pointer

```
*****
Program for illustrating null pointer problem
*****
#include <iostream.h>
#include <string.h>
void main()
{
    char *str=NULL;
    strcpy(str,"HelloFriends");
    cout<str;
}
```

In above program we are trying to copy some string in the Null pointer. One can not copy something to Null pointer. Due to which the Null Pointer problem occurs.

2.28 void Pointer

- The void pointer is a special type of pointer that can be used to point to the objects of any data type. It is also known as generic pointer.
- The void pointer is declared like normal pointer declaration, using the keyword void.
- For example :

```
void *ptr;
```

Following is a simple C++ program in which we have used void pointer. The address of integer variable is assigned to void pointer. And finally using this void pointer the integer data can be accessed.

```
#include<iostream.h>
void main()
{
//As the void pointer does not know
//what type of object it is pointing to,
```

```
//it can not be dereferenced!
//Hence, the void pointer must first be explicitly cast to another pointer type
//before it is dereferenced.

int int_val = 10;
void *pVoid = &int_val;
// can not dereference pVoid because it is a void pointer
int *pInt = static_cast<int*>(pVoid); // cast from void* to int*
cout << *pInt << endl; // can dereference pInt
}
```

Output

10



Object Oriented Programming - Laboratory

Group A

- Experiment 1** Implement a class complex which represents the complex number data type.
Implement the following
1. Constructor (including a default constructor which creates the complex number $0+0i$).
2. Overloaded operator+ to add two complex numbers.
3. Overloaded operator* to multiply two complex numbers.
4. Overloaded << and >> to print and read complex numbers. L - 2
- Experiment 2** Write a C++ program create a calculator for an arithmetic operator (+, -, *, /). The program should take two operands from user and performs the operation on those two operands depending upon the operator entered by user. Use a switch statement to select the operation. Finally, display the result..... L - 3
- Experiment 3** Develop an object oriented program in C++ to create a database of student information system containing the following information : Name, Roll number, Class, Division, Date of birth, Blood group, Contact address, Telephone number, Driving license no. and other. Construct the database with suitable member functions for initializing and destroying the data viz constructor, default constructor, Copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators-new and delete. L - 5
- Experiment 4** Imagine a publishing company which does marketing for book and audio cassette versions. Create a class publication that stores the title (a string) and price (type float) of a publication.
From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float).
Write a program that instantiates the book and tape classes, allows user to enter data and displays the data members. If an exception is caught, replace all the data member values with zero values..... L - 8
- Experiment 5** A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message Required copies not in stock is displayed.
Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required. Implement C++ program for the system. L - 11
- Experiment 6** Create employee bio-data using following classes i) Personal record ii))Professional record iii)Academic record Assume appropriate data members and member function to accept required data & print bio-data. Create bio-data using multiple inheritance using C++. L - 14

Group A

Experiment 1 Implement a class complex which represents the complex number data type.

Implement the following

1. Constructor (including a default constructor which creates the complex number 0+0i).
2. Overloaded operator+ to add two complex numbers.
3. Overloaded operator* to multiply two complex numbers.
4. Overloaded << and >> to print and read complex numbers.

C++ Program

```
#include<iostream>
using namespace std;
class complex
{
public:
    float real, img;
    complex() {}
    complex operator+ (complex);
    complex operator* (complex);
    friend ostream &operator<<(ostream &,complex&);
    friend istream &operator>>(istream &,complex&);
};
complex complex::operator+ (complex obj)
{
    complex temp;
    temp.real = real + obj.real;
    temp.img = img + obj.img;
    return (temp);
}
istream &operator >(istream &is, complex &obj)
{
    is >>obj.real;
    is >> obj.img;
    return is;
}
ostream &operator<(ostream &outt, complex &obj)
{
    outt<< " "<obj.real;
    outt << "+"<obj.img<<"i";
    return outt;
}
complex complex::operator* (complex obj)
{
```

```

complex temp;
temp.real = real*obj.real - img*obj.img;
temp.img = img*obj.real + real*obj.img;
return (temp);
}
int main()
{
    complex a,b,c,d;
    cout << "\n The first Complex number is: ";
    cout << "\nEnter real and img: ";
    cin >> a;
    cout << "\n The second Complex number is: ";
    cout << "\nEnter real and img: ";
    cin >> b;
    cout << "\n\n\t\t Arithmetic operations ";
    c = a + b;
    cout << "\n Addition = ";
    cout << c;
    d = a*b;
    cout << "\n Multiplication = ";
    cout << d;
    cout << endl;
    return 0;
}

```

Output

The first Complex number is :

Enter real and img : 2 6

The second Complex number is :

Enter real and img : 4 1

Arithmetic operations

Addition = 6 + 7i

Multiplication = 2 + 26i

Experiment 2 Write a C++ program create a calculator for an arithmetic operator (+, -, *, /).

The program should take two operands from user and performs the operation on those two operands depending upon the operator entered by user. Use a switch statement to select the operation. Finally, display the result.

C++ Program

```

#include <iostream>
using namespace std;
class Calculator
{
private:

```

```
float num1, num2;
char oper;
public:
    void input_data();
    friend void compute(float,float, char);
};

void Calculator::input_data()
{
    char ans='y';
    do
    {
        cout << "\n Enter first number,operator,second number: ";
        cin >> num1;
        cin >> oper;
        cin >> num2;
        compute(num1,num2,oper);
        cout << "\n Do another(y/n)? ";
        cin >>ans;
    } while (ans == 'y');
}

void compute(float num1,float num2,char op)
{
    float result;
    switch (op)
    {
        case '+':result = num1 + num2;
                    cout << "\n Answer = " << result;
                    break;
        case '-':result = num1 - num2;
                    cout << "\n Answer = " << result;
                    break;
        case '*':result = num1 * num2;
                    cout << "\n Answer = " << result;
                    break;
        case '/':if (num2 != 0)
        {
            result = num1 / num2;
            cout << "\n Answer = " << result;
        }
        else
            cout << "\n Division is not possible!!";
        break;
    }
}

int main()
{
    Calculator obj;
```

```
    obj.input_data();
    return 0;
}
```

Output

Enter first number, operator,second number : 10 / 3

Answer = 3.33333

Do another(y / n) ? y

Enter first number, operator,second number : 12 + 100

Answer = 112

Do another(y / n) ? n

Experiment 3 Develop an object oriented program in C++ to create a database of student information system containing the following information : Name, Roll number, Class, Division, date of birth, Blood group, Contact address, Telephone number, Driving license no. and other. Construct the database with suitable member functions for initializing and destroying the data viz constructor, default constructor, Copy constructor, destructor, static member functions, friend class, this pointer, inline code and dynamic memory allocation operators-new and delete.

C++ Program

```
#include<iostream>
#include<string>
#include<cstring>
using namespace std;

class PersonClass
{
private:
    char name[40], clas[10], div[2], dob[15], bloodgrp[5];
    int roll;
public:
    static int count;//static data
    friend class PersonnelClass;
    PersonClass()
    {
        char *name = new char[40];
        char *dob = new char[15];
        char *bloodgrp = new char[5];
        char *cls = new char[10];
        char *div = new char[2];
        roll = 0;
```

```
    }
    static void TotalRecordCount()//static method
    {
        cout < "\n\nTOTAL NUMBER OF RECORDS CREATED: " < count;
    }
};

class PersonnelClass
{
private:
    char address[30], telephone_no[15], policy_no[10], license_no[10];
public:
    PersonnelClass()//constructor
    {
        strcpy(address, "");
        strcpy(telephone_no, "");
        strcpy(policy_no, "");
        strcpy(license_no, "");
    }
    void InputData(PersonClass *obj);
    void DisplayData(PersonClass *obj);
    friend class PersonClass;

};

int PersonClass::count = 0;//static data initialized using scope resolution
// operator

void PersonnelClass::InputData(PersonClass *obj)
{
    cout << "\nROLLNO: ";
    cin >> obj->roll;
    cout << "\nNAME: ";
    cin >> obj->name;
    cout << "\nCLASS: ";
    cin >> obj->clas;
    cout << "\nDIVISION: ";
    cin >> obj->div;
    cout << "\nDATE OF BIRTH(DD-MM-YYYY): ";
    cin >> obj->dob;
    cout << "\nBLOOD GROUP: ";
    cin >> obj->bloodgrp;
    cout << "\nADDRESS: ";
    cin >> this->address;
    cout << "\nTELEPHONE NUMBER: ";
    cin >> this->telephone_no;
    cout << "\nDRIVING LICENSE NUMBER: ";
    cin >> this->license_no;
```

```
cout << "\nPOLICY NUMBER: ";
cin >> this->policy_no;

    obj->count++;
}

void PersonnelClass::DisplayData(PersonClass *obj)
{
    cout << "\n";
    cout << obj->roll << "    "
        << obj->name << "    "
        << obj->clas << "    "
        << obj->div << "    "
        << obj->dob << "    " << this->address << "    " << this->telephone_no \
        << "    " << obj->bloodgrp << "    "
        << this->license_no << "    " << this->policy_no;
}

int main()
{
    PersonnelClass *a[10];
    PersonClass *c[10];
    int n = 0, i, choice;
    char ans;
    do
    {
        cout << "\n\nMENU: ";
        cout << "\n\t1.Input Data\n\t2.Display Data";
        cout <<< "\n\nEnter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\n\n\tENTER THE DETAILS";
                cout << "\n    _____";
                do
                {
                    a[n] = new PersonnelClass;
                    c[n] = new PersonClass;
                    a[n]->InputData(c[n]);
                    n++;
                    PersonnelClass::TotalRecordCount();
                    cout << "\n\nDo you want to add more
records?(y/n): ";
                    cin >> ans;
                } while (ans == 'y' || ans == 'Y');
                break;
```

```

case 2:
    cout << "\n-----";
    cout << "\n Roll Name Class Div BirthDate Address Telephone
          Blood_Gr Licence Policy ";
    cout << "\n-----";
    for (i = 0; i<n; i++)
        a[i]->DisplayData(c[i]);
    PersonClass::TotalRecordCount();
    break;
}
cout << "\n\nDo you want to go to main menu?(y/n): ";
cin >> ans;
cin.ignore(1, '\n');
} while (ans == 'y' || ans == 'Y');
return 0;
}

```

Output

TOTAL NUMBER OF RECORDS CREATED : 1

Do you want to add more records ? (y / n) : n

Do you want to go to main menu ? (y / n) : y

MENU :

1.Input Data

2.Display Data

Enter your choice : 2

Roll Name Class Div BirthDate Address Telephone Blood_Gr Licence Policy

10 AAA Tenth A 12 - 12 - 2001 Pune 11111 A + ve 22222 33333

TOTAL NUMBER OF RECORDS CREATED : 1

Do you want to go to main menu ? (y / n) : n

Experiment 4 Imagine a publishing company which does marketing for book and audio cassette versions. Create a class publication that stores the title (a string) and price (type float) of a publication.

From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float).

Write a program that instantiates the book and tape classes, allows user to enter data and displays the data members. If an exception is caught, replace all the data member values with zero values.

C++ Program

```
#include <iostream>
#include <string>
#include <conio.h>
```

```
using namespace std;
class publication
{
    private:
        string title;
        float price;
    public:
        void getdata(void)
        {
            string t;
            float p;
            cout << "Enter title of publication: ";
            cin >> t;
            cout << "Enter price of publication: ";
            cin >> p;
            title = t;
            price = p;
        }
        void putdata(void)
        {
            cout << "Publication title: " << title << endl;
            cout << "Publication price: " << price << endl;
        }
};
class book :public publication
{
    private:
        int pagecount;
    public:
        void getdata(void)
        {
            publication::getdata(); //call publication class function to get data
            cout << "Enter Book Page Count: "; //Acquire book data from user
            cin >> pagecount;
        }
        void putdata(void)
        {
            publication::putdata(); //Show Publication data
            cout << "Book page count: " << pagecount << endl; //Show book data
        }
};
class tape :public publication
{
    private:
        float ptime;
    public:
        void getdata(void)
```

```
{  
    publication::getdata();  
    cout << "Enter tape's playing time(in min): ";  
    cin >> ptime;  
}  
void putdata(void)  
{  
    publication::putdata();  
    cout << "Tape's playing time: " << ptime << endl;  
}  
};  
int main(void)  
{  
    book b;  
    tape t;  
    b.getdata();  
    t.getdata();  
    b.putdata();  
    t.putdata();  
    return 0;  
}
```

Output

```
Enter title of publication: HarryPotter  
Enter price of publication: 200  
Enter Book Page Count: 150  
Enter title of publication: LoveSongs  
Enter price of publication: 100  
Enter tape's playing time(in min): 90  
Publication title: HarryPotter  
Publication price: 200  
Book page count: 150  
Publication title: LoveSongs  
Publication price: 100  
Tape's playing time: 90
```

Experiment 5 A book shop maintains the inventory of books that are being sold at the shop.

The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message Required copies not in stock is displayed. Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required. Implement C++ program for the system.

C++ Program

```
#include<iostream>
#include<string>
using namespace std;
class book
{
    char author[50];
    char title[50];
    char pub[50];
    double price;
    int numcopies;
public:
    book();
    int SearchBook(char t[],char a[]);
    void InputData();
    void DisplayRecords();
    void RequestCopies(int);
};
book::book()
{
    char *author=new char[50];
    char *title=new char[50];
    char *pub=new char[50];
    price=0;
    numcopies=0;
}
void book::DisplayRecords()
{
    cout<<"\n"<<title<<"\t"<<author<<"\t"<<pub
        <<"\t"<<price<<"\t"<<numcopies;
}
void book::InputData()
```

```
{  
    cout<<"\nTitle: ";  
    cin.getline(title,50);  
    std::cin.clear();  
    cout<<"\nAuthor: ";  
    cin.getline(author,50);  
    std::cin.clear();  
    cout<<"\nPublisher: ";  
    cin.getline(pub,50);  
    std::cin.clear();  
    cout<<"\nPrices: ";  
    cin>>price;  
    cout<<"\ncopies available: ";  
    cin>>numcopies;  
}  
  
int book::SearchBook(char t[],char a[])  
{  
    if(strcmp(title,t)&&(strcmp(author,a)))  
        return 0;  
    else return 1;  
}  
void book::RequestCopies(int num)  
{  
    if(numcopies>=num)  
    {  
        cout<<"\n Title is available";  
        cout<<"\nCost of "<<num<<" books is Rs. "<<(price*num);  
    }  
    else  
        cout<<"\nRequired copies not in Stock!!!";  
}  
  
void main()  
{  
    book obj[10];  
    char ans;  
    char key_title[50],key_author[50];  
    int n,i,copies,flag=0;  
    i=0;  
    cout<<"Enter details of books";  
    do  
    {  
        obj[i].InputData();  
        cout<<"Press y for entering more records";  
        cin>>ans;  
        std::cin.ignore(1,'\n');
```

```
i++;
}while(ans=='y');
n=i;
cout<<"\nTitle\tAuthor\tPublisher\tPrice\tCopies";
for(i=0;i<n;i++)
{
    obj[i].DisplayRecords();
}
cout<<endl;
cout<<"\n Enter title of required book\n";
cin.getline(key_title,50);
cout<<"\n Enter author of required book\n";
cin.getline(key_author,50);

for(i=0;i<n;i++)
{
    if(obj[i].SearchBook(key_title,key_author))
    {
        flag=1;
        break;
    }
}
if(flag==1)
{
    cout<<"\nPlease, Enter the number of copies of the book: ";
    cin>>copies;
    obj[i].RequestCopies(copies);
}
else
    cout<<"\n Book is not available";
}
```

Output

Enter details of books
Title: Digital Communication

Author:J.S.Chitode

Publisher:Technical

Prices:200

copies available:10
Press y for entering more records y

Title: Data Structures

Author:A.A.Puntambekar

Publisher:Technical

Prices:250

copies available:5

Press y for entering more records y

Title: Operating System

Author:I.A.Dhotre

Publisher:Technical

Prices:190

copies available:7

Press y for entering more records n

Title	Author	Publisher	Price	Copies
Digital Communication	J.S.Chitode	Technical	200	10
Data Structures	A.A.Puntambekar	Technical	250	5
Operating System	I.A.Dhotre	Technical	190	7

Enter title of required book

Operating System

Enter author of required book

I.A.Dhotre

Please, Enter the number of copies of the book: 5

Title is available

Cost of 5 books is Rs. 950

Experiment 6 Create employee bio-data using following classes i) Personal record ii))Professional record iii)Academic record Assume appropriate data members and member function to accept required data & print bio-data. Create bio-data using multiple inheritance using C++.

C++ Program

```
#include<iostream>
using namespace std;
class PersonalRecord
```

```
{  
protected:  
    char name[50];  
    char address[80];  
    char email[30];  
  
};  
class ProfessionalRecord  
{  
protected:  
    char qualification[50];  
    float experience_in_years;  
};  
class AcademicRecord:public PersonalRecord, public ProfessionalRecord  
{  
protected:  
    int ExamNo;  
    float marks;  
public:  
    void get_data()  
    {  
        cout<<"\n Enter name: ";  
        cin>>name;  
        cout<<"\n Enter address: ";  
        cin>>address;  
        cout<<"\n Enter Email address: ";  
        cin>>email;  
        cout<<"\n Enter Qualification: ";  
        cin>>qualification;  
        cout<<"\n Enter experience_in_years: ";  
        cin>>experience_in_years;  
        cout<<"\n Enter Exam Number: ";  
        cin>>ExamNo;  
        cout<<"\n Enter marks in percentage: ";  
        cin>>marks;  
    }  
    void put_data()  
    {  
        cout<<"\n ExamNo: "<<ExamNo;  
        cout<<"\n Name: "<<name;  
        cout<<"\n Percentage: "<<marks;  
        cout<<"\n Address: "<<address;  
        cout<<"\n Email: "<<email;  
        cout<<"\n Qualification: "<<qualification;  
        cout<<"\n Experience: "<<experience_in_years<<" years";  
    }  
}
```

```
};  
int main()  
{  
    AcademicRecord person;  
    person.get_data();  
    person.put_data();  
}
```

Output

```
Enter name: AAA  
Enter address: Pune  
Enter Email address: aaa.bbb@gmail.com  
Enter Qualification: BEComputer  
Enter experience_in_years: 10  
Enter Exam Number: 101  
Enter marks in percentage: 95  
ExamNo: 101  
Name: AAA  
Percentage: 95  
Address: Pune  
Email: aaa.bbb@gmail.com  
Qualification: BEComputer  
Experience: 10 years
```



SUBJECT CODE : 210243

As per Revised Syllabus of
SAVITRIBAI PHULE PUNE UNIVERSITY
Choice Based Credit System (CBCS)
S.E. (Computer) Semester - I

OBJECT ORIENTED PROGRAMMING (OOP)

(For END SEM Exam - 70 Marks)

Anuradha A. Puntambekar

M.E. (Computer)

Formerly Assistant Professor in
P.E.S. Modern College of Engineering,
Pune

Dr. Gayatri M. Bhandari

Ph. D in Computer Engineering,

M. Tech (CE), BE (CE)

Professor (Computer Dept.)

JSPM's, Bhivarabai Sawant Institute of Technology & Research
Wagholi, Pune



OBJECT ORIENTED PROGRAMMING (OOP)

(For END SEM Exam - 70 Marks)

Subject Code : 210243

S.E. (Computer) Semester - I

© Copyright with A. A. Puntambekar

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :



Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :

Yogiraj Printers & Binders
Sr.No. 10/1A,
Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-93-332-2156-6

A standard 1D barcode representing the ISBN number 978-93-332-2156-6.

978933221566 [1]

(ii)

SPPU 19

PREFACE

The importance of **Object Oriented Programming** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Object Oriented Programming**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors

A. A. Puntambekar

Dr. Gayatri M. Bhandari

Dedicated to God.

SYLLABUS

Object Oriented Programming (OOP) - (210243)

Credit	Examination Scheme
03	End_Sem (Theory) : 70 Marks

UNIT III Polymorphism

Polymorphism - Introduction to Polymorphism, Types of Polymorphism, Operator Overloading - concept of overloading, operator overloading, Overloading Unary Operators, Overloading Binary Operators, Data Conversion, Type casting (implicit and explicit), Pitfalls of Operator Overloading and Conversion, Keywords explicit and mutable. Function overloading.
Run Time Polymorphism - Pointers to Base class, virtual function and its significance in C++, pure virtual function and virtual table, virtual destructor, abstract base class. (**Chapter - 3**)

UNIT IV Files and Streams

Data hierarchy, Stream and files, Stream Classes, Stream Errors, Disk File I/O with Streams, File Pointers and Error Handling in File I/O, File I/O with Member Functions, Overloading the Extraction and Insertion Operators, memory as a Stream Object, Command-Line Arguments, Printer output. (**Chapter - 4**)

UNIT V Exception Handling and Templates

Exception Handling - Fundamentals, other error handling techniques, simple exception handling - Divide by Zero, Multiple catching, re-throwing an exception, exception specifications, user defined exceptions, processing unexpected exceptions, constructor, destructor and exception handling, exception and inheritance. **Templates** - The power of Templates, Function Template, Overloading Function templates and class template, class template and Nontype parameters, template and friends Generic Functions, The type name and export keywords. (**Chapter - 5**)

UNIT VI Standard Template Library (STL)

Introduction to STL, STL Components, Containers - Sequence container and associative containers, container adapters, Application of Container classes : vector, list,

Algorithms - basic searching and sorting algorithms, min-max algorithm, set operations, heap sort,

Iterators - input, output, forward, bidirectional and random access. Object Oriented Programming - a road map to future. (**Chapter - 6**)

TABLE OF CONTENTS

Unit - III

Chapter - 3 Polymorphism	(3 - 1) to (3 - 50)
3.1 Introduction to Polymorphism.....	3 - 2
3.1.1 Difference between Inheritance and Polymorphism	3 - 2
3.2 Early and Late Binding.....	3 - 2
3.2.1 Early Binding.....	3 - 2
3.2.2 Late Binding	3 - 3
3.3 Types of Polymorphism.....	3 - 5
3.4 Concept of Overloading	3 - 6
3.5 Operator Overloading	3 - 6
3.5.1 Rules for Operator Overloading	3 - 7
3.5.2 Overloading Unary Operators	3 - 8
3.5.3 Overloading Binary Operators	3 - 12
3.5.4 More Programs on Operator Overloading.....	3 - 14
3.6 Type Casting (Implicit and Explicit)	3 - 21
3.6.1 Basic Type to Class Type	3 - 22
3.6.2 Class Type to Basic Type	3 - 23
3.6.3 Class Type to another Class Type	3 - 23
3.7 Pitfalls of Operator Overloading and Conversion	3 - 25
3.8 Explicit and Mutable Keywords.....	3 - 26
3.8.1 The Explicit Keyword	3 - 26
3.8.2 The Mutable Keyword	3 - 27
3.9 Function Overloading	3 - 28
3.9.1 Rules for Function Overloading	3 - 28
3.10 Run Time Polymorphism	3 - 33

3.11 Pointers to Base Class	3 - 36
3.12 Virtual Function and its Significance in C++	3 - 37
3.13 Pure Virtual Function and Virtual Table	3 - 43
3.14 Virtual Destructor	3 - 44
3.15 Abstract Base Class	3 - 46

Unit - IV

Chapter - 4 Files and Streams	(4 - 1) to (4 - 38)
---	----------------------------

4.1 Stream and Files.....	4 - 2
4.2 Libraries.....	4 - 2
4.3 Stream Classes	4 - 3
4.4 Stream Errors	4 - 5
4.5 Disk File I/O with Streams	4 - 7
4.5.1 Open File Operation.....	4 - 8
4.5.2 Close File Operation.....	4 - 9
4.5.3 Handling Multiple Files	4 - 12
4.5.4 Handling Binary Files	4 - 16
4.5.5 Finding the End of the File.....	4 - 17
4.5.6 Reading and Writing Class Objects for Files.....	4 - 17
4.6 Unformatted I/O Functions.....	4 - 18
4.7 Formatted I/O Functions and I/O Manipulators	4 - 20
4.7.1 I/O Manipulators	4 - 22
4.8 File Pointers.....	4 - 24
4.9 Error Handling in File I/O.....	4 - 27
4.10 File I/O with Member Functions	4 - 28
4.11 Overloading the Extraction and Insertion Operators.....	4 - 31
4.12 Memory as a Stream Object	4 - 34
4.13 Command-Line Arguments	4 - 35
4.14 Printer Output.....	4 - 37

Chapter - 5 Exception Handling and Templates (5 - 1) to (5 - 36)

Part I : Exception Handling

5.1 Error Handling Techniques	5 - 2
5.2 Simple Exception Handling.....	5 - 2
5.2.1 try-catch-throw	5 - 3
5.3 Divide by Zero	5 - 5
5.4 Multiple Catching	5 - 6
5.5 Re-throwing an Exception.....	5 - 8
5.6 Exception Specifications.....	5 - 9
5.7 User Defined Exceptions	5 - 10
5.8 Processing Unexpected Exceptions.....	5 - 12
5.9 Constructor, Destructor and Exception Handling	5 - 13
5.10 Exception and Inheritance	5 - 14

Part II : Templates

5.11 Introduction to Templates	5 - 16
5.12 The Power of Templates	5 - 18
5.13 Function Template	5 - 18
5.14 Overloading Function Templates	5 - 19
5.14.1 Function Overloading vs Function Template	5 - 20
5.15 Class Template	5 - 20
5.16 Template Arguments	5 - 22
5.17 More Programs in Templates.....	5 - 23
5.18 Class Template and Nontype Parameters	5 - 28
5.19 Template and Friends	5 - 29
5.20 Generic Functions	5 - 30

5.20.1 Restrictions on Generic Functions	5 - 31
5.20.2 Applying Generic Functions	5 - 32
5.21 The Typename and Export Keywords	5 - 33

Unit - VI

Chapter - 6 Standard Template Library (STL)	(6 - 1) to (6 - 56)
---	----------------------------

6.1 Introduction to STL.....	6 - 2
6.2 STL Components.....	6 - 2
6.3 Sequence Container	6 - 3
6.3.1 Vectors	6 - 3
6.3.2 Deque	6 - 5
6.3.3 List	6 - 9
6.4 Associative Container.....	6 - 12
6.4.1 Set	6 - 12
6.4.2 Multi Set	6 - 15
6.4.3 Map	6 - 19
6.4.4 Multimap	6 - 23
6.5 Container Adapter.....	6 - 26
6.5.1 Stack	6 - 27
6.5.2 Queue	6 - 30
6.5.3 Priority Queues	6 - 34
6.6 Algorithms.....	6 - 37
6.6.1 Searching Algorithm	6 - 39
6.6.2 Sorting Algorithm.....	6 - 42
6.6.3 Min Max Algorithm	6 - 43
6.7 Set Operations	6 - 45
6.8 Heap Sort	6 - 49
6.9 Iterators- Input, Output, Forward, Bidirectional and Random Access.....	6 - 52
6.10 Object Oriented Programming - A Road Map to Future	6 - 55

Object Oriented Programming - Laboratory	(L - 1) to (L - 14)
---	----------------------------

Solved Model Question Paper	(M - 1) to (M - 2)
------------------------------------	---------------------------

Unit - III

3

Polymorphism

Syllabus

Introduction to Polymorphism, Types of Polymorphism, Operator Overloading - concept of overloading, operator overloading, Overloading Unary Operators, Overloading Binary Operators, Data Conversion, Type casting (implicit and explicit), Pitfalls of Operator Overloading and Conversion, Keywords explicit and mutable. Function overloading, Run Time Polymorphism - Pointers to Base class, virtual function and its significance in C++, pure virtual function and virtual table, virtual destructor, abstract base class.

Contents

- 3.1 *Introduction to Polymorphism*
- 3.2 *Early and Late Binding*
- 3.3 *Types of Polymorphism*
- 3.4 *Concept of Overloading*
- 3.5 *Operator Overloading*
- 3.6 *Type Casting (Implicit and Explicit)*
- 3.7 *Pitfalls of Operator Overloading and Conversion*
- 3.8 *Explicit and Mutable Keywords*
- 3.9 *Function Overloading*
- 3.10 *Run Time Polymorphism*
- 3.11 *Pointers to Base Class*
- 3.12 *Virtual Function and its Significance in C++*
- 3.13 *Pure Virtual Function and Virtual Table*
- 3.14 *Virtual Destructor*
- 3.15 *Abstract Base Class*

3.1 Introduction to Polymorphism

- Polymorphism means **many forms**. It is one of the important features of OOP.
- Polymorphism is basically an ability to create a variable, a function, or an object that has **more than one form**.
- The **primary goal** of polymorphism is an ability of the object of different types to respond to methods and data values by using the same name. The programmer does not have to know the exact type of the object in advance hence exact behavior of the object is determined at the run time.
- Polymorphism is concerned with the application of specific implementation or use of abstract base class.

3.1.1 Difference between Inheritance and Polymorphism

Sr. No.	Inheritance	Polymorphism
1.	Inheritance is a property in which some of the properties and methods of base class can be derived by the derived class.	Polymorphism is ability for an object to used different forms. The name of the function remains the same but it can perform different tasks.
2.	Various types of inheritance can be single inheritance, multiple inheritance, multilevel inheritance and hybrid inheritance.	Various types of polymorphism are compile time polymorphism and run time polymorphism. In compile time polymorphism there are two types of overloading possible. - functional overloading and operator overloading. In run time polymorphism there is a use of virtual function.

3.2 Early and Late Binding

Binding means connecting function calls to appropriate function definition. The function call can be associated with its definition at compile time or at run time. Depending upon it there are two types of binding defined - 1) **Early binding** or **static binding** and 2) **Late binding** or **dynamic binding**.

3.2.1 Early Binding

This is a type of binding in which the **function call is associated with the function definition** at the **compile time** only. Hence this type of binding is also called as early binding. Following example illustrates this concept -

```
*****
Program to demonstrate the static binding concept
*****/
#include<iostream>
```

```
using namespace std;
class Base
{
public :
    void display()
    {
        cout<<"\n In base class"<<endl;
    }
};

class Derived : public Base
{
public :
    void display()
    {
        cout<<"\n In derived class"<<endl;
    }
};

int main()
{
    Base b;
    Derived d;
    b.display();
    d.display();
    return 0;
}
```

Output

In base class

In derived class

Program Explanation

In above program, we have declared the single level inheritance. That is the **Derived** class is derived from the base class **Base**. In both of these classes we have declared a function by the same name as **display**. This function will get appropriately called when it is called using its object. This process of calling the correct function at correct time is called the early binding.

3.2.2 Late Binding

C++ provides facility to specify that the compiler should match function calls with the correct definition at the **run time**; this is called **late binding** or **dynamic binding**.

This type of binding is called the late binding because the compiler can not resolve the call to appropriate function late until the run time.

Late binding is achieved using **virtual functions**.

Following program illustrates this concept -

```
*****
Program to demonstrate the dynamic binding concept
*****/
```

```
#include<iostream>
using namespace std;
class Base
{
public :
    virtual void display()
    {
        cout<<"\n In base class"<<endl;
    }
};

class Derived : public Base
{
public :
    void display()
    {
        cout<<"\n In derived class"<<endl;
    }
};

int main()
{
    Base *ptr;
    Base b;
    Derived d;
    ptr=&b;
    ptr->display();
    ptr=&d;
    ptr->display();
    return 0;
}
```

Output

In base class

In derived class

Program Explanation

Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time. If we do not associate the keyword **virtual** to the **display()** function of base class then even though the address of derived class is assigned to the base class pointer **ptr**, it will select the **display** function of base class only. Thus late binding or dynamic binding is achieved by means of virtual function.

Difference between Early binding and Late binding

Sr. No.	Early binding	Late binding
1.	Static binding happens at the compile time.	Late binding happens at run time.
2.	Static binding is also called as early binding.	Late binding is also called as dynamic binding.
3.	There is no use of virtual function in this type of binding.	The virtual function is used in dynamic binding.
4.	It is more efficient than the late binding as extra level of indirection is involved in late binding.	It is more flexible than the early binding.

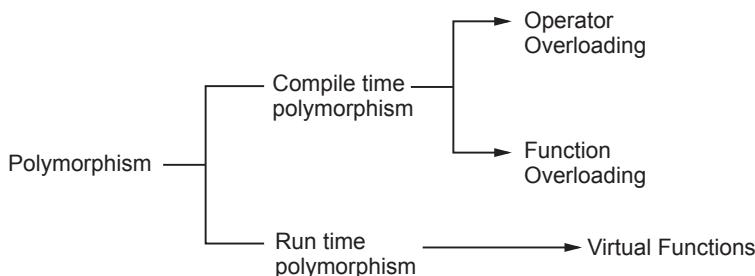
Review Question

1. Explain the following - Early binding and late binding.

3.3 Types of Polymorphism

Implementing Polymorphism

Polymorphism is an ability to have multiple forms using only one name. In C++ because of polymorphism feature with one function name we can perform different tasks at a time.



3.4 Concept of Overloading

- In C++, it is possible to specify more than one definition to function or an operator. This mechanism is called overloading
- There are two ways by which overloading occurs -
 - **Operator overloading**
 - **Function overloading**
- The overloaded declarations has the same name. That means, if the function is overloaded then there can be more than one function present in the C++ program with the same name. (It is perfectly allows in C++ although it raises error in C). But the overloaded declaration has different arguments and different implementation body.

3.5 Operator Overloading

Operator overloading is confusing even for excellent programmer but it is a strong feature of C++ if you could master it. The operators are used in mathematical expressions like

```
c=a+b;  
area=3.14*r*r;
```

It would be very nice if we could use these operators in our own objects. That means the string class can use + to concatenate two strings. That also means operators can be programmed to whatever we wish to do with them.

Operator overloading can be defined as an ability to define a new meaning for an existing (built-in) “operator”.

Various types of operators are -

- Mathematical operators such as + - * / ++
- Relational operators such as < > ==
- Logical operators such as && ||
- Access operators [] ->
- Assignment operator =
- Stream I/O operators << >>
- Type conversion operators and several others.

All of these operators have a predefined and unchangeable meaning for the built-in types. All of these operators can be given a specific interpretation for different classes or combination of classes. C++ provides the flexibility to the programmers in extending these built-in operators.

How to overload operator ?

Define a function with keyword *operator*. Then write the operator(such as +, [] or any other valid operator) as a function name. That means we can program that specific operator.

Restrictions on use of operators

- It's not possible to change an operator's precedence.
- It's not possible to create new operators, For example ^ which is used in some languages for exponentiation.
- You can not redefine ::, sizeof, ?:, or . (dot).
- =, [], and -> must be member functions if they are overloaded.
- ++ and -- need special treatment because they are prefix or postfix operators.
- Assignment (=) should always be overloaded if an object dynamically allocates memory.
- It can not change the number of required operands(unary, binary, ternary).
- Overloaded operator must be either,
 - Non static member function of class or
 - At least one parameter should be class or enumeration.

Makes no assumptions about similar operators. For example, the fact that you overloaded + does not mean that you have also defined += for your class type.

3.5.1 Rules for Operator Overloading

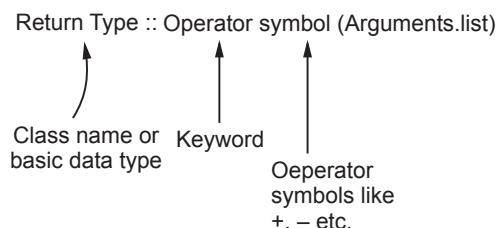
1. Only existing operators can be overloaded.
2. The basic meaning of the operator can not be changed.
3. Overloaded operators must follow the syntax of original operator. For example for binary operator operand1 operator operand2 is the syntax and this can not be changed during overloading.
4. Overloaded operators must have at least one operand that is of user defined type.
5. Binary arithmetic operators(+, -, * and /) must return a value.
6. When binary operators overloaded through a member function, the left hand operand must be an object of relevant class.
7. Binary operators overloaded through a member function must take one explicit argument.

8. Binary operators overloaded through friend function takes **two arguments**.
9. Unary operators **overloaded through a member function** must take **no explicit argument and no return value**.
10. **Unary operators** overloaded through friend function takes **one explicit argument**.

3.5.2 Overloading Unary Operators

- The unary operators require only one operand. In C++ the unary operators are -, +, !, ~, & and *.
- It can be declared as **member functions** taking no arguments. That means for any operator - - obj is interpreted as obj.operator-()
- It can be declared as **non member functions** taking one argument that must be the variable of class type (i.e. Object) or reference. That means, for any operator - the - obj is interpreted as operator-(obj).
- If both types of definitions are present then, the function declared as member takes the precedence.

Syntax : Function definition for operator overloading



Examples :

- 1) Point operator - ()
- 2) Vector Operator + (vector)

Following program illustrates the operator overloading of an unary operator !.

```
*****
Program for overloading ! operator. The overloading function can be a member or a
non member function.
*****
```

```
#include <iostream>
using namespace std;
class X
{
};

void operator!(X) //Defined operator
{
    cout<<"We are using operator!(X)"<< endl;
}
class Y
```

Defined operator
overloading function
as a **non - member**
function

```
{
public:
    void operator!() //Defined operator
    {
        cout<<"we are using Y::operator!()"<<endl;
    }
};

int main()
{
    X obj_x;
    Y obj_y;
    !obj_x; //invoking the non member function
    !obj_y; //invoking the member function
    return 0;
}
```

Defined operator
overloading function
as a **member** **function**

Output

```
We are using operator!(X)
we are using Y::operator!()
```

Program explanation :

In above example the operator function is -

- a non-member function with only one argument or
- a member function with no argument.

When there is an argument passed to a non-member function (like for class X) this argument is usually object of a class or reference to object of a class. The operator

function call `!obj_x` will be interpreted as `operator!(x)` and call to `!obj_y` will be interpreted as `Y.operator!()`

Example 3.5.1 Write a C++ program for overloading a unary minus operator.

Solution :

```
/*************************************************************************
Program for overloading an unary minus operator
*************************************************************************/
#include <iostream>
using namespace std;
class point
{
private:
    int x, y; // co-ordinate values
public:
    point( ) { x = 0; y = 0; }//constructor
    point(int i, int j) { x = i; y = j; } //constructor
    void get_xy(int &i, int &j) { i = x; j = y; }
    point operator-( ); // operator overload for unary
                         // minus
};
point point::operator-( )
{
    x = - x;
    y = - y;
    return *this; // Use of this pointer
}

int main( )
{
    point obj(10, 10);
    int x, y;
    clrscr();
    obj.get_xy(x, y);
    obj = - obj;           Negation calls operator - ()
    obj.get_xy(x, y);
    cout << "\n The use of unary operator is ... " < endl;
    cout << " X: " << x << ", Y: " << y;
    return 0;
}
```

Output

The use of unary operator is ...

X: -10, Y: -10

Program explanation :

Note that in above code the unary minus operator function is overloaded by passing no argument to it.

```
obj = -obj;
```

When this statement occurs the call to operator `-()` function is given. Thereby negated values of `x` and `y` are obtained.

Example 3.5.2 Write a C++ program for overloading a increment operator `++`.

Solution :

```
#include <iostream>
using namespace std;
class coord
{
private:
    int x, y; // co-ordinate values
public:
    coord( ) { x = 0; y = 0; } //constructor for obj
    coord(int i, int j) { x = i; y = j; } //constructor with param
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator++( ); //unary operator overloading
};

// Overload ++ operator for coord class
coord coord::operator++( )
{
    x++;
    return *this; //returning the current instance
}

int main( )
{
    int x, y;
    cout << "\n Enter the co-ordinates x and y ";
    cin >> x >> y;
    coord obj(x,y);
    ++obj; Calls coord operator++()
    obj.get_xy(x, y);
    cout << "The increment operator increments the co-ordinates as..." << endl;
    cout << "X: " << x << ", Y: " << y;
    getch();
    return 0;
}
```

Output

Enter the co-ordinates x and y 10 20

The increment operator increments the co-ordinates as...

X: 11, Y: 21

3.5.3 Overloading Binary Operators

- It can be declared as **member functions** taking one argument. That is, for any operator +, $x + y$ is interpreted as $x.\text{operator}+(y)$.
- It can be declared as **non-member functions** taking two arguments; one of these must be a variable of the class type or a reference to one. That is, for any operator @ except the assignment operator =, $x @ y$ is interpreted as operator @(x, y). For example $x+y$ is interpreted as operator +(x, y).
- If both kinds of definitions are present, the function declared as member takes precedence.

Examples : Binary plus and & (“bitwise and”)

```
class A {
A operator+( A& );
A operator&&( A& );
};
```

C++ Program

```
#include <iostream>
using namespace std;

class vector {
public:
    int p,q;
    vector() {p=0;q=0;} // constructor without parameters
    vector (int,int); //constructor with parameters
    vector operator + (vector); //definition of operator +
};

vector::vector (int a, int b) {
    p = a;
    q = b;
}

vector vector::operator+ (vector obj)
{
    vector temp;
    temp.p = p + obj.p;
    temp.q = q + obj.q;
    return (temp);
}

int main ()
```

```
{
    vector a (10,20);
    vector b (1,2);
    vector c;
    c = a + b;
    cout<<"\n The Addition of Two vectors is... ";
    cout << c.p << " and " << c.q;
    retrun 0;
}
```

Output

The Addition of Two vectors is...11 and 22

The vector class is created to store two vectors a(10,20) and b(1,2). The operator + is overloaded and now it will perform (10+1, 20+2) and thereby c.p will hold 11 and c.q will hold 22.

In above example, a constructor with parameters is defined

```
vector ::vector(int a,int b)
```

We have also defined another constructor

```
vector::vector( )
{
    p=0;
    q=0;
}
```

We have to explicitly define this initializing constructor because we have already defined one constructor with parameter. Now to create the objects of the class vector we need some initializing constructor.

```
c=a+b
```

As we perform addition the function operator + will be invoked and the addition of one vector (10+1) will be stored in c.p and (20+2) will be stored in c.q. We can replace c=a+b by c=a.operator+(b) because it is one and the same.

Example 3.5.3 Consider fruit basket with no. of apples and no. of mangoes as data members.

Overload the '+' operator to add the two objects of this class.

Solution :

```
#include<iostream>
using namespace std;
class FruitBasket
```

```

{
public:
    int NoOfApples;
    int NoOfMangoes;
    FruitBasket()
    {
        NoOfApples=0;NoOfMangoes=0;
    }
    FruitBasket(int,int);
    FruitBasket operator +(FruitBasket);
};

FruitBasket::FruitBasket(int a,int b)
{
    NoOfApples=a;
    NoOfMangoes=b;
}
FruitBasket FruitBasket::operator +(FruitBasket obj)
{
    FruitBasket temp;
    temp.NoOfApples=NoOfApples+obj.NoOfApples;
    temp.NoOfMangoes=NoOfMangoes+obj.NoOfMangoes;
    return temp;
}
int main()
{
    FruitBasket Basket1(100,200);
    FruitBasket Basket2(400,300);
    FruitBasket Total;
    Total=Basket1+Basket2;
    cout<<"The total Apples are: "<<Total.NoOfApples<<endl;
    cout<<"The total Mangoes are: "<<Total.NoOfMangoes<<endl;
    retrun 0;
}

```

Output

The total Apples are: 500
 The total Mangoes are: 500

3.5.4 More Programs on Operator Overloading

Example 3.5.4 Write a C++ program to concatenate two strings using operator overloading on + operator.

Solution :

```
#include<iostream>
#include<cstring>
```

```

using namespace std;
class string1
{
public:
    char S[15];
    string1()
    {
        strcpy(S,"\\0");
    }
    string1(char T[15])
    {
        strcpy(S,T);
    }
    string1 operator+(string1 k)
    {
        strcat(S,k.S);
        strcat(S,"\\0");
        return S;
    }
};
int main()
{
    string1 s1("Hello"),s2("Friends");
    string1 s;
    s=s1+s2;
    cout<<s.S<<endl;
    return 0;
}

```

Example 3.5.5 Define a class DATE, use overloaded + operator to add two dates and display the resultant-date. Assume non-leap year dates.

Solution :

```

#include <iostream>
using namespace std;
class DATE
{
private:
    int dd;
    int mm;
    int yy;
public:
    DATE(){}
    DATE(int d,int m,int y)
    {dd=d;mm=m;yy=y;}
    DATE operator+(DATE);

```

```
        void display();
};

DATE DATE::operator+(DATE D)
{
    DATE temp;
    int day,flag=0;
    int month;
    day=dd+D.dd;
    if(day>30)//total days exceeding month
    {
        day=day-30;
        flag=1; //flag 1 means sum of days exceeds one month
    }
    temp.dd=day;
    month=mm+D.mm;
    if(month>12)
    {
        if(flag==1)
            month=(month+1)-12;
        else
            month=month-12;
    }
    temp.mm=month;
    if(yy==D.yy)
        temp.yy=D.yy;
    else
        temp.yy=yy+D.yy;
    return temp;
}
void DATE::display(void)
{
    cout<<"\n";
    cout<<" Day: "<<dd;
    cout<<" Month: "<<mm;
    cout<<" Year: "<<yy;
}
int main()
{
    DATE d1,d2,d3;
    d1=DATE(30,10,10);
    d2=DATE(5,5,01);
    d3=d1+d2;
    d1.display();
    d2.display();
    d3.display();
    return 0;
}
```

Example 3.5.6 With an example, explain how to overload the pointer-to-member (\rightarrow) operator.

Solution : Overloading \rightarrow operator

The \rightarrow is a pointer operator for accessing the member of the pointer variable. The \rightarrow returns the pointer to the object of the class on which operator $\rightarrow()$ depends upon. Following is a simple program which illustrates the overloading of \rightarrow operator.

```
#include <iostream>
using namespace std;

class POINTER
{
public:
    int val;
    POINTER *operator->()
    {
        return this;
    }
};

int main()
{
    POINTER obj;
    obj->val=99;
    cout<<"Value assigned to the object is "<<obj->val;
    return 0;
}
```

Example 3.5.7 Explain how to overload subscript [].

Solution :

```
#include<iostream>
using namespace std;
class TEST {
    int ar[4];
public:
    TEST(int a,int b,int c,int d){
        ar[0]=a;
        ar[1]=b;
        ar[2]=c;
        ar[3]=d;
    }
    int &operator[](int i)
    {
        return ar[i];
    }
};

int main()
{
```

```

TEST obj(10,20,30,40);
clrscr();
cout<<"\nFirst element is: "<<obj[0];
cout<<"\n Second element is: "<<obj[2];
cout<<"\n Storing new element in an array";

obj[1]=77;
cout<<"\n New element at index 1 is: "<<obj[1];
return 0;
}

```

Example 3.5.8 Define class 'string'. Use overload '==' operator to compare two strings.

Solution : String is a collection of characters.

```

#include<iostream>
#include<cstring>
using namespace std;
class string1 {
    char S[10];
public:
    string1(){ }
    string1(char T[])
    {
        strcpy(S,T);
    }
    int operator==(string1 k)
    {

        if(strcmp(S,k.S)==0)
            return 1;
        else
            return 0;
    }
};
int main()
{
    string1 s1("testing"),s2("testing");
    if(s1==s2)
        cout<<"\nBoth the strings are equal!!";
    else
        cout<<"\nTwo strings are not equal!!";
    return 0;
}

```

Example 3.5.9 What is operator overloading ? Overload the numerical operators + and / for complex numbers addition and division respectively.

Solution : Operator overloading is defined as an ability to define a new meaning for an existing operator.

```
/*Program for overloading numerical operator + and / for complex numbers
addition and division*/
#include<iostream>
using namespace std;
class complex {
public:
    float real,img;
    complex(){real=0;img=0;}
    complex(float,float);
    complex operator+ (complex);
    complex operator/ (complex);
};
complex::complex(float r,float i)
{
    real=r;
    img=i;
}
complex complex::operator+ (complex obj)
{
    complex temp;
    temp.real=real+obj.real;
    temp.img=img+obj.img;
    return (temp);
}
complex complex::operator/ (complex obj)
{
    complex temp;
    float new_temp;
    new_temp=(obj.real*obj.real)+(obj.img*obj.img);
    temp.real=((real*obj.real)+(img*obj.img))/new_temp;
    temp.img=new_temp;
    return(temp);
}
int main()
{
    complex a(2,6);
    complex b(4,1);
    complex c;
    c=a+b;
    cout<<"\n The addition of two complex numbers is...";
    cout<<c.real<<" and "<<c.img<<"i";
    c=a/b;
    cout<<"\n The Division of two complex numbers is...";
    cout<<c.real<<" and "<<c.img<<"i";
    retrun 0;
}
```

Example 3.5.10 Write a C++ program that takes two values of time (hr, min, sec) and outputs their sum using constructors and operator overloading.

Solution :

```
#include <iostream>
using namespace std;
class DATE
{
private:
    int hr;
    int min;
    int sec;
public:
    DATE(){}
    DATE(int h,int m,int s)
    {hr=h;min=m;sec=s;}
    DATE operator+(DATE);
    void display();
};

DATE DATE::operator+(DATE D)
{
    DATE temp;
    int hour,minute,seconds,flag1,flag2;
    flag1=0;
    flag2=0;
    hour=hr+D.hr;
    minute=min+D.min;
    seconds=sec+D.sec;
    if(minute>60)
    {
        minute=minute-60;
        flag1=1;
    }
    if(seconds>60)
    {
        seconds=seconds-60;
        flag2=1;
    }
    if(flag1==1)
    {
        hour=hour+1;
    }
    if(flag2==1)
    {
        minute=minute+1;
    }
    temp.hr=hour;
    temp.min=minute;
```

```

        temp.sec=seconds;
        return temp;
    }
void DATE::display(void)
{
    cout<<"\n";
    cout<<" hour: "<<hr;
    cout<<" Minutes: "<<min;
    cout<<" Seconds: "<<sec;
}
int main()
{
    DATE t1,t2,t3;
    t1=DATE(30,10,10);
    t2=DATE(5,57,81);
    t3=t1+t2;
    t1.display();
    t2.display();
    t3.display();
    cout<<endl;
    return 0;
}

```

Output

hour: 30 Minutes: 10 Seconds: 10
 hour: 5 Minutes: 57 Seconds: 81
 hour: 36 Minutes: 8 Seconds: 31

Review Questions

1. Enlist the rules used for operator overloading.
2. What is operator overloading ? Write a program to overload operator + for concatenation of two string.

SPPU : May-14, Marks 8**3.6 Type Casting (Implicit and Explicit)**

The type conversion is a process in which variable in one data type can be converted to another data type.

Some data types can be automatically converted by the compiler. This **automatic** conversion is called as **implicit conversion**.

For example :

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

```

int x;
float y=99.99;
x=y;----- here value of x becomes 99 and .99 will be truncated.
}

```

Similarly, we can make use of type cast to explicitly convert one type to another basic type. This type of conversion is called explicit conversion.

3.6.1 Basic Type to Class Type

We can assign some value associated with some basic type to a class. Following program allows us to assign **int** value to a **class**. This can be accomplished with the help of constructor.

C++ Program

```

#include<iostream>
using namespace std;
class image
{
    int ht;
    int wd;
public:
    image(int a)
    {
        ht=a;
        wd=a*5;
    }
    void display()
    {
        int area=ht*wd;
        cout<<area;
    }
};
int main()
{
    image obj1(10);
    cout<<"\nArea of 10 and 50 is: ";
    obj1.display();
    int new_val=11;
    cout<<"\nArea of 11 and 55 is: ";
    obj1=new_val; //int value is assigned to the class
    obj1.display();
    return 0;
}

```

Output

Area of 10 and 50 is: 500
Area of 11 and 55 is: 605

3.6.2 Class Type to Basic Type

We can get a class value and assign it to some variable. This type of conversion is possible by **casting the operator**.

Here is the illustration-

```
#include<iostream>
using namespace std;
class test
{
public:
operator int();
};
test::operator int()
{
    int sum;
    sum=5;
    return sum;
}
int main()
{
    test obj;
    int x;
    x=10;
    cout<<"\nValue of x: "<<x;
    x=int(obj); //class assigns value to a variable
    cout<<"\nValue of x assigned by the class: "<<x;
    retrun 0;
}
```

Syntax of conversion function

```
operator data_type ( )
{
    .....
    .....
    .....
    return value;
}
```

Output

Value of x: 10
Value of x assigned by the class: 5

3.6.3 Class Type to another Class Type

The variable of one class type can be converted to the another class type. The variable of class type is nothing but an object of that class. That means the object of one class can be converted to the object of another class.

For this conversion there are two ways -

1. Use of some conversion function generally denoted by the keyword **operator**
2. Use of constructor

```
#include <iostream>
#include <cstring>
using namespace std;
class Minutes
{
```

```
private:  
    double m;  
  
public:  
    Minutes(double x)  
    {  
        m=x;  
    }  
    void display()  
    {  
        cout<<m<<" minutes ";  
    }  
    double get_data()  
    {  
        return m;  
    }  
};  
class Seconds  
{  
    double s;  
public:  
    Seconds(double y)  
    {  
        s=y;  
    }  
    void display()  
    {  
        cout<<s<<" Seconds ";  
    }  
    operator Minutes() //converter function  
    {  
        return Minutes(s/60);  
    }  
    Seconds(Minutes m) // constructor  
    {  
        s=m.get_data()*60;  
    }  
};  
int main()  
{  
    cout<<"Conversion using the operator function\n";  
    Seconds sec1=90;  
    Minutes min1=sec1; //conversion from Seconds class to Minutes class  
    sec1.display();  
    cout<<" = ";  
    min1.display();  
    cout<<"\n\nConversion using Constructor\n";
```

```
Minutes min2=5;
Seconds sec2 = min2; //conversion from Minutes to Seconds class
min2.display();
cout << " = ";
sec2.display();
cout << endl;
return 0;
}
```

Output

Conversion using the operator function
90 Seconds = 1.5 minutes

Conversion using Constructor
5 minutes = 300 Seconds

Review Questions

1. Enlist the restrictions on use of operators in operator overloading.
2. What is the need of operator overloading ?
3. Enlist the rules used for operator overloading.
4. Write a C++ program for overloading unary minus operator.
5. Explain how friend function can be used in operator overloading ?
6. Explain a C++ program that illustrates the type conversion from basic type to class type.

3.7 Pitfalls of Operator Overloading and Conversion

There are following pitfalls of operator overloading and conversion -

1. The built-in operation can become **expensive**.
2. There is **increased complexity** in the program
3. The operator overloading represents **odd syntax**, hence it is difficult to trace the call to the function. For example - it is easy to trace the call for the function **addition** but it is difficult to trace for **+**. Due to this, the development and debugging activity gets slow down.
4. There are some operators that can not be overloaded. For example **sizeof**, **::**, **?:**
5. One **cannot change syntactical characteristics** of operator even if it is overloaded.
6. In many cases operator overloading is **not preferred**. For example **+** operator can be overloaded to add the objects of distance class but it is not preferred to add the employee data of employee class.

7. The operator overloading may lead to **ambiguity**. For example - Suppose you use both a one-argument constructor and a conversion operator to perform the same conversion then compiler will not be able to understand which conversion to use.

3.8 Explicit and Mutable Keywords

In this section we will discuss two less commonly used keywords **explicit** and **mutable**.

3.8.1 The Explicit Keyword

- The constructor is for creating the instance of class(i.e. Object). The name of the constructor is same as the class name.
- By default, the constructors are provided by the compiler but the programmer can explicitly create it.
- In C++, the compiler is allowed to make one **implicit conversion** to resolve the parameters to a function. Prefixing the **explicit** keyword to the constructor prevents the compiler from using that constructor for **implicit conversions**.
- In C++, a constructor with only one required parameter is considered an implicit conversion function. It converts the parameter type to the class type.
- Following C++ program shows this type of implicit conversion.

```
#include<iostream>
using namespace std;
class Test
{
    int Val;
public:
    Test(int x) :Val(x)
    {
        cout << "\n Val= " << Val;
    };
};
int main()
{
    Test obj = 100;// Implicit conversion from int to object
    return 0;
}
```

Output

Val = 100

- Now if we simply add the keyword **explicit** in above code for the constructor then we will get the compilation error. Here is the code illustrating the keyword **explicit**.

```
#include<iostream>
using namespace std;
class Test
{
    int Val;
public:
    explicit Test(int x) :Val(x)
    {
        cout << "\n Val= " << Val;
    };
};
int main()
{
    Test obj = 100;
    return 0;
}
```

Output

Error: 'initializing': cannot convert from 'int' to 'Test'

Program Explanation : In above program, due to use of keyword **explicit**, the implicit conversion is prevented. Thus the keyword explicit forces the compiler not to do an implicit conversion and throws an error for this kind of code. And thus make sure that the constructor is explicitly called with the right syntax. This keyword is used only for the constructors.

Advantage of using explicit conversion

The explicit keyword makes a conversion constructor to non-conversion constructor. As a result, the code is less error prone.

3.8.2 The Mutable Keyword

Before understanding the mutable keyword let us first understand the concept of **constant object**.

const class object

When the object is declared or created with const then its data members can not be changed.

Mutable Keyword

Mutable keyword is used with the data members which we want to change even if the object of that class is constant.

Following program shows the use of mutable keyword

```
#include<iostream>
using namespace std;
```

```
class Test
{
    int a;
    mutable int b;
public:
    Test()
    {
        a = 0;
        b = 0;
    }
    void change()const
    {
        a = a + 10;//Error: 'a' cannot be modified
        //because it is being accessed through a const object
    }
    b = b + 20;// No error for this line as b is mutable
}
};

int main()
{
    const Test obj;
    obj.change();
    return 0;
}
```

Thus using the keyword **mutable** the data members of class with constant object can be changed.

Review Question

1. Explain the keywords **explicit** and **mutable** with illustrating examples.

3.9 Function Overloading

Definition : Function overloading is a concept in which one can use **many functions** having **same function name** but can pass different number of parameters or different types of parameters.

3.9.1 Rules for Function Overloading

Following are the **rules for function overloading** -

1. The overloaded functions may differ by number of parameters.
2. The overloaded functions may differ by data types.
3. The same function name is used for various instances of function call.

For example, if there is a function *sum* which performs addition operation then one can use overloading functions like this -

```
int sum(int a,int b);
int sum(int a,int b,int c);
int sum(int a,int b,int c,int d);
```

Similarly the function overloading can be achieved like this -

```
int sum(int a,int b);
float sum(float a,float b);
double sum(double a,double b);
char sum(char a,char b);
```

That means we can handle different number of parameters or different types of parameters using the same function name.

Following program takes different number of parameters and performs the task of addition. Note that name of these functions is same and that is *sum*.

C++ Program

```
#include <iostream>
using namespace std;
class test {
    private:
        int a,b,c,d;
    public:
        int sum(int,int);
        int sum(int,int,int);
        int sum(int,int,int,int);
    };
int test::sum(int a,int b)
{
    return (a+b);
}
int test::sum(int a,int b,int c)
{
    return (a+b+c);
}
int test::sum(int a,int b,int c,int d)
{
    return (a+b+c+d);
}
int main()
{
    test obj;
    int a, b, c, d,choice;
    int result=0;
    cout<<"\n\t\t Main Menu";
```

```

cout<<"\n\t 1.Addition of two numbers";
cout<<"\n\t 2.Addition of three numbers";
cout<<"\n\t 3.Addition of four numbers"<endl;
cout<<"\n Enter Your choice : ";
cin >> choice;
switch(choice)
{
    case 1: cout <<"\nEnter 2 numbers: ";
              cin >> a >> b;
              result = obj.sum(a,b);
              break;
    case 2: cout <<"\nEnter 3 numbers: ";
              cin >> a >> b >> c;
              result = obj.sum(a,b,c);
              break;
    case 3: cout <<"\nEnter 4 numbers: ";
              cin >> a >> b >> c >> d;
              result = obj.sum(a,b,c,d);
              break;
    default: cout <<"\nWrong Choice";
              break;
}
cout <<"\n\nresult: " << result << endl;
return 0;
}

```

Overloaded Functions

Output

Main Menu
 1.Addition of two numbers
 2.Addition of three numbers
 3.Addition of four numbers

Enter Your choice : 2

Enter 3 numbers: 10 20 30
 result: 60

Now following program will take the parameters of different data types. This makes the function to overload.

C++ Program

```

#include <iostream>
using namespace std;
class test {
public:
    int sum(int,int);
    float sum(float,float);
    double sum(double,double);
}

```

```
    };
int test::sum(int a,int b)
{
return (a+b);
}
float test::sum(float a,float b)
{
return (a+b);
}
double test::sum(double a,double b)
{
return (a+b);
}
int main()
{
test obj;
int choice;
int a,b;
float x,y;
double m,n;
double result=0;
cout<<"\n\t\t Main Menu";
cout<<"\n\t 1.Addition of two integer numbers";
cout<<"\n\t 2.Addition of float numbers ";
cout<<"\n\t 3.Addition of double numbers" << endl;
cout<<"\n Enter Your choice : ";
cin >> choice;
switch(choice)
{
case 1: cout <<"\nEnter 2 numbers: ";
cin >> a >> b;
result = obj.sum(a,b);
break;
case 2: cout <<"\nEnter 2 numbers: ";
cin >> x >> y;
result = obj.sum(x,y);
break;
case 3: cout <<"\nEnter 2 numbers: ";
cin >> m >> n;
result = obj.sum(m,n);
break;
default: cout <<"\nWrong Choice";
break;
}
cout <<"\n\nresult: " << result << endl;
return 0;
}
```

Output

Main Menu

1.Addition of two integer numbers

2.Addition of float numbers

3.Addition of double numbers

Enter Your choice : 2

Enter 2 numbers: 1.13 5.6

result: 6.73

Example 3.9.1 Write a C++ program that can take either two integers or two floating point numbers and outputs the smallest number using class, function overloading.

Solution :

```
#include<iostream>
using namespace std;
class Test
{
public:

    int smallest(int,int);
    float smallest(float,float);
};

int Test::smallest(int a,int b)
{
    if(a<b)
        return a;
    else
        return b;
}
float Test::smallest(float x,float y)
{
    if(x<y)
        return x;
    else
        return y;
}
int main()
{
    Test obj;
    int choice;
    int a,b;
    float x,y;
    cout<<"\n Menu"<<endl;
    cout<<"\n 1. Integers \n 2.Float "<<endl;
    cout<<"\n Enter your choice: ";
    cin>>choice;
```

```

switch(choice)
{
case 1:    cout<<"\n Enter two numbers: "<<endl;
            cin>>a;
            cin>>b;
            cout<<"Smallest Number is : "<<obj.smallest(a,b);
            break;
case 2:    cout<<"\n Enter two numbers: "<<endl;
            cin>>x;
            cin>>y;
            cout<<"Smallest Number is : "<<obj.smallest(x,y);
            break;
}
return 0 ;
}

```

Output

Menu

1. Integers
2. Float

Enter your choice: 1

Enter two numbers:

11

7

Smallest Number is : 7

3.10 Run Time Polymorphism**SPPU : Dec.-16, Marks 4**

- Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.
- The runtime polymorphism can be achieved by function overriding or method overriding.
- Redefining a function in a derived class is called function overriding.
- Function overloading means within the class we can declare same function name, but arguments and return types are different and function overriding means the function name is same but the task carried out with it is different.

C++ Program

```

#include <iostream>
using namespace std;
class A
{
private:
    int a,b;

```

```

public:
    void get_msg()
    {
        a=10;
        b=20;
    }
    void print_msg()const
    {
        int c;
        c=a+b;//performing addition
        cout<<"\n C(10+20)= "<<c;
        cout<<"\n I'm print_msg() in class A";
    }
};

class B : public A
{
private:
    int a,b;
public:
    void set_msg()
    {
        a=100;
        b=10;
    }
    void print_msg()
    {
        int c;
        c=a-b;//performing subtraction in this function
        cout<<"\n\n C(100-10) = "<<c;
        cout<<"\n I'm print_msg() in class B ";
    }
};

void main()
{
    A obj_base;
    B obj_derived;
    obj_base.get_msg();
    obj_base.print_msg(); //same function name
    obj_derived.set_msg();
    obj_derived.print_msg(); //but different tasks
}

```

Output

C(10+20)= 30
I'm print_msg() in class A

C(100-10) = 90
I'm print_msg() in class B

Program Explanation : In above program,

- 1) We have written two functions `print_msg()` by the same name but performing different operation.
- 2) The `print_msg()` function in **class A** (i.e. base class) is performing addition of two numbers and the `print_msg()` function in **class B** is performing the subtraction of two numbers.
- 3) Both the functions have same return type, same name and no parameters but their role is changing.
- 4) This mechanism of changing the task of some function in derived class is called **function overriding**.

Example 3.10.1 Differentiate compile time and run time polymorphism.

Solution :

Sr. No.	Compile time polymorphism	Run time polymorphism
1.	The call to the functions having the same name is resolved at compile time .	The call to the functions having same name is resolved at run time .
2.	In this type of polymorphism the function overloading mechanism is used.	In this type of polymorphism, the function overriding mechanism is used.
3.	During compile time polymorphism, the function overloading and operator overloading techniques are used.	During run time polymorphism, the virtual functions and pointers are used.
4.	It provides fast execution .	It provides slow execution .
5.	It is less flexible as all the decisions are to be taken at compile time itself.	It is more flexible as the execution is delayed for execution time.

Example 3.10.2 Differentiate between function overloading and function overriding.

Solution :

Sr. No.	Function overloading	Function overriding
1.	Function overloading occurs in same class.	The function overriding occurs in child class when the child class function overrides the parent class function.
2.	In function overloading the signature must be different for all overloaded functions.	In function overriding, the signature must be the same.

3.	Function overloading occurs during compile time polymorphism.	Function overriding occurs during run time polymorphism.
4.	In function overloading, we can have any number of overloaded functions.	In function overriding we can have only one overriding function in child class.

3.11 Pointers to Base Class

One of the key feature of inheritance mechanism is that the pointer to derived class is type compatible with pointer to its base class.

That means we can create a pointer variable of a base class and assign the addresses of derived class's objects to it.

For example -

```
#include <iostream>
using namespace std;

class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
    { width=a; height=b; }
};

class Rectangle: public Polygon {
public:
    int area()
    { return width*height; }
};

class Triangle: public Polygon {
public:
    int area()
    { return width*height/2; }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon * ppol1 = &rect;
    Polygon * ppol2 = &trgl;
    ppol1->set_values (4,5);
    ppol2->set_values (4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```

3.12 Virtual Function and its Significance in C++

Definition : "A *virtual function* is a member function that is declared within a base class and redefined by a derived class."

The **virtual** keyword is preceded to the function name. The virtual function can be redefined in the derived class. Thus using virtual functions we can have **one interface, multiple functions** performing **different task**. This feature is called **polymorphism**. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class indicate some different task related to derived class.

When a virtual function is redefined by a derived class, the keyword **virtual** is not needed. The virtual function written in base class acts as **interface** and the function defined in derived classes act as different forms of the same function. This property of virtual function brings the **runtime polymorphism**. The following program shows this property -

```
#include <iostream>
using namespace std;
class base
{
public:
    int i;
    base(int x)//constructor
    {
        i = x;
    }
    virtual void function() { // The function is with keyword virtual
    {
        cout << "Using base version of function(): ";
        cout << i << "\n";
    }
};
class derived1 : public base
{
public:
//constructor for object creation
derived1(int x):base(x){ }
void function()
{
    cout < "Using derived1's version of function(): ";
    cout < i+i << "\n";
}
};
```

```
class derived2 : public base
{
public:
    derived2(int x):base(x){ }//constructor for object creation
    void function()
    {
        cout << "Using derived2's version of function(): ";
        cout << i*i << "\n";
    }
};

int main()
{
    base *p;
    int num;
    cout<<"\n Enter Some number ";
    cin>>num;
    base obj(num);
    derived1 d1_obj(num);
    derived2 d2_obj(num);
    p=&obj;//base object's address
    p->function(); //base class function
    p = &d1_obj;//derived1 object's address
    p->function(); //derived1 class function()
    p = &d2_obj;//derived2 object's address
    p->function(); //derived2 class function()
    return 0;
}
```

Output

```
Enter Some number 5
Using base version of function(): 5
Using derived1's version of function(): 10
Using derived2's version of function(): 25
```

In above program we have written one interface function in base class preceded by the keyword *virtual*. Note that the derived class *derived1* is also having *function* who is performing addition operation. This time this function is not preceded by keyword *virtual*. Similarly we have written another derived class *derived2* in which function is performing the task of multiplication.

Using the object pointer we can access different forms of the functions.

Example 3.12.1 Write a C++ program to create a base class house. There are two classes called door and window available. The house class has members which provide information related to the area of construction, doors and windows detail. It delegates the responsibility of computing cost of doors and window construction to door and window classes respectively. Write a C++ program to model the above relationship and find the cost of constructing the house.

Solution :

```
#include<iostream>
using namespace std;
class House
{
public:
    int area;
    double rate;
    int noofdoors;
    int noofwindows;
    void GetVal()
    {
        cout<<"\n Enter the area for construction(in sq.feet): ";
        cin>>area;
        cout<<"\n Enter the rate per sq.feet: ";
        cin>>rate;
        cout<<"\n Enter the number of dooors: ";
        cin>>noofdoors;
        cout<<"\n Enter the number of windows: ";
        cin>>noofwindows;
    }
    double AreaCost()
    {
        return (area*rate);
    }
};

class door:virtual public House
{
public:
    double costofdoor;
    void GetDoorVal()
    {
        cout<<"\n Enter the cost of each door: ";
        cin>>costofdoor;
    }
    double DoorCost()
    {
        return (costofdoor*noofdoors);
    }
}
```

```
};

class window:virtual public House
{
public:
    double costofwindow;
    void GetWindowVal()
    {
        cout<<"\n Enter the cost of each window: ";
        cin>>costofwindow;
    }

    double WindowCost()
    {
        return (costofwindow*noofwindows);
    }
};

class derived:public door,public window
{
public:
    double ConstructionCost()
    {
        double total;
        double c1=WindowCost();
        double c2=DoorCost();
        double c3=AreaCost();
        total=c1+c2+c3;
        return total;
    }
};

int main()
{
    derived obj;
    obj.GetVal();
    obj.GetDoorVal();
    obj.GetWindowVal();
    cout<<"\n The construction cost is: "<<obj.ConstructionCost();

    return 0;
}
```

Output

Enter the area for construction(in sq.feet): 500

Enter the rate per sq.feet: 100

Enter the number of dooors: 4

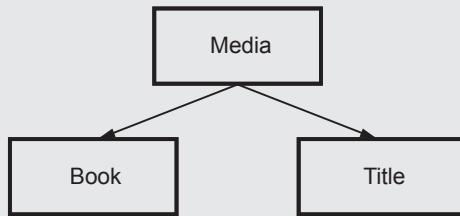
Enter the number of windows: 2

Enter the cost of each door: 40

Enter the cost of each window: 20

The construction cost is: 50200

Example 3.12.2 Consider an example of a book shop which sells book and video tapes. These two classes are inherited from the base class called media. The media class has command data members such as title and publication. The book class has data members for storing number of pages in book and tape class has the playing time in a tape. Each class will have member function such as read() and show() in the base class, these members have to be defined as virtual functions, write a program which models the class hierarchy for the book shop and process the objects of these classes using pointers to the base class.



Solution :

```

#include <iostream>
using namespace std;
class Media
{
public:
    char Title[10], publication[20];
    virtual void Read()
    {
        cout << "\n Enter the title: ";
        cin >> Title;
        cout << "\n Enter the publication: ";
        cin >> publication;
    }
    virtual void show()
    {
        cout << "Title: " << Title << endl;
        cout << "Publication: " << publication << endl;
    }
};
  
```

```
class Book : public Media
{
public:
    int pages;
    virtual void Read(int num)
    {

        pages = num;
    }
    virtual void show()
    {
        cout << "Number of pages of book: ";
        cout << pages << "\n";
    }
};

class Tape : public Media
{

public:
    int playtime;
    virtual void Read(int num)
    {

        playtime = num;
    }
    virtual void show()
    {
        cout << "Playing time in tape: ";
        cout << playtime << "\n";
    }
};

int main()
{
    Media *p;
    Media obj;
    int num;
    Book b;
    Tape t;
    obj.Read();
    obj.show();
    cout << "\nEnter number of pages: ";
    cin >> num;
    b.Read(num);
    p=&b;
    p->show();
    cout << "\nEnter Time in minutes: ";
```

```

    cin >> num;
    t.Read(num);
    p = &t;
    p->show();
    return 0;
}

```

Output

Enter the title: AAA

Enter the publication: BBB

Title: AAA

Publication: BBB

Enter number of pages: 100

Number of pages of book: 100

Enter Time in minutes: 3

Playing time in tape: 3

3.13 Pure Virtual Function and Virtual Table

A pure virtual function is a virtual function which is to be implemented by derived class. The class that contains the pure virtual function is called the **abstract class**.

The pure virtual functions are declared using pure specifier i.e. = 0

Following program demonstrate the use of pure virtual function.

The class A is an abstract class in which the pure virtual function Display() is declared. This function is overridden by the two derived classes - class B and class C.

```

/*********************************************************************
Program for demonstrating the pure virtual function
*****
#include<iostream>
using namespace std;
class A
{
public:
    virtual void Display()=0; // Pure Virtual Function
};

class B : public A
{
public:
    void Display()
    {
        cout<<"\n In Derived Class B"<<endl;
    }
};

```

```
    }
};

class C : public A
{
public:
    void Display()
    {
        cout<<"\n In Derived Class C" << endl;
    }
};

int main()
{
    A *p;
    B b;
    C c;
    p = &b;
    p->Display(); //using pointer the method of derived class is accessed
    p = &c;
    p->Display();
    return 0;
}
```

Output

In Derived Class B

In Derived Class C

Program Explanation :

The base class **A** derived two classes namely - **B** and **C**. In class **A**, the pure virtual function **Display** is defined by assigning = 0.

The **Display** function is overridden in class **B** and class **C**.

In main function, the pointer object to the base class is declared. It is denoted by variable **p**. By assigning the address of class **B** to the pointer variable **p**, the function **Display** of class **B** is invoked. Similarly by assigning the address of class **C** the **Display** function of class **C** is invoked.

3.14 Virtual Destructor

Destructor is basically used to **de-allocate the memory** allocated for the objects. Thus use of destructor is to clean up the memory allocated for the class members. The destructor is denoted using ~ symbol.

When a class is derived from a base class then on calling the destructor, it does not destruct the memory of derived class. This problem can be fixed up by making the base class destructor virtual.

Following program illustrates this idea

```
#include<iostream>
using namespace std;
class Base
{
public:
Base()
{
    cout<<"\n Calling Base class Constructor";
}
~Base()
{
    cout<<"\n Calling Base class Destructor";
}
};

class Derived:public Base
{
public:
Derived()
{
    cout<<"\n Calling Derived class Constructor";
}
~Derived()
{
    cout<<"\n Calling Derived class Destructor";
}
};

int main()
{
Base *obj=new Derived(); //object creation
delete obj;
return 0;
}
```

Output

```
Calling Base class Constructor
Calling Derived class Constructor
Calling Base class Destructor
```

Now change the above program as

```
#include<iostream>
using namespace std;
class Base
{
public:
Base()
```

```
{  
    cout<<"\n Calling Base class Constructor";  
}  
virtual ~Base()          //Note that this is virtual constructor  
{  
    cout<<"\n Calling Base class Destructor";  
}  
};  
class Derived:public Base  
{  
public:  
    Derived()  
    {  
        cout<<"\n Calling Derived class Constructor";  
    }  
    ~Derived()  
    {  
        cout<<"\n Calling Derived class Destructor";  
    }  
};  
int main()  
{  
    Base *obj=new Derived(); //object creation  
    delete obj;  
    return 0;  
}
```

Output

Calling Base class Constructor
Calling Derived class Constructor
Calling Derived class Destructor
Calling Base class Destructor

Review Question

1. Explain virtual destructor with example.

3.15 Abstract Base Class

- Abstract class is a class which is mostly used as a base class. It contains at **least one pure virtual function**. Abstract classes can be used to specify an interface that must be implemented by all subclasses.
- The virtual function is function having nobody but specified **by = 0**. This tells the compiler that nobody exists for this function relative to the base class. When a *virtual* function is made **pure**, it forces any derived class to override it. If a derived

class does not, an error occurs. Thus, making a *virtual* function **pure** is a way to guarantee that a **derived class will provide its own redefinition**.

For example

```
#include <iostream>
using namespace std;

class area
{
    double dim1, dim2;
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0; // pure virtual function
};

class square : public area
{
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area
{
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
```

```

area *p;
square s;
triangle t;
int num1,num2;
cout<<"\n Enter The two dimensions for calculating area of
square";
cin>>num1>>num2;
s.setarea(num1,num2);
p = &s;
cout << "Area of square is : " << p->getarea() << '\n';
cout<<"\n Enter The two dimensions for calculating area of
triangle";
cin>>num1>>num2;
t.setarea(num1,num2);
p = &t;
cout << "Area of Triangle is: " << p->getarea() << '\n';
return 0;
}

```

Output

Enter The two dimensions for calculating area of square
10 20
Area of square is : 200

Enter The two dimensions for calculating area of triangle
6
8
Area of Triangle is: 24

In above code the *getarea()* is a function which is defined as pure virtual function in base class. But it is redefined in derived class *square* and *triangle*. In derived class *square* the *getarea()* function calculates the area of square and in derived class *triangle* the *getarea()* function calculates the area of triangle. The definition of *getarea* in base class is overridden by the definitions of functions in derived class. The class *area* acts as an abstract class because -

- It specifies an interface which is used by all the derived classes. Thus it is never used directly it simply gives skeleton to other derived classes.
- It contains one pure virtual function *getarea()*.

Example 3.15.1 What are abstract classes ? Write a program having student as an abstract class and create many derived classes such as engineering, science, medical etc. from the student class. Create their object and process them.

Solution :

```

#include<iostream>
#include<cstring>
using namespace std;

```

```
class Student
{
    char name[10];
public:
    void SetName(char n[10])
    {
        strcpy(name,n);
    }
    void GetName(char n[10])
    {
        strcpy(n,name);
    }
    virtual void qualification()=0;
};

class Engg:public Student
{
public:
    void qualification()
    {
        char n[10];
        GetName(n);
        cout<<n<<" is a an engineering student"<<endl;
    }
};
class Medical:public Student
{
public:
    void qualification()
    {
        char n[10];
        GetName(n);
        cout<<n<<" is a medical student"<<endl;
    }
};
int main()
{
    Student *s;
    Engg e;
    Medical m;
    char nm[10];
    cout<<"\n Enter the name: "<<endl;
    cin>>nm;
    e.SetName(nm);
    s=&e;
    s->qualification();
    cout<<"\n Enter the name: "<<endl;
```

```
    cin >> nm;
    m.SetName(nm);
    s = &m;
    s->qualification();
    return 0;
}
```

Output

Enter the name:

Ramesh

Ramesh is a an engineering student

Enter the name:

Suresh

Suresh is a medical student

Review Question

1. Write a short note on - abstract classes.



Unit - IV

4

Files and Streams

Syllabus

Data hierarchy, Stream and files, Stream Classes, Stream Errors, Disk File I/O with Streams, File Pointers and Error Handling in File I/O, File I/O with Member Functions, Overloading the Extraction and Insertion Operators, memory as a Stream Object, Command-Line Arguments, Printer output.

Contents

- 4.1 Stream and Files
- 4.2 Libraries
- 4.3 Stream Classes
- 4.4 Stream Errors
- 4.5 Disk File I/O with Streams
- 4.6 Unformatted I/O Functions
- 4.7 Formatted I/O Functions and I/O Manipulators
- 4.8 File Pointers
- 4.9 Error Handling in File I/O
- 4.10 File I/O with Member Functions
- 4.11 Overloading the Extraction and Insertion Operators
- 4.12 Memory as a Stream Object
- 4.13 Command-Line Arguments
- 4.14 Printer Output

4.1 Stream and Files

Stream is basically a **channel** on which data flow from sender to receiver. Data can be sent out from the program on an output stream or received into the program on an input stream. For example, at the start of a program, the standard input stream "cin" is connected to the keyboard and the standard output stream "cout" is connected to the screen. In fact, input and output streams such as "**cin**" and "**cout**" are examples of stream objects. In C++ the entire I/O (Input/Output) system operates through streams. A stream is connected to a physical device by the C++ input output system (Popularly known as I/O system).

When C program begins execution three **predefined streams** are automatically opened and those are `stdin`, `stdout`, `stderr` and when C++ program begins four streams get opened - `cin`, `cout`, `cerr` and `clog`.

Stream	Meaning	Physical device
cin	Standard input	Connected to Keyboard
cout	Standard output	Connected to Screen
cerr	Standard error	Connected to Screen
clog	Buffer of error	Connected to Screen

The header file named *iostream.h* supports these I/O operations.

4.2 Libraries

As we know stream is basically name given to flow of data. In C++ the stream is represented by object of a particular class. For example **cin** and **cout** are basically the objects of *istream_withassign* and *ostream_withassign* classes. And these two classes are basically derived from *istream* and *ostream* classes. The declaration of the object *cin* and *cout* is done already in *iostream.h* file. And therefore we need not have to declare *cin* and *cout* objects in our program. Hence we simply include *iostream.h* so that compiler can then understand *cin* and *cout*. Thus in C++ library is a collection of classes and functions. The standard I/O library is called *iostream* library. The *iostream* header files are summarized in following table.

Header file	Meaning
<i>iostream.h</i>	This is the most commonly used header file in any C++ program. It defines hierarchy of classes which includes collection of classes from low level I/O to high level I/O.
<i>iomanip.h</i>	It defines the set of manipulator. Using these manipulators the streams can be modified and useful effects can be given.

strstream.h	It includes set of classes with respect to character array. The definition of various classes such as <i>istrstream</i> , <i>ostrstream</i> and <i>strstream</i> is included in this header file.
fstream.h	It includes set of classes that support file I/O. The definition of various classes such as <i>ifstream</i> , <i>ofstream</i> and <i>fstream</i> is included in this header file.

4.3 Stream Classes

The stream can represent file, console, block of memory or hardware device. The *iostream* library provides the common set of functions for handling these streams. The general representation is as shown in following Fig. 4.3.1.

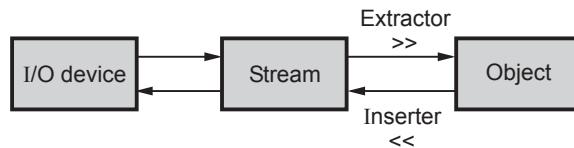


Fig. 4.3.1 Stream

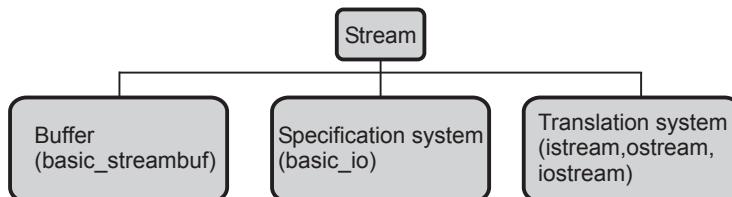


Fig. 4.3.2 Basic level of stream class

The stream class hierarchy is divided into three *areas* :

1. A buffer system given by **basic_streambuf** class. This class supports the basic low level input output operations. For advanced I/O programming the **basic_streambuf** class is used directly.
2. The specification system implemented by **basic_ios** class. This is high level I/O class that provides formatting, error checking. **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_oiostream** and **basic_iostream**. These classes are used to create streams capable of input, output and input/output, respectively.
3. A translation system implemented by certain classes like **istream**, **ostream**, **iostream**. This system converts the objects to a sequence of characters. (Refer Fig. 4.3.2)

The classes like basic_streambuf or basic_ios are known as template classes. From these template classes the 8-bit character based classes are derived -

Template class	8-bit character class
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

This can be presented by following Fig. 4.3.3 of class hierarchy.

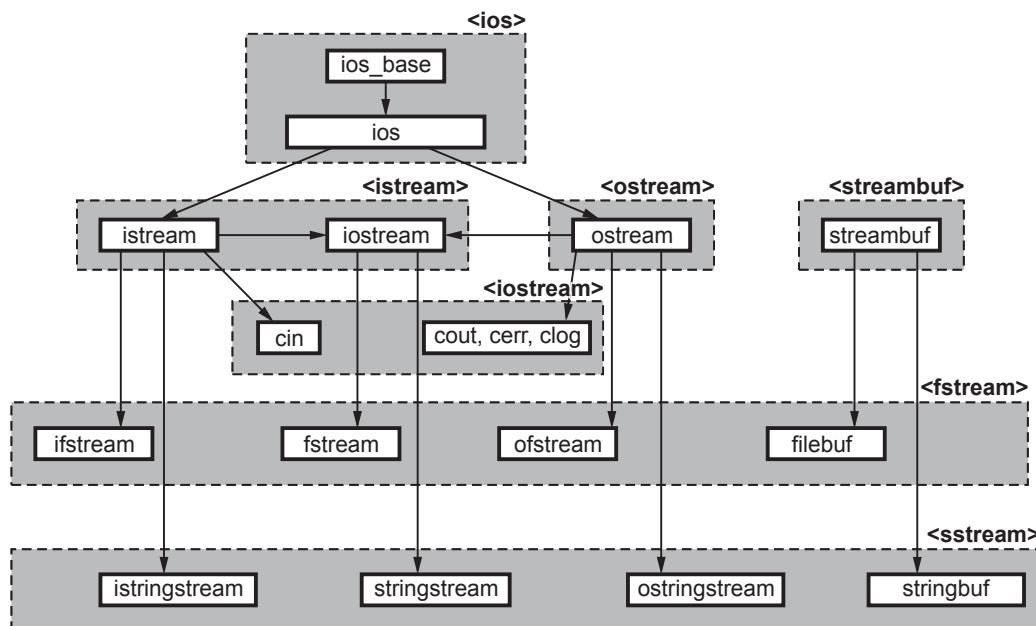


Fig. 4.3.3 Stream class hierarchy

Now onwards the character class names will be used because these names are normally used in the program.

4.4 Stream Errors

There are many abnormal conditions while handling the input and output. When errors occurred during input or output operations, the errors are reported by stream state.

Every stream (istream or ostream) has a **state** associated with it. Error conditions are detected and handled by testing the state of the stream.

The **ios** class defines some member functions using which the state of the stream can be checked.

Member Function	Purpose
bool ios::good()	This function returns true if there is no error
bool ios::bad()	This function returns true if no read/write operation is performed or invalid read/write operation is performed.
bool ios::eof()	This function returns true if the input operation is reached end of file
bool ios::fail()	This function returns true if the input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.
void ios::clear()	This function clears all the flag if no argument is supplied. It can also be used to clear a particular state flag if supplied as argument.

Following C++ program shows how to use the member functions for checking the state of the stream.

Example 1 : Following C++ program shows the state of the stream when wrong input is provided.

```
#include<iostream>
using namespace std;
int main()
{
    int val;
    char ans = 'y';
    cout << "\n Enter an integer value: ";
    cin >> val;
    if (cin.good() == 0)//if error
    {
        cout << "\n Incorrect input";
        cout << "\n The good() function returns: " << cin.good();
        cin.clear();
    }
    else
    {
```

```
    cout << "\n Correct input";
    cout << "\nThe good() function returns: " << cin.good();
}

return 0;
}
```

Output(Run1)

Enter an integer value: a

Incorrect input
The good() function returns: 0

Output(Run2)

Enter an integer value: 10

Correct input
The good() function returns: 1

Program Explanation : In above program, when the integer input is expected and if we provide character type data as input then the **good()** function will return 0 otherwise it will return 1.

Example 2 : Following C++ program illustrates the fail() function. In this program if wrong operation is performed on the file, then failbit is set.

```
#include<iostream>
#include <fstream>
using namespace std;

int main()
{
    char buffer[80];
    fstream myfile;

    myfile.open("d:\\test.txt", fstream::in);
    myfile << "This line is from test.txt\\0";
    if (myfile.fail())//Returns true as input operation is failed to write to file
    {
        cout << "Error writing to test.txt\\n";
        myfile.clear();
    }

    myfile.getline(buffer,80);
    cout << "\\n Reading from file ...\\n";
    cout << buffer;
```

```

cout << "\n Nothing is written to file because in open function mode is fstream::in
instead of fstream::out";

return 0;
}

```

Output

Error writing to test.txt

Reading from file ...

Nothing is written to file because in open function mode is fstream::in instead of
fstream::out

Program Explanation : In above program **myfile** is open for reading the data(fstream::in) from the file but we are trying to write data to the file. Further the **clear()** operation is performed to remove the flag and allow the further operation like reading from the file using **getline**.

Review Question

1. Write and explain purpose of various member functions used to handle stream errors.

4.5 Disk File I/O with Streams

C++ provides following classes to perform input and output of characters to and from the files.

ofstream	This stream class is used to write on files.
ifstream	This stream class is used to read from files.
fstream	This stream class is used for both read and write from/to files.

To perform file I/O, we need to include **<fstream.h>** in the program. It defines several classes, including **ifstream**, **ofstream** and **fstream**. These classes are derived from **ios**, so **ifstream**, **ofstream** and **fstream** have access to all operations defined by **ios**. While using file I/O we need to do following tasks -

1. To create an input stream, declare an object of type **ifstream**.
2. To create an output stream, declare an object of type **ofstream**.
3. To create an input/output stream, declare an object of type **fstream**.

Let us write a simple program.

```

#include <iostream.>
#include <fstream>
using namespace std;
int main();

```

```

{
    ofstream fobj; //creating object
    fobj.open ("output.txt"); //opening the file using object
    fobj << "Writing...Tested OK!!!\n"; //writing to the file
    fobj.close(); //closing the file
    return 0;
}

```

The above program basically creates a file called **output.txt** and writes the message "Writing...Tested OK!!!" in that file. At the end of the program the file is closed.

4.5.1 Open File Operation

The file operations are associated with the object of one of the classes : *ifstream*, *ofstream* or *fstream*. Hence we need to create an object of the corresponding class.

The file can be opened by the function called *open()*. The syntax of file open is

Open(filename,mode)

The *filename* is a null terminated string that represents the name of the file that is to be opened. The mode is optional parameter and is used with the flags as given below -

ios::in	Open for input operation.
ios::out	Open for output operation.
ios::binary	Open for binary operations.
ios::ate	If this flag is set then initial position is set at the end of the file otherwise initial position is at the beginning of the file.
ios::app	The output operations are appended to the file. This is an appending mode. That means contents are inserted at the end of the file.
ios::trunc	The contents of pre existing file get destroyed and it is replaced by new one.

We can use **open()** function using the above given syntax as -

```

ofstream obj;
obj.open("sample.bin",ios::out|ios::binary)

```

That means the file **sample.bin** is opened for output operation in binary mode. Thus we can combine the flags using OR operator (|).

The *is_open()* is a boolean function that can be used to check whether the file is open or not.

For example

```

if(obj.is_open())
{

```

```
cout<<"File is Successfully opened for operations";
}
```

4.5.2 Close File Operation

To close the file the member function *close()* is used. The *close* function takes no parameter and returns no value.

For example

```
obj.close();
```

You can detect when the end of an input file has been reached by using the *eof()* member function of *ios*. It returns true when the end of the file has been encountered and false otherwise.

Example 4.5.1 Write a C++ program to read the contents of a text file

Solution :

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
int main()
{
    ifstream in;
    char Data[80];
    in.open("Sample.dat");
    if (!in)
    {
        // Print an error and exit
        cerr << "Sample.dat could not be opened for reading!" << endl;
        exit(1);
    }
    cout << "The Contents of the file are..." << endl;
    while(in)
    {
        in.getline(Data,80);
        cout << "\n" << Data;
    }
    in.close();
    return 0;
}
```

Output

The Contents of the file are...

Statement 1
Statement 2

```
Statement 3
Statement 4
Statement 5
Statement 6
Statement 7
```

Note : The Sample.dat file is already created with the data as obtained in above output

Example 4.5.2 Write a C++ program that reads a file and counts the number of sentences, words and characters present in it.

Solution :

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
int main()
{
    char Data[80];
    int wc=0;
    int cc=0;
    int lc=0;
    ifstream in_obj;
    in_obj.open("Odd.dat");
    in_obj.read((char *)&Data,sizeof(Data));
    while(in_obj)
    {
        in_obj.getline(Data,80);
        int n=strlen(Data);
        cc+=n;
        lc++;
        for(int i=0;i<n;i++)
        {
            if(Data[i]==' ')
                wc++;
        }
    }
    in_obj.close();
    cout<<"\n Number of Sentences: "<<lc;
    cout<<"\n Number of Characters: "<<cc;
    cout<<"\n Number of Words: "<<wc;
    cout<<endl;
    return 0;
}
```

Example 4.5.3 Write a program that reads an array of number from file and creates another two files store the odd number in one file and even numbers in another file.

Solution :

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out_obj;
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    int b[10];
    out_obj.open("input.dat");
    out_obj.write((char *)&a,sizeof(a));
    out_obj.close();
    ifstream in_obj;
    ifstream fp1,fp2;
    in_obj.open("input.dat");
    in_obj.read((char *)&b,sizeof(b)); //storing the values file in b[]
    fp1.open("Even.dat");
    fp2.open("Odd.dat");
    for(int i=0;i<10;i++)
    {
        if((b[i]%2)==0)
            fp1<<b[i]<<" "; //writing to even file
        else
            fp2<<b[i]<<" "; //writing to odd file
    }
    in_obj.close();
    fp1.close();
    fp2.close();
    ifstream fp;
    char ch;
    fp.open("Even.dat");
    cout<<"\n The contents of even file are ..."<<endl;
    while(fp) //reading even file
    {
        fp.get(ch);
        cout<<ch;
    }
    fp.close();
    fp.open("Odd.dat");
    cout<<"\n The contents of odd file are ..."<<endl;
    while(fp)//reading odd file
    {
        fp.get(ch);
        cout<<ch;
    }
    fp.close();
    return 0;
}
```

```
}
```

Output

The contents of even file are ...

2 4 6 8 10

The contents of odd file are ...

1 3 5 7 9

4.5.3 Handling Multiple Files

We can open more than one files and can write the contents to them. In the following program we have created two different files namely : **emp.dat** and **dept.dat**. Through our C++ program we are writing some contents to them. Later on reading those contents and displaying them on the console. The program is

C++ Program

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out_obj;
    //creating the file for writing purpose
    out_obj.open("emp.dat");
    //writing the data to the file
    out_obj<<"Rahul\n";
    out_obj<<"Lekhana\n";
    out_obj<<"Nandan\n";
    out_obj<<"Archana\n";
    out_obj<<"Yogesh\n";
    //closing the file
    out_obj.close();
    out_obj.open("dept.dat");
    //writing the data to the file
    out_obj<<"Accounts\n";
    out_obj<<"Proof\n";
    out_obj<<"Marketing\n";
    out_obj<<"DTP\n";
    out_obj<<"Graphics Design\n";
    out_obj.close();
    //Reading from the files
    char Data[80];
    ifstream in_obj;
    in_obj.open("emp.dat");
    cout<<"\n Following are the contents of emp.dat file...\n";
```

```

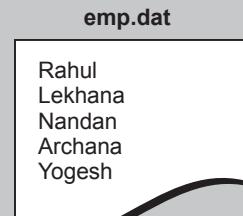
while(in_obj)
{
    in_obj.getline(Data,80);
    cout<<"\n"<<Data;
}
in_obj.close();
in_obj.open("dept.dat");
cout<<"\n Following are the contents of dept.dat file...\n";
while(in_obj)
{
    in_obj.getline(Data,80);
    cout<<"\n"<<Data;
}
in_obj.close();
return 0;
}

```

Output

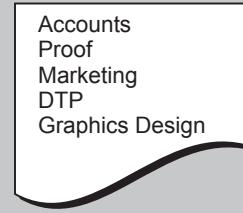
Following are the contents of emp.dat file...

Rahul
Lekhana
Nandan
Archana
Yogesh



Following are the contents of dept.dat file...

Accounts
Proof
Marketing
DTP
Graphics Design



Example 4.5.4 Write a program which copies the content of one file to a new file by removing unnecessary spaces between words.

Solution :

```

#include<iostream>
#include<fstream>
using namespace std;

int main()
{
    ifstream infile;
    ofstream outfile;
    infile.open("Sample.dat");

```

```

outfile.open("Myoutput.dat");
char output;
if(infile==NULL)
{
    cout<<"Error"<<endl;
    exit(0);
}
while(infile)
{
    infile.get(output);
    if(output!=' ')
    {
        outfile<<output;
    }
}
infile.close();
outfile.close();
ifstream fp;
fp.open("Myoutput.dat");
cout<<"\n Following are the contents of Myoutput.dat file...\n";
while(fp)
{
    fp.get(output);
    cout.put(output);
}
fp.close();

}

<Sample.dat>
Statement 1 Statement 2 Statement 3
return 0;

```

Output

Following are the contents of Myoutput.dat file...
Statement1Statement2Statement3

Example 4.5.5 Using file handling methods of C++ write a program and explain how to merge the contents of two files into one file.

Solution : Following is a simple C++ program in which we have created two files namely - emp.dat and dept.dat. The contents of both the files are read and merged into the third file named Combined.dat.

```

#include<iostream>
#include<fstream>
using namespace std;
int main()
{

```

```
//Reading from the files
char Data[80];
ifstream in_obj;
in_obj.open("emp.dat");
ofstream out_obj;
out_obj.open("Combined.dat");
while(in_obj)
{
    in_obj.getline(Data,80);
    out_obj<<Data;
    out_obj<<"\n";
}
cout<<"\n The contents of emp.dat file are written...\n";
in_obj.close();
in_obj.open("dept.dat");
cout<<"\n The contents of dept.dat file are written...\n";
while(in_obj)
{
    in_obj.getline(Data,80);
    out_obj<<Data;
    out_obj<<"\n";
}
in_obj.close();
out_obj.close();
ifstream fp;
fp.open("Combined.dat");
cout<<"\n Following are the contents of Combined.dat file...\n";
while(fp)
{
    fp.getline(Data,80);
    cout<<"\n"<<Data;
}
fp.close();
return 0;
}
```

Program Explanation :

In above program, initially the **emp.dat** file is opened in a read mode and the **Combined.dat** is opened in writing mode. The contents are read from the emp.dat and written in the **Combined.dat**. Similarly, the **dept.dat** file is opened in read mode and written in **Combined.dat**. All these files are then closed. Finally only **Combined.dat** file is opened in read mode and the merged contents are displayed on the console.

4.5.4 Handling Binary Files

We can make use of two functions namely: **read** and **write** for handling the binary file format.

The syntax of read and write functions will be -

```
input_obj.read((char *) &variable, sizeof(variable));
           _____
           |         |
           |         memory block
           |         _____
           |         |         |
           |         |         size of block
```

```
output_obj.write ((char*) & variable, size of (variable));
```

The memory block must be type cast to pointer to character type This block acts as a buffer in which read contents can stored or the data to be written in the file can be stored.

Following is a simple program in which we have some data in array **a**. We will be writing the contents of array **a** to the file "**input.dat**". Then we will read the file and retrieve the contents in another array **b**. Finally the array **b** will be displayed on the console.

C++ Program

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream out_obj;
    int a[5]={10,20,30,40,50};
    int b[5];
    out_obj.open("input.dat");
    out_obj.write((char *)&a,sizeof(a));
    out_obj.close();
    ifstream in_obj;
    in_obj.open("input.dat");
    in_obj.read((char *)&b,sizeof(b));
    cout<<"\n The contents of input.dat file are...\\n";
    for(int i=0;i<5;i++)
    {
        cout<<"\\n" << b[i];
    }
    in_obj.close();
    return 0;
}
```

Output

The contents of input.dat file are...

```
10  
20  
30  
40  
50
```

4.5.5 Finding the End of the File

For finding end of the file we use **eof()** function. This function is a member function of **ios** class. If end of file is encountered then it returns a non-zero value.

For example

```
if (seqfile. eof () != 0)  
{  
    cout << " You are at the end of the file";  
}
```

4.5.6 Reading and Writing Class Objects for Files

As C++ program is object oriented programming language, it is possible to write or read the class object to/from the file. Following example illustrates this concept

```
#include <iostream>  
#include <fstream>  
#include<string.h>  
using namespace std;  
class Employee  
{  
public:  
    char Name[40];  
    int ID;  
    double Salary;  
};  
void main()  
{  
    Employee emp1;  
    cout<<"Writing Data to the stream..."<<endl;  
    strcpy(emp1.Name, "Parth");  
    emp1.ID = 10;  
    emp1.Salary = 17000;  
    ofstream fp("Sample.dat", ios::binary);  
    fp.write((char *)&emp1,sizeof(emp1));  
    fp.close();  
    cout<<"Reading Data from the stream..."<<endl; // writing object to file  
    Employee emp2;  
    ifstream fpr("Sample.dat", ios::binary);
```

```
fpr.read((char *)&emp2,sizeof(emp2));
cout<<"Name: "<<emp2.Name<<endl;
cout<<"ID: "<<emp2.ID<<endl;
cout<<"Salary: "<<emp2.Salary<<endl;
fpr.close();

}
```

Output

Writing Data to the stream...
Reading Data from the stream...
Name: Parth
ID: 10
Salary: 17000

Review Question

1. Explain how a class object can be used to read to or write from the file.

4.6 Unformatted I/O Functions

1. Get and put functions

The get and put functions are used to read and display the contents. The syntax of get and put functions is

```
get (char ch);
put(char ch);
```

Following is a simple C++ program which reads the contents of the input file and displays them. It also displays the total number of characters in one line.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    int count=0;
    char c;
    ifstream input;
    cout<<"Reading the contents from the file"<<endl;
    input.open("Sample.dat");
    input.get(c);
    while(c!='\n')
    {
        cout.put(c);
        count++;
        input.get(c);
    }
}
```

```
    cout<<"\n Number of characters in the file = "<<count<<endl;
    return 0;
}
```

2. getline and write functions

The **getline** function is used to read the file line by line and the write statement is used to write the contents either to the file or to the console. Following C++ program reads the content from the file and displays each line. Finally the total number of lines count is also displayed.

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
int main()
{
    int count=0;
    char data[20];
    ifstream input;
    cout<<"Reading the contents from the file"<<endl;
    input.open("Sample.dat");
    input.getline(data,20,'n');
    int n=strlen(data);
    while(input)
    {
        cout.write(data,n);
        cout<<endl;
        count++;
        input.getline(data,20);
        n=strlen(data);
    }
    cout<<"\n Number of lines in the file = "<<count<<endl;
    return 0;
}
```

Output

Reading the contents from the file

Statement 1

Statement 2

Statement 3

Number of lines in the file = 3

4.7 Formatted I/O Functions and I/O Manipulators

In this section we will learn how to control the information flow. There is a set of format flags that control the way information is formatted. The *ios* class declares the values that are used to set or clear the format flags -

Following are the format flags along with their meaning.

Flags	Meaning
skipws	If this flag is set then leading white spaces are discarded (skipped). On clearing this flag the leading white spaces are not discarded.
left	On setting this flag the output becomes left justified.
right	On setting this flag the output becomes right justified.
internal	If it is set then numeric value is padded to fill the field between any sign or base character and number.
oct	It is set means numeric values are outputted in octal.
dec	It is set means numeric values are outputted in decimal.
hex	It is set means numeric values are outputted in hex.
showbase	If set then it shows base indicator(for example 0X for hex).
showpos	It shows + sign before positive numbers.
showpoint	On set it shows decimal point and trailing zeros for all floating point numbers.
scientific	On set it shows exponential format on floating point numbers.
fixed	On set it shows fixed format on floating point numbers.
boolalpha	Booleans can be set using <i>true</i> or <i>false</i> .
uppercase	On setting this flag upper case A-F are used for hex values and E for scientific values.

The **oct**, **dec** and **hex** fields are collectively referred as **basefield**. The **left**, **right** and **internal** fields can be referred as **adjustfield**, similarly **scientific** and **fixed** can be referred as **floatfield**.

The **setf()** function is used to set the flag and **unsetf()** function is used to clear the flag.

For example, if we want the text to be displayed on the console should be right justified then -

```
cout.setf(ios::right);
cout<<"Twinkle Twinkle Little Stars";
```

Following is a program that shows how several format flags can be used.

```
#include <iostream>
using namespace std;
int main()
{
    cout << 100.75 << " hello India " << 100 << "\n";
    cout << 10 << " " << -10 << "\n";
    cout << 100.0 << "\n\n";
    cout.unsetf(ios::dec);

    cout.setf(ios::hex|ios::scientific);
    cout << 100.75 << " hello India " << 100 << "\n";
    cout.setf(ios::showpos);
    cout << 10 << " " << -10 << "\n";
    cout.setf(ios::showpoint | ios::fixed);
    cout << 100.0;
    return 0;
}
```

Output

100.75 hello India 100

10 -10

100

1.0075e+02 hello India 64

a ffffff6

+100.000000

width(), precision() and fill()

There are three important member functions that define the fields like width, precision and fill characters.

By default the output value contains the number of characters as per the space occupied by it. But we can define the width of the output by using **width()** function.

For example

```
cout.width(10);
cout << "ABCD"
```

It will set minimum field width of 10. That means in a line total number of characters are 10. In above cout statement the text "ABCD" contains 4 characters only. Hence first 6 blank characters and then ABCD will be printed on the output screen. The output will be

ABCD

That means we get the right justified output.

Following program illustrates the use of **width**, **precision** and **fill** functions.

```
#include<iostream>
using namespace std;
int main()
{
    cout << "ABCD " << "\n";
    cout.width(10);
    cout << "ABCD " << "\n"; //sets right justified text
    cout.fill('*'); // set fill character
    cout.width(10); // set width
    cout << "ABCD" << "\n"; // right justified text with preceding *
    //hence the total number of characters are 10
    cout.setf(ios::left); // left justify
    cout.width(10); // set width
    cout << "ABCD" << "\n"; // output left justified
    cout.width(10); // set width
    cout.precision(10); //set 10 digits of precision
    cout << 10.203040 << "\n";
    cout.precision(4); // set 4 digits of precision
    cout << 102.0304050 << "\n";
    cout.width(10);
    cout.fill('*');
    cout << ""; //simply prints * 10 times
    return 0;
}
```

Output

```
ABCD
     ABCD
*****ABCD
ABCD*****
10.20304**
102.0304
*****
```

The program itself is self explanatory. Thus the width, precision and fill functions are used to format the output.

4.7.1 I/O Manipulators

Manipulators are the operators in C++ that are used for formatting the output. Various commonly used manipulators are **setw**, **endl** and **setfill**.

The **setw** is used to set the minimum field width. We can set the right justified numbers using **setw** operator. Note that the **setw** function is defined in **<iomanip.h>** file.

C++ Program

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    int a=4444,b=22,c=333;
    cout<<"Using setw(30)\n";
    cout<<setw(30)<<a<<"\n";
    cout<<setw(30)<<b<<"\n";
    cout<<setw(30)<<c<<"\n";
    cout<<"Using setw(20)\n";
    cout<<setw(20)<<a<<"\n";
    cout<<setw(20)<<b<<"\n";
    cout<<setw(20)<<c<<"\n";
    cout<<"Using setw(10)\n";
    cout<<setw(10)<<a<<"\n";
    cout<<setw(10)<<b<<"\n";
    cout<<setw(10)<<c<<"\n";
    return 0;
}
```

Output

Using setw(30)

```
4444
    22
    333
```

Using setw(20)

```
4444
    22
    333
```

Using setw(10)

```
4444
    22
    333
```

The functionality of **endl** is similar to "\n" i.e. newline character.

C++ Program

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    cout<<"This is first line"<<endl;
    cout<<"This is second line"<<endl;
    cout<<"This is third line"<<endl;
```

```

cout<<"This is forth line"<<endl;
return 0;
}

```

Output

This is first line
 This is second line
 This is third line
 This is forth line

The **setfill** manipulator is used to fill the fields entirely using some character specified within it.

C++ Program

```

#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
  int a=4444,b=22,c=333;
  cout<<setw(10)<<setfill('*')<<a<<endl;
  cout<<setw(10)<<setfill('*')<<b<<endl;
  cout<<setw(10)<<setfill('*')<<c<<endl;
  return 0;
}

```

Output

*****4444
 *****22
 *****333

Review Question

1. Explain I/O manipulators.

4.8 File Pointers

- The file pointers are used for locating the position in the file.
- With each file object there are two pointers associated with it. The **get pointer** and **put pointer**. These pointers basically return the current get position and current put positions.

While performing file operations, we must be able to reach at any desired position inside the file. For this purpose there are two commonly used functions -

1) seek

The seek operation is using two functions seekg and seekp.

- **seekg** means get pointer of specific location for reading of record.
- **seekp** means get pointer of specific location for writing of record.

The syntax of **seek** is

`seekg (offset, reference - position);`

`seekp (offset, reference - position);`

where, **offset** is any constant specifying the location.

reference - position is for specifying beginning, end or current position. It can be specified as,

<code>ios :: beg</code>	for beginning location
<code>ios :: end</code>	for end of file
<code>ios :: cur</code>	for current location

2) tell

This function tells us the current position.

For example

<code>seqfile. tellg ()</code>	-	gives current position of get pointer (for reading the record).
<code>seqfile. tellp ()</code>	-	gives current position of put pointer (for writing the record).

```
*****
Program for demonstrating random access to the file
*****
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream in;
    char Data[80];
    in.open("Sample.dat");
    if (!in)
    {
        // Print an error and exit
        cerr << "Sample.dat could not be opened for reading!" << endl;
        exit(1);
    }
    cout << "First statement of the file ..." << endl;
    in.seekg(0,ios::beg); // move to the beginning character
    // Get the rest of the line and print it
    in.getline(Data,80);
```

```
cout <<"\t" << Data << endl;
cout << "The current position is: " << in.tellg() << endl;
cout << "Moving to 3rd statement of the file ..." << endl;
in.seekg(13, ios::cur);
// Get rest of the line and print it
in.getline(Data,80);
cout << "\t" << Data << endl;

cout << "Moving to 6th statement of the file ..." << endl;
in.seekg(26, ios::cur);
// Get rest of the line and print it
in.getline(Data,80);
cout << "\t" << Data << endl;
cout << "Moving to last statement of the file ..." << endl;
in.seekg(-14, ios::end);
// Get rest of the line and print it
in.getline(Data,80);
cout << "\t" << Data << endl;
in.close();
return 0;
}
```

<Sample.dat>

Statement 1
Statement 2
Statement 3
Statement 4
Statement 5
Statement 6
Statement 7
Statement 8
Statement 9
Statement 10

Output

First statement of the file ...
Statement 1
The current position is: 13
Moving to 3rd statement of the file ...
Statement 3
Moving to 6th statement of the file ...
Statement 6
Moving to last statement of the file ...
Statement 10

Review Question

1. Explain the seekp and seekg function with illustrative example.

4.9 Error Handling in File I/O

When error occurs in file handling, flags are set in the state according to the general category of the error. Flags and their error categories are summarized in the following table.

state flag	Purpose
ios::goodbit	This state flag indicates that there is no error with streams. In this case the status variables has value 0.
ios::badbit	This state flag indicates that the stream is corrupted and no read/write operation can be performed.
ios::failbit	This state flag indicates that input/output operation failed.
ios::eofbit	This state flag indicates that the input operation reached end of input sequence.

When error occurs while performing file I/O operations, the appropriate message is displayed and then the file terminates.

Following program illustrates how to use appropriate error messages on corresponding read and write error-causing situations.

```
#include <iostream>
#include <fstream>
using namespace std;
#include <process.h>
const int MAX = 10;
int array1[MAX] = { 10,20,30,40,50 };
int array2[MAX];
int main()
{
    ofstream os; //create output stream
    os.open("d:\\test.dat", ios::trunc | ios::binary); //Opening file
    if (!os)
    {
        cerr << "Could not open output file\n"; //Error Handling
        exit(1);
    }
    cout << "Writing the contents to the file...\n\n";
    os.write((char*)&array1,sizeof(array1)); //writing 'array1' to file
    if (!os)
    {
        cerr << "Could not write to file\n"; //Error handling
        exit(1);
    }
    os.close(); //close the file
```

```

ifstream is; //create input stream for reading the contents from file
is.open("d:\\test.dat", ios::binary);
if (!is)
{
    cerr << "Could not open input file\\n"; //Error Handling
    exit(1);
}
cout << "Reading the contents from the file ...\\n";
is.read((char*)&array2,sizeof(array2)); //reading the contents in another array
                                         //array2
if (!is)
{
    cerr << "Could not read from file\\n"; //Error Handling
    exit(1);
}
for (int j = 0; j < MAX; j++) //check data
    cout << " " << array2[j];
return 0;
}

```

Output

Writing the contents to the file...

Reading the contents from the file ...

10 20 30 40 50 0 0 0 0 0

Program Explanation : In above program we have created one array of integers namely **array1**. Then in the **test.dat** file the contents of this array are written. Again the file is opened in read mode and contents of the file are read in another array namely **array2** and then those contents are displayed on the console.

During, open, read and write modes of the file, appropriate error handling is done. Using the **cerr** the error messages can be displayed.

4.10 File I/O with Member Functions

It is possible to perform file read and write operations using the member function of a class.

Following C++ program shows that the class named **Student** is created. The file read and write operations are performed using **ReadFile** and **WriteFile** member functions

```

#include <iostream>
#include <fstream>
using namespace std;
class Student
{
protected:

```

```
int rollNo;
char name[40];
float marks;

public:
    void input_Data(void)
    {
        cout << "\n Enter Roll Number: ";
        cin >> rollNo;
        cout << "\n Enter name: ";
        cin >> name;
        cout << "\n Enter marks: ";
        cin >> marks;
    }
    void display(void)
    {
        cout << "\n Roll No : " << rollNo;
        cout << "\n Name : " << name;
        cout << "\n Marks :" << marks;
        cout << "\n-----";
    }
    void ReadFile(int);
    void WriteFile();
    static int TotalRec();
};

void Student::ReadFile(int index)           // Member function for read file
{
    ifstream in_obj;
    in_obj.open("test.dat", ios::binary);
    in_obj.seekg(index*sizeof(Student));
    in_obj.read((char*)this, sizeof(*this)); //read one record at a time
}
void Student::WriteFile()                 // Member function for write file
{
    ofstream out_obj;

    out_obj.open("test.dat", ios::app | ios::binary);
    out_obj.write((char*)this, sizeof(*this)); //writes current record to file
}
int Student::TotalRec()
{
    ifstream in_obj;
    in_obj.open("test.dat", ios::binary);
    in_obj.seekg(0, ios::end);
    //by tellg getting count for total number of records
    int size= (int)in_obj.tellg() / sizeof(Student);
    return size;
```

```
}

int main()
{
    Student obj;
    char ans;
    cout << "\n\t Program for Student Database\n";
    do
    {
        cout << "Enter Student Record:";
        obj.input_Data();
        obj.writeFile(); //write to disk
        cout << "Do you want to enter more record?(y/n) ? ";
        cin >> ans;
    } while (ans == 'y');

    int n = obj.TotalRec(); //finds total number of records
    cout << "There are " << n << " records in file\n";
    for (int j = 0; j < n; j++) //Reading each record one by one
    {
        obj.ReadFile(j);
        obj.display();
    }
    return 0;
}
```

Output

```
Program for Student Database
Enter Student Record:
Enter Roll Number: 10
Enter name: AAA
Enter marks: 93.55
Do you want to enter more record?(y/n) ? y
Enter Student Record:
Enter Roll Number: 20
Enter name: BBB
Enter marks: 67.89
Do you want to enter more record?(y/n) ? y
Enter Student Record:
Enter Roll Number: 30
Enter name: CCC
Enter marks: 88.25
Do you want to enter more record?(y/n) ? n
There are 3 records in file

Roll No : 10
Name : AAA
Marks :93.55
```

```
-----
Roll No : 20
Name : BBB
Marks :67.89
-----
```

```
-----
Roll No : 30
Name : CCC
Marks :88.25
-----
```

Review Question

1. Explain how a member function of class can take part in file read and write operation with the help of necessary C++ code.

4.11 Overloading the Extraction and Insertion Operators

In C++ the output operation is called insertion and `<<` is called **insertion operator**. When `<<` is used for output the insertion function is created. That means output operator inserts information into the stream. The syntax of inserter function is -

```
ostream &operator<<(ostream &stream, class-name obj)
{
    // body of inserter

    return stream;
}
```

The **ostream** is a class and first parameter in above given insertion function is reference to the object of type **ostream** and the second parameter will be object that will be output. The inserter function returns the reference **stream**.

Hence we can write -

```
cout<<obj1<<obj2;
```

The following program illustrates the insertion function -

```
#include <iostream>
using namespace std;
class Point
{
public:
    int x, y; // must be public
    Point() //constructor
```

```

    {
        x = 0;
        y = 0;
    }
    Point(int i, int j)//constructor with parameter
    {
        x = i;
        y = j;
    }
};

//inserter function
ostream &operator<(ostream &st, Point obj)
{
    st << obj.x << ", " << obj.y << "\n"; //outputting the values
    return st; //returning stream
}

int main()
{
    Point a(10, 20);
    Point b(100, 200);
    cout << a << b;
    return 0;
}

```

Output

10, 20
100, 200

Thus we can overload `<<` operator. Similarly, we can overload `>>` operator which is input operator. In C++ the `>>` operator is called as **extractor operator**. The input operation is called extractor because the data is extracted from the stream.

The syntax for extractor function is -

The general form of an extractor function is :

```

istream &operator>>(istream &stream, class-name &obj)
{
    // body of extractor
    return stream ;
}

```

The first parameter is a reference to the class **istream**. The **istream** class is the input stream. The second parameter is a object for receiving input.

```
#include <iostream>
using namespace std;
class Point
{
public:
    int x, y; // must be public
    Point() //constructor
    {
        x = 0;
        y = 0;
    }
    Point(int i, int j)//constructor with parameter
    {
        x = i;
        y = j;
    }
};

// inserter function
ostream &operator<<(ostream &st, Point obj)
{
    st << obj.x << ", " << obj.y << "\n"; //outputting the values
    return st; //returning stream
}
// extractor function
istream &operator>>(istream &st, Point &obj)
{
    cout << "Enter x: ";
    st >> obj.x; //inputting the values
    cout << "Enter y: ";
    st >> obj.y;
    return st;
}

int main()
{
    Point a,b;
    clrscr();
    cin >>a >>b;
    cout<<"\n First Point ";
    cout << a;
    cout<<"\n Last Point ";
    cout << b;
    return 0;
}
```

Output

Enter x: 10
 Enter y: 20
 Enter x: 100
 Enter y: 200

First Point 10, 20
 Last Point 100, 200

4.12 Memory as a Stream Object

- In C++, it is possible to treat particular section of memory as an object and data can be inserted into this section of memory with the help of memory object.
- The memory as a stream object is normally used when the formatted output is required for a C++ program.
- The stream classes **ostrstream** is commonly used to create memory object
- The syntax for creating such memory object is

ostrstream ObjectName(buffer,size_of_buffer)

- Following is a simple program that shows how to use memory as a stream object.

C++ Program

```
// writes formatted data into memory
#include <strstream>
#include <iostream>
#include <iomanip> //for setw()
using namespace std;
const int SIZE = 20; //size of memory buffer
int main()
{
    int RollNo = 10;
    char name[] = "Rashmi";
    char Data[SIZE]; //Data Buffer for memory object
    ostrstream memObj(Data, SIZE); //create stream object by passing buffer and size
    memObj << "Roll Number = " << RollNo
        << setw(10)
        << "Name = " << name
        << '\0'; //end the buffer with '\0'
    cout << Data; //display the data buffer
    return 0;
}
```

Output

Roll Number = 10 Name = Rashmi

Program Explanation : In above program we have two variables - one integer variable(RollNo) and another string variable(name). The buffer is denoted by an array Data[].

The memory object named **memObj** is created using **ostrstream**

This Data along with its Size is passed as an argument to this object.

The insertion operator << is used to insert the formatted data into the object.

4.13 Command-Line Arguments

The command line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

The command line arguments are handled with the help of **main()** function. In that case the main function will look like this-

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

Here the **arc** represents the total number of arguments and **argv** is a an array of characters that store the command line arguments.

The **argv[0]** is the name of the program, or an empty string if the name is not available. After that, every element number less than **argc** is a command line argument. You can use each **argv** element just like a string.

Following C++ program shows the use of command line arguments.

C++ Program

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    cout << "\nTotal Number of arguments = " << argc;
    for (int i = 0; i<argc; i++) //display arguments
        cout << "\nArgument " << i << " = " << argv[i];
    return 0;
}
```

Output

```
D:\>test 10 20 30 40 50
Total Number of arguments = 6
Argument 0 = test
Argument 1 = 10
Argument 2 = 20
Argument 3 = 30
Argument 4 = 40
Argument 5 = 50
D:\>
```

Now, using this command line arguments, user can read the contents of the input file. Following application program illustrates this idea.

Step 1 : Create a input file and store some data in it. For instance I have created

Input.dat

```
a
b
c
d
```

Step 2 : Now create a C++ program, which will read the name of the input file(created in step 1) using command line argument.

test.cpp

```
#include <fstream>
#include <iostream>
using namespace std;
#include <process.h>
int main(int argc, char* argv[] )
{
    if( argc != 2 )
    {
        cerr << "\nFile Name is Missing!!!";
        exit(-1);
    }
    char ch;
    ifstream in_file;
    in_file.open( argv[1] );
```

```

cout<<"\n The contents of "<<argv[1]<<" are...\n";
while( in_file.get(ch) != 0 )
    cout << ch;
return 0;
}

```

Step 3 : Now compile the above code and open the command prompt and give the following command

Prompt:>test input.dat

The contents of input.dat are...

a
b
c
d

Review Question

1. Explain the command line arguments using C++ program.

4.14 Printer Output

The printer is connected to a port to your system. Following is a list of ports that are used for devices.

Name of Port	Device
con	Keyboard and Screen
aux	First serial port
com2	Second serial port
lpt1	First parallel printer
lp2	Second parallel printer
lpt3	Third parallel printer

In C++ it is possible to send data to the printer and print and get the formatted output.

C++ Program

```
#include <fstream> //for file streams
using namespace std;
```

```
int main()
{
    ofstream printer; //make a file
    printer.open("PRN"); //open it for printer
    printer << "Hello friends!!!" << endl; //send data to printer
    printer << '\x0C'; //formfeed to eject page
    printer.close();
    return 0;
}
```



Unit - V

5

Exception Handling and Templates

Syllabus

Exception Handling - Fundamentals, other error handling techniques, simple exception handling - Divide by Zero, Multiple catching, re-throwing an exception, exception specifications, user defined exceptions, processing unexpected exceptions, constructor, destructor and exception handling, exception and inheritance. **Templates** - The power of Templates, Function Template, Overloading Function templates and class template, class template and Nontype parameters, template and friends Generic Functions, The type name and export keywords.

Contents

- 5.1 Error Handling Techniques
- 5.2 Simple Exception Handling
- 5.3 Divide by Zero
- 5.4 Multiple Catching
- 5.5 Re-throwing an Exception
- 5.6 Exception Specifications
- 5.7 User Defined Exceptions
- 5.8 Processing Unexpected Exceptions
- 5.9 Constructor, Destructor and Exception Handling
- 5.10 Exception and Inheritance
- 5.11 Introduction to Templates
- 5.12 The Power of Templates
- 5.13 Function Template
- 5.14 Overloading Function Templates
- 5.15 Class Template
- 5.16 Template Arguments
- 5.17 More Programs in Templates
- 5.18 Class Template and Nontype Parameters
- 5.19 Template and Friends
- 5.20 Generic Functions
- 5.21 The Typename and Export Keywords

Part I : Exception Handling

5.1 Error Handling Techniques

In the initial stages of development of C++, there were not standard error handling mechanisms. Programmers used to write some C functions to indicate error situations. There were three approached that were used more often -

1. Terminate the program on error situation
2. Write the error code and assign it to global variable. On getting the error, return the corresponding code.
3. Some status code can be predefined and on success or failure the corresponding code can be returned.
4. Use of **goto** for jumping to desired statement.

Typically the functions like **abort()** or **exit()** are used for getting out of the error-prone situations.

But this created problem for large scale applications. Sometimes due to use of such functionalities, the memory for the objects does not get de-allocated properly. Sometimes the application program may get crashed abruptly. Hence the concept of exception handling has come up.

5.2 Simple Exception Handling

Definition : When any unavoidable circumstances (or runtime errors) occur in our program then exceptions are raised by handing control to special functions called **handlers**. This provides build-in error handling mechanism which is known as **exception handling**.

The purpose of exception handling mechanism is to detect and report the exceptional situations so that appropriate action can be taken. The exception handling mechanism performs following task -

1. Find the problem - Hit the exception
2. Report for the occurrence of the error - Throw the exception
3. Obtain the information about the error - Catch the exception
4. Perform some corrective actions - Handling of exception.

Difference between Error and Exception

Sr. No.	Error	Exception
1.	Errors are some abnormal or wrong situations that occur in the program.	Exceptions are some abnormal or wrong situations that occur in the program.
2.	Error cannot be handled.	Exception can be handled using exception handler.
3.	Error is uncoverable.	Exception is coverable.
4.	Program crashes or stops working when an error occurs.	Program reports user friendly message about the abnormal situation and exits gracefully.
5.	Error cannot be covered but it is fixed by the programmer.	The exception can be handled using try, catch and throw statements.
6.	Error is compile time error.	Exception is run time error.

5.2.1 try-catch-throw

C++ exception handling mechanism makes use of three keywords - **try**, **catch** and **throw**. The **try** represents the block of statements in which there are chances of occurring some exceptional conditions.

When exception detected it is thrown using the **throw** statement.

There exists a block of statements in which the exception thrown is handled appropriately. This block is called **catch block**.

Following figures illustrates the concept of exception handling mechanisms -

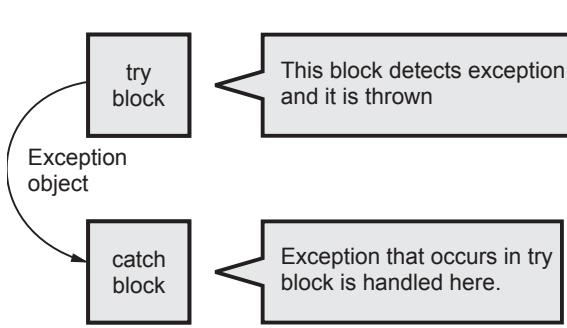


Fig. 5.2.1 (a) try-catch mechanism

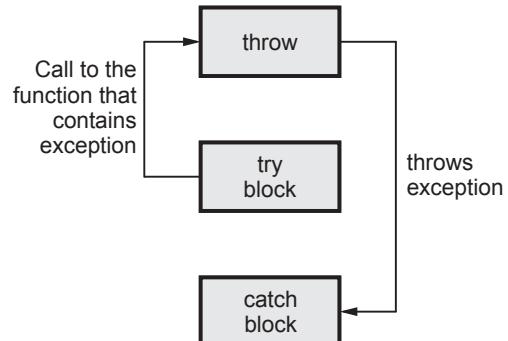


Fig. 5.2.1 (b) throw-try-catch mechanism

Mainly exception handling is done with the help of three keywords : *try*, *catch* and *throw*. The try and catch block is as shown below -

```
try
{
    throw exception;
    //exception is some value
    // the portion of the code that is to be monitored for error detection
}
catch( argument)
{
    //catch block softly handles the exception
}
```

The *try* block contains the portion of the program that is to be examined for error detection. If an exception (i.e. error) occurs in this block then it is thrown using *throw*. Using *catch* the exception is caught and processed. If *try* block contains all the code included in main then effectively the complete program must be scanned for errors. Any exception is caught by the catch block. This catch block should be immediately followed by the *try* block.

Let us understand it by a simple program of exception handling -

```
#include <iostream>
using namespace std;
int main ()
{
    try
    {
        throw 100;
    }
    catch (int x)
    {
        cout << "An exception at " << x << endl;
    }
    return 0;
}
```

Output

An exception at 100

In above program the *try* block contains the portion of the code for exception handling. In this program the exception is simply thrown by a *throw* statement. It accepts only one parameter 100 which is passed to exception handler as argument.

5.3 Divide by Zero

The purpose of exception handling is to handle abnormal events. The following program illustrates that in division operation the denominator value should not be zero and if at all it is zero then how to handle such error.

```
*****
Program to handle divide by zero error using exception handling mechanism
*****
#include<iostream>
using namespace std;

int main()
{
    double i, j;
    void divide(double, double);
    cout << "Enter numerator : ";
    cin >> i;
    cout << "Enter denominator : ";
    cin >> j;
    divide(i, j);
}
void divide(double a, double b)
{
    try
    {
        if (b == 0)
            throw b; // divide-by-zero
        cout << "Result: " << a / b << endl; // for non zero value
    }
    catch (double b)
    {
        cout << "Can't divide by zero.\n";
    }
}
```

Output

```
Enter numerator : 10
Enter denominator : 0
Can't divide by zero.
```

Program Explanation : In above program, if we enter the value of denominator as 0 the exception is raised which is handled by catch block. Inside the catch block the message "Can't divide by zero." Will be printed on the console and then the program terminates gracefully.

5.4 Multiple Catching

The exception handler is declared by the keyword *catch* which is responsible to handle the exception thrown by the *try* block. Normally the code within the catch statement attempts to rectify the errors by taking appropriate action. When an exception occurs then the control is transferred to the catch block and at that time *try* block is terminated. There can be multiple exceptions in **multiple catch** statements with one *try* block. Following is a structure of the program when multiple catch blocks are allowed.

```
void function
{
    ...
    try
    {
        ...
    }
    catch(datatype1 arg)
    {
        ...
    }
    catch(datatype2 arg)
    {
        ...
    }
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

The program illustrating multiple catch statements is as shown below -

```
#include<iostream>
using namespace std;
void function(int num)
{
    try
    {
        if(num)
            throw num;
        else throw "Value is zero";
    }
    //multiple catch for single try block
    catch(int i)
    {
```

```

cout << "Exception for number is handled: " << i << "\n";
}
catch(const char *str)
{
    cout << "Exception for string is handled: ";
    cout < str << "\n";
}
}
int main()
{
    cout << "Start" << endl;
    function(0); // As value of num is 'not true' else part will
    function(1); // be executed
    function(2);
    function(3);
    cout << "End" << endl;
    retrun 0;
}

```

Output

Start
Exception for string is handled: Value is zero
Exception for number is handled: 1
Exception for number is handled: 2
Exception for number is handled: 3
End

Example 5.4.1 Write a program to read 10 odd numbers. Generate the exceptions for negative and even numbers . Provide handlers to display appropriate message.

Solution :

```

#include<iostream>
#include<math.h>
using namespace std;
void MyFun(int a[10])
{
    for(int i=0;i<10;i++)
    {
        int val;
        val=a[i];
        try
        {
            if (val < 0.0)
                throw "Negative";
            else if ((val %2)== 0.0)
                throw val;
            else
                cout << "The number "<<val<<"is odd"<<endl;
        }
    }
}

```

```

        }
        catch(char *str)
        {
            cout<<"Error!! "<<str<<" Number"<<endl;
        }
        catch(int e)
        {
            cout<<"Error!! "<<e<<" is Even number"<<endl;
        }
    }
int main()
{
    int a[10];
    cout<<"\n Enter some number: ";
    for(int i=0;i<10;i++)
        cin>>a[i];
    MyFun(a);
    return 0;
}

```

Review Question

1. How to handle multiple exceptions occurred in a program ?

5.5 Re-throwing an Exception

When the exception is thrown inside the try block it is propagated to the catch block that immediately follows it. But sometimes handler may decide to rethrow it to propagate to next catch statement. In such a situation the exception can be rethrown by simply writing throw without any argument.

Following program shows the rethrowing of the exception.

```
*****
Program to demonstrate the rethrowing of an exception
*****
#include<iostream>
using namespace std;
void function()
{
    try
    {
        throw "myworld";
    }
    catch(char*)
    {

```

```

cout<<"\nInside the catch statement of function()";
throw; //rethrowing the exception
}

}

int main()
{
    try
    {
        function();
    }
    catch(char*)
    {
        cout<<"\n\nInside the catch statement of main()";
    }
    return 0;
}

```

The exception which is rethrown
is handled here by this catch
block.

Output

Inside the catch statement of function()
Inside the catch statement of main()

5.6 Exception Specifications

Exception specification is used to provide the information about the kind of exceptions that can be thrown. For example -

void f() throw(int,double)

this means that the function **f()** throws an exception of types integer or double. If it throws an exception with a different type(here other than int or double type), either directly or indirectly, it cannot be caught by a regular int or double type handler.

An exception specification with an empty throw, as in

void f() throw()

tells the compiler that the function does not throw any exceptions.

For example -

```
*****
Program for demonstrating exception specification
*****
#include <iostream.h>
void function() throw (int,double)
{
    throw(12.34);
}
```

```

void main()
{
    try
    {
        function();
    }
    catch(int i)
    {
        cout<<"Handling the integer value "<<i<<endl;
    }
    catch(double i)
    {
        cout<<"Handling the double value "<<i<<endl;
    }
    catch(...)
    {
        cout<<"Exception handling for something else"<<endl;
    }
}

```

Output

Handling the double value 12.34

Program Explanation : In above program the specification for the double or integer value is given. If we change the sentence as

throw("Hello");

then it will be handled by the third catch statement.

5.7 User Defined Exceptions

- User defined exception is a kind of exception defined by the user.
- As most of the exceptions that we need to handle in C++ are of class type.
- The object of class type is passed to exception handler (i.e. to catch routine) and with the help of object an error is processed.

Following is a simple program that illustrates the use of user defined exception.

```

#include<iostream>
#include<cstring>
using namespace std;
// Catching class type exceptions.
class My_Exception
{
public:
    char str[50];
    int num;
    My_Exception() { *str = 0; num = 0; } // constructor

```

```

My_Exception(char *s, int i)
{
    strcpy(str, s);
    num = i;
}
};

void main()
{
    int a;
    //testing if the number is positive or not
    try
    {
        cout << "Enter a positive number: ";
        cin >> a;
        if(a<0)
            throw My_Exception("It's a negative number", a);
        else
            cout <<"It's a positive number"<< endl;
    }
    catch (My_Exception obj)
    { // catch an error
        cout << obj.str << ": ";
        cout << obj.num << endl;
    }
}
}

```

Output 1

Enter a positive number: 10
It's a positive number

Output 2

Enter a positive number: -5
It's a negative number: -5

Program Explanation : For the above code, two outputs are given one when the number which we input is positive and another when the number is negative.

Example 5.7.1 Write a C++ program to compute the square root of a number. The input value must be tested for validity. If it is negative, the user defined function mysqrt() should raise exception.

Solution :

```
*****
Program to handle square root function using exception handling
*****/#include<iostream>
#include<math.h>
void MySqrt(double val)
{
```

```

try
{
if (val < 0.0)
    throw "Negative";
else
    cout << "The sqrt of "<<val<<" is "<<sqrt(val)<<endl;
}
catch(char *str)
{
    cout<<"Can not handle "<<str<<" number"<<endl;
}
}

int main()
{
    cout<<"\n Enter some number: ";
    double num;
    cin>>num;
    MySqrt(num);
    return 0;
}

```

Output

Enter some number: -5
Can not handle Negative number

Review Question

1. *What are user defined exceptions ? Explain with the help of illustrative examples.*

5.8 Processing Unexpected Exceptions

For processing the unexcepted exception, the **set_unexpected** function is used . This function is also called as **unexpected handler function**. The unexpected handler by default calls the **terminate** function. Following program shows the processing of unexcepted exception.

C++ Program

```

*****
Program to demonstrate the unexpected function
*****
#include<iostream>
using namespace std;
#include<exception> //Required for set_unexpected()
void myunexpected ()
{
    cerr<<"unexpected called\n";
}

```

```

        throw 0;
    }
void function() throw (int,double)
{
    throw("Hello");
}
void main()
{
    set_unexpected (myunexpected);//seting unexpected exception handler
    try
    {
        function();
    }
    catch(int i)
    {
        cout<<"Handling the integer value"<<i<<endl;
    }
    catch(double i)
    {
        cout<<"Handling the double value"<<i<<endl;
    }
    catch(...)
    {
        cout<<"Exception handling for something else"<<endl;
    }
}

```

Exception specification lists **int** and **double** values.

The string is not specified in the list hence unexcepted exception occurs.

Output

Exception handling for something else

Program Explanation : In above program, inside the main(), the function named **function()** is called inside a try block. The function **function()** has a exception specification that lists the exceptions for integer and double value. But in this function, the exception for **string** occurs. This is unexepcted exception which gets occurred inside **function()**. To handle this exception, the catch block other than integer and double values will be called. Hence is the output.

5.9 Constructor, Destructor and Exception Handling

The exception handling activity is carried out by try-catch block.

When the exception is raised inside a try block, for all the objects created inside the blocks the destructor is called first and then the catch block executes to handle the exception raised in try block. Following C++ program illustrates this idea

C++ Program

```
#include <iostream>
using namespace std;
class Test
{
public:
    Test() { cout << "\n Constructor is called"; }
    ~Test() { cout << "\n Destructor is called"; }
};
int main()
{
    try {
        Test obj1,obj2; //calling constructor for twice
        throw 100; //calling destructors twice and then catch block executes
    }
    catch (int e)
    {
        cout << "\nException is caught!!!" << e;
    }
    return 0;
}
```

Output

```
Constructor is called
Constructor is called
Destructor is called
Destructor is called
Exception is caught!!!100
```

5.10 Exception and Inheritance

The inheritance is a mechanism in which the derived class is derived from the base class.

It is possible to throw the object of derived class as exception, in that case the exception handler will check - who is the base class from which the derived class is derived. For example - consider following C++ program

```
#include <iostream>
using namespace std;
class B
{
};
class D :public B
{
```

```

int main()
{
    D d;
    try
    {
        throw d;
    }
    catch (B obj)
    {
        cout << "\n Base class Exception handling catch block";
    }
    catch (D obj)
    {
        cout << "\n Derived class Exception handling catch block";
    }

    return 0;
}

```

Output

Base class Exception handling catch block

Program Explanation : In above program, when the exception occurs for derived class D, then the exception handler does two things i) it will match the class with specific type ii) then it checks who is the base class for the derived class ?. Here the match of the derived class with its specific type is found(because any derived class is basically of type base class "is-a" relationship). Now the next thing which exception handler does is finding out the base class for the corresponding derived class. It is class B, and in sequence we get the catch block for base class B. In this situation, the **catch block for derived class will never be executed**. Hence we get the output.

Now if we change the order of catch block as follows -

```

#include <iostream>
using namespace std;
class B
{
};
class D :public B
{
};

int main()
{
    D d;
    try
    {
        throw d;
    }

```

```

    }
    catch (D obj)
    {
        cout << "\n Derived class Exception handling catch block";
    }
    catch (B obj)
    {
        cout << "\n Base class Exception handling catch block";
    }

    return 0;
}

```

Output

Derived class Exception handling catch block

Program Explanation : Here the exception handler first finds the match for the type for the derived class, in the first catch block itself the match for derived class is found. Hence is the output.

Review Question

1. Explain the exception handling mechanism in case of inheritance

Part II : Templates**5.11 Introduction to Templates**

The **generic programming** is a technique that allows to write the code for **any data type** elements. The template is used as a tool for generic programming.

```

int max(int left, int right)
{
    if(left<right)
        return right;
    else
        return left;
}

```

```

double max(double left,double right)
{
    if(left<right)
        return right;
    else
        return left;
}

```

The above two functions are nearly identical but the function on the left side is for comparing the two integer numbers and the function on the right hand side is for comparing the two double type numbers. Now we want the generic programming algorithm which will perform the functionality of both the data type elements. Hence we can replace int and double by a type T. Here is an example -

```
#include<iostream>
using namespace std;

template <class T> //T helps us to take any type of data
T max(T left,T right)
{
if(left<right)
    return right;
else
    return left;
}
int main()
{
int int1,int2;
cout<<"\n\t Comparison of Two integer numbers";
cout<<"\n Enter first integer number";
cin>>int1;
cout<<"\n Enter second integer number";
cin>>int2;
int max1=max(int1,int2);//passing two integer values
cout<<" The Maximum integer number is"<<max1<<endl;
double d1,d2;
cout<<"\n\t Comparison of Two double numbers";
cout<<"\n Enter first double number";
cin>>d1;
cout<<"\n Enter second double number";
cin>>d2;
double max2=max(d1,d2);// passing two double values
cout<<"The maximum double number is"<<max2<<endl;
return 0;
}
```

Output

Comparison of two integer numbers

Enter first integer number20

Enter second integer number10

The Maximum integer number is20

Comparison of two double numbers

Enter first double number6.2

Enter second double number77.99

The maximum double number is77.99

Templates allow the reusability of the code. There are two categories of templates -

1. Function template
2. Class template

Review Question

1. Explain the following - Generic Programming.

1

5.12 The Power of Templates

- Templates are useful for **code reusability**.
- The generic functions and generic classes are useful tool for creating the generalized code that can handle any datatype elements.
- The Standard Template Library(STL) is built upon the concept of templates, in which the commonly used library classes or functions are written as templates.
- It helps in generating high performance object code.

5.13 Function Template

To perform identical operations for each type of data compactly and conveniently, the function templates are used. One can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately. Thus the same functional code with different data type elements can be handled. For instance using add function one can perform addition of two integer values or two double values or two float values.

The syntax of function template is as follows -

```
template <class name_of_data_type>
name_of_data_type function_name (name_of_data_type id1,... name_of_data_type id2)
```

For example :

```
template<class T>
T min(T a, T b)
```

Here **template** is a keyword used to represent the template. Then inside the angular bracket keyword *class* is followed by the data type name T. The compiler will replace T by the appropriate data types.

Function templates are implemented like regular functions, except they are prefixed with the keyword **template**. Here is a sample with a function template.

C++ Program

```
#include <iostream>
using namespace std;
//min returns the minimum of the two elements
```

```

template <class T>
T min(T a, T b)
{
    if (a<b)
        return a;
    else
        return b;
}
int main()
{
    cout << "min(10, 20) = " << min(10, 20) << endl ;
    cout << "min('p', 't') = " << min('p', 't') << endl ;
    cout << "min(10.3, 67.2) = " << min(10.3, 67.2) << endl;
    return 0;
}

```

Output

```

min(10, 20) = 10
min('p', 't') = p
min(10.3, 67.2) = 10.3

```

Use of function templates is very easy : We can use them like regular functions. When the compiler sees an instantiation of the function template, for example : the call **min(10, 20)** in function main, the compiler generates a function **min(int, int)**. Similarly the compiler generates definitions for **min(char, char)** and **min(float, float)** on encountering the corresponding values.

Review Question

1. What is the concept of functional template ?

5.14 Overloading Function Templates

- It is possible to overload the function template by some non-template function or by template function. This non template function has the same name as the template function but it is not related to the template function.
- The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions.
- Following is an example of overloading the function template

```

#include<iostream>
using namespace std;
template <class T>

```

```

void display(T x)
{
    cout << x;
}
void display(int a)
{
    cout << "\n Inside the non template function";
}
int main()
{
    cout << "\n Displaying string: ";
    display("Hello");
    cout << "\n Displaying floating number: ";
    display(10.55);
    display(10); //calls overloaded non template function
    return 0;
}

```

Output

Displaying string: Hello
 Displaying floating number: 10.55
 Inside the non template function

5.14.1 Function Overloading vs Function Template

Function overloading allows the definition of more than one function with the same name and provides a means of choosing between the functions based on parameter matching. Template functions tell the compiler how to create new functions, based on the instantiation datatypes. Functions generated from the template (instantiations) behave like normal functions.

The function body will differ in function overloading whereas the function body will not differ in function template.

Review Question

- Explain the concept of function overloading. Also differentiate between overloading a function and overloading a function template.*

5.15 Class Template

Using class template we can write a class whose members use template parameters as types.

The syntax of class template declaration is

```
template < class Type >
class classname
```

```
{
...
    //body of class
...
};

}
```

In above example *Type* can be of any data type. Then template class member function is defined. The complete program using class template is as given below.

```
#include <iostream>
using namespace std;
template <class T>
class Compare { //writing the class as usual
    T a, b;//note we have used data type as T
public:
    Compare (T first, T second)
    {
        a=first;
        b=second;
    }
    T max ()//finds the maximum element among two
};
//template class member function definition
//here the member function of template class is max
template <class T>
T Compare <T> ::max ()
{
    T val;
    if(a>b)
        val=a;
    else
        val=b;
    return val;
}

int main ()
{
    Compare <int> obj1 (100, 60);//comparing two integers
    Compare <char> obj2('p','t');//comparing two characters
    cout << "\n maximum(100,60) = " <obj1.max();
    cout << "\n maximum('p','t') = " <obj2.max();
    retrun 0;
}
```

Output

```
maximum(100,60) = 100
maximum('p','t') = t
```

In above program, **Compare** is a class in which two variable a and b are declared for comparison. The function max is used to find the maximum number among the two. The T is used to indicate the data type. If we create an object of type integer by

```
Compare <int> obj1(100,60)
```

then two integer values will be compared. Similarly one can compare two characters, two real values by declaring appropriate objects.

Difference between function template and class template

Function templates are those functions which can handle different data types without separate code for each of them. For a similar operation on several kinds of data types, a programmer need not write different functions.

Using class template we can write a class whose members use template parameters as types.

5.16 Template Arguments

Templates can have multiple arguments. These arguments are normally denoted by the data type **T**. But along with **T** we can also define other data type arguments. For example we can define the template function as

```
Compare (T first, int second); //First argument is of type T and second is int
```

Following program illustrates this concept -

```
#include <iostream>
using namespace std;
template <class T>
class Compare
{ //writing the class as usual
    T a, b;//note we have used data type as T
public:
    Compare (T first, int second);
        T max ()//finds the maximum element among two
    };
//template class member function definition
//here the member function is constructor
template <class T>
Compare<T>::Compare(T first, int second)
{
    a=first;
    b=second;
}
    //here the member function of template class is max
template <class T>
```

```

T Compare <T>::max ()
{
    T val;
    if(a>b)
        val=a;
    else
        val=b;
    return val;
}
int main ()
{
    Compare <int> obj1 (100, 160); //comparing two integers
    cout << "\n maximum(100,160) = " << obj1.max();
    return 0;
}

```

5.17 More Programs in Templates

Example 5.17.1 Write a function template for finding the minimum value contained in array.

Solution :

```

#include<iostream.h>
using namespace std;

template <class T>
T min(T a[10])
{
    T min;
    min=a[0];
    for(int i=0;i<5;i++)
    {
        if(a[i]<min)
        {
            min=a[i];
        }
    }
    return min;
}

int main()
{
    int a[10];
    int i;
    for(i=0;i<5;i++)
    {
        cout << "Enter the integer values " << endl;
        cin >> a[i];
    }
}

```

```

}

cout<<"\n The minimum element of an array is: "<<min(a)<<endl;
float b[10];
for( i=0;i<5;i++)
{
    cout<<"Enter the real values "<<endl;
    cin>>b[i];
}
cout<<"\n The minimum element of an array is: "<<min(b)<<endl;
return 0;
}

```

Output

Enter the integer values
10
Enter the integer values
4
Enter the integer values
11
Enter the integer values
5
Enter the integer values
12

The minimum element of an array is: 4

Enter the real values
10.10
Enter the real values
5.5
Enter the real values
11.11
Enter the real values
6.6
Enter the real values
7.7

The minimum element of an array is : 5.5.

Example 5.17.2 Write a C++ program using class template for finding the scalar product for int type vector and float type vector.

Solution :

```
#include<iostream.h>
using namespace std;
template <class T>
class Scalar
{
    T a1,a2,a3,b1,b2,b3;
```

```
T ans;
public:
    void GetData()
    {
        cout<<"\n Enter the value of a bar: "<<endl;
        cout<<"\n Enter a1_i: ";
        cin>>a1;
        cout<<"\n Enter a2_j: ";
        cin>>a2;
        cout<<"\n Enter a3_k: ";
        cin>>a3;
        cout<<"\n Enter the value of b bar: "<<endl;
        cout<<"\n Enter b1_i: ";
        cin>>b1;
        cout<<"\n Enter b2_j: ";
        cin>>b2;
        cout<<"\n Enter b3_k: ";
        cin>>b3;
    }
    T Mul()
    {
        T A,B,C;
        cout<<"\n The scalar product is: "<<endl;
        A=a1*b1;
        B=a2*b2;
        C=a3*b3;
        ans=A+B+C;
        return ans;
    }
};

int main()
{
    Scalar <int> obj1;
    Scalar <float> obj2;
    cout<<"\n Enter int type values"<<endl;
    obj1.GetData();
    cout<<obj1.Mul()<<endl;
    cout<<"\n Enter float type values"<<endl;
    obj2.GetData();
    cout<<obj2.Mul()<<endl;
    return 0;
}
```

Output

Enter int type values

Enter the value of a bar:

```
Enter a1_i: 1
Enter a2_j: 2

Enter a3_k: 3

Enter the value of b bar:

Enter b1_i: 4

Enter b2_j: 5

Enter b3_k: 6

The scalar product is:
32

Enter float type values

Enter the value of a bar:

Enter a1_i: 1.1

Enter a2_j: 2.2

Enter a3_k: 3.3

Enter the value of b bar:
Enter b1_i: 4.4

Enter b2_j: 5.5

Enter b3_k: 6.6
The scalar product is:
38.72
```

Example 5.17.3 Develop a program in C++ to implement linear search of an array (Array can be of any type).

Solution :

```
*****
Program to implement Linear search using the class template
*****
#include<iostream.h>
#include<cstring>
using namespace std;
template <class T>
class LinSearch
```

```
{  
    T Array[10];  
    T key;  
public:  
    LinSearch(T k){ key=k;}  
    void Get();  
    T Find();  
};  
template <class T>  
void LinSearch<T>::Get()  
{  
    int i;  
    for(i=0;i<=5;i++)  
    {  
        cout<<"\n Enter the element: ";  
        cin>>Array[i];  
    }  
}  
template <class T>  
T LinSearch<T>::Find()  
{  
    Get();  
    for(int i=0;i<=5;i++)  
        if(Array[i]==key)  
            return key;  
    return -1;  
}  
void main()  
{  
    LinSearch <int> obj1(10);  
    LinSearch <double> obj2(10.10);  
    cout<<"\n\t Handling the integer data..."<<endl;  
    if(obj1.Find()>=0)  
        cout<<"The element is present in the list"<<endl;  
    else  
        cout<<"The element is not present"<<endl;  
    cout<<"\n\t Handling the double data...";  
    if(obj2.Find()>=0)  
        cout<<"The element is present in the list"<<endl;  
    else  
        cout<<"The element is not present"<<endl;  
}
```

Output

Handling the integer data...
Enter the element: 12

Enter the element: 11

```
Enter the element: 10
Enter the element: 3
Enter the element: 33
Enter the element: 5
The element is present in the list
Handling the double data...
Enter the element: 1.1
Enter the element: 2.2
Enter the element: 10.10
Enter the element: 4.5
Enter the element: 32.23
Enter the element: 7
The element is present in the list.
```

5.18 Class Template and Nontype Parameters

Definition : A non-type template argument provided within a template argument list is an expression whose value can be

- integral,
 - enumeration,
 - pointer,
 - reference,
 - or pointer to member type,
 - and must be constant at compile time.
- Floating point values are not allowed as template parameters.
 - Objects of class, struct or union type are not allowed as non-type template parameters, although pointers to such objects are allowed.
 - Arrays passed as non-type template parameters are converted into pointers.
 - Functions passed as non-type parameters are treated as function pointers.
 - String literals are not allowed as template parameters.
 - Nontype template arguments are normally used to initialize a class or to specify the sizes of class members.

- The Non-type template parameters provide the ability to pass a constant expression at compile time. For example -

```
#include<iostream>
using namespace std;
template<int i> class C //integral expression
{
public:
    int k;
    C()
    {
        k = i;
    }
    void display() { cout << "\n " << k; }

};

int main()
{
    C<100> x;//initialization of class with non-type template argument
    C<200> y;
    x.display();
    y.display();
    return 0;
}
```

Output

```
100
200
```

5.19 Template and Friends

A friend function can be used within a template class. The layout for such template class and friend function is as follows -

```
template <class T>
class className
{
    ..body of class with private and public members

//declaration of friend function in template class
template <class T> friend void f(Object_of_class);
};

template <class T>
friend void f(Object_of_class)
{
    //body of friend function
}

Following C++ program illustrates how to use friend function inside the template class.
```

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#include<iostream>
using namespace std;

template <class T>
class Test
{
    template <class T> friend void f(Test<T>&); //declaration of friend function
                                                //in template class

private:
    T value;
public:
    T get() { return value; }
    void set(T v) { value = v; }

};

template<class T>
void f(Test<T>& obj)
{
    cout << "Inside Friend Function!!!";
    cout << "\n\t You have entered..."<<obj.get();
}

int main()
{
    Test<int> object;
    int num;
    cout << "\n Enter the value to set ";
    cin >> num;
    object.set(num);
    f(object);//calling friend function
    return 0;
}
```

Output

```
Enter the value to set 100
Inside Friend Function!!!
    You have entered...100
```

5.20 Generic Functions

- Generic functions are the set of functions that define the general set of operations for various data types.
- Due to general function, single procedure can be applied for computing various data type elements. For example - generic function **sum** can be applied for addition of two integers or two floating point numbers.

- Normally generic function is created using **template**. Hence generic function is also called as template function.
- When compiler creates a specific version of this function, then it is called **specialization**. This is called **generated function**. The generated function is a specific instance of template function.
- The generic function uses the keyword **template** which should directly precede the function definition.
- For example -

```
#include<iostream>
using namespace std;
template <class T>
void display(T x)
{
    cout << x;
}
int main()
{
    cout << "\n Displaying string: ";
    display("Hello");
    cout << "\n Displaying floating number: ";
    display(10.55);
    cout << "\n Displaying integer: ";
    display(10);
    return 0;
}
```

Output

```
Displaying string: Hello
Displaying floating number: 10.55
Displaying integer: 10
```

5.20.1 Restrictions on Generic Functions

The generic functions are similar to overloaded functions. But generic functions are restrictive in nature. Because the overloaded function can perform different actions but the generic function has same general action for all the data members. Here only data type varies.

Following program illustrates this restriction

```
#include<iostream>
using namespace std;
template <class T>
int display(int x,int y)
{
```

```

        return x+y;
}
double display(double x,double y)
{
    return x*y;
}
int main()
{
    cout<<display(10,20);
    cout<<"\n"<<display(10.55,20.20);
    return 0;
}

```

Now both the display functions are performing altogether different tasks, hence it is not possible to make it as general function.

5.20.2 Applying Generic Functions

The generic function is applied when a general common task need to be performed for different types of elements. For example, searching a number from the list of elements, or for sorting the numbers. Here numbers can be integer numbers or floating point numbers.

Following is a generic function that is applied to searching a number from the list. The list can be of integer numbers or it can be of double(i.e. floating point numbers). Simple linear search technique is used.

```

#include<iostream>
using namespace std;
template <class T>
void display(T *list,T key, int count)
{
    int flag=0;
    for (int i = 0; i < count; i++)
    {
        if (key == list[i])
            flag = 1;

    }
    if (flag == 1)
        cout << "\n The element is present in the list";
    else
        cout << "\n The element is not present in the list";
}
int main()
{
    int a[5] = { 33,12,78,100,55 };

```

```

cout << "\n Searching element 100 from the list";
display(a, 100, 5);
double b[7] = { 33.33,12.12,78.78,100.10,55.55,66.66,91.67};
cout << "\n Searching element 78.78 from the list";
display(b,78.78,7);
return 0;
}

```

Output

Searching element 100 from the list
The element is present in the list
Searching element 78.78 from the list
The element is present in the list

The above program shows that the logic of the function is independent of the type of the data. Hence it can be made as a generic function.

5.21 The Typename and Export Keywords

The keyword **typename** can be used in place of **class** in template parameter list. It can be used in a template declaration and definition.

template<class type1, class type2>

can be replaced by

template<typename type1,typename type2>

For example

```

#include<iostream>
using namespace std;

#include <iostream>
using namespace std;

template <typename T>
class Test {
    T a, b;
public:
    Test (T first, T second)
        {a=first; b=second;}
    T getmax ();
};

template <typename T>
T Test<T>::getmax ()
{
    T retval;
    if (a > b)

```

```

        retval = a;
else
    retval = b;
return retval;
}

int main () {
Test <int> myobject (100, 55);
cout << myobject.getmax();
return 0;
}

```

Output

100

Uses of keyword typename

1. It is used to replace the keyword **class** used in template specification.
2. The use of **typename** is to inform the compiler that a name used in a template declaration is a type name rather than an object name.

The **export** keyword can be used to export the template definitions to other files. The keyword **export** is preceded the **template** keyword in the template definition.

For example -

```

export template <typename T>
void display(T *list, T key, int count);

```

can be placed in some header file say **test.h**. This test.h can then be included in the main application program which uses the templates.

For example

//test.h

```

export template <typename T>
void display(T *list, T key, int count);

```

//test.cpp

```

#include "d:\test.h"
template <typename T>
void display(T *list, T key, int count)
{
    int flag = 0;
    for (int i = 0; i < count; i++)
    {
        if (key == list[i])
            flag = 1;

    }
    if (flag == 1)

```

```
        cout << "\n The element is present in the list";
    else
        cout << "\n The element is not present in the list",
}
//main.cpp

#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include "D:\test.cpp"
using namespace std;

int main()
{
    int a[5] = { 33,12,78,100,55 };
    cout << "\n Searching element 100 from the list";
    display(a, 100, 5);
    double b[7] = { 33.33,12.12,78.78,100.10,55.55,66.66,91.67 };
    cout << "\n Searching element 78.78 from the list";
    display(b, 78.78, 7);
    return 0;
}
```



Notes

Unit - VI

6

Standard Template Library (STL)

Syllabus

Introduction to STL, STL Components, Containers - Sequence container and associative containers, container adapters, Application of Container classes : vector, list,

Algorithms - basic searching and sorting algorithms, min-max algorithm, set operations, heap sort,

Iterators - input, output, forward, bidirectional and random access. Object Oriented Programming - a road map to future.

Contents

- 6.1 Introduction to STL
- 6.2 STL Components
- 6.3 Sequence Container
- 6.4 Associative Container
- 6.5 Container Adapter
- 6.6 Algorithms
- 6.7 Set Operations
- 6.8 Heap Sort
- 6.9 Iterators- Input, Output, Forward, Bidirectional and Random Access
- 6.10 Object Oriented Programming - A Road Map to Future

6.1 Introduction to STL

The standard template library(STL) is collection of well structured generic C++ classes (templates) and functions. Basically STL consists three basic components -

1. Container
2. Algorithms
3. Iterators

The standard template library is built using template and it is orthogonal in design because components can be used in combination with one another. Let us discuss these components in detail.

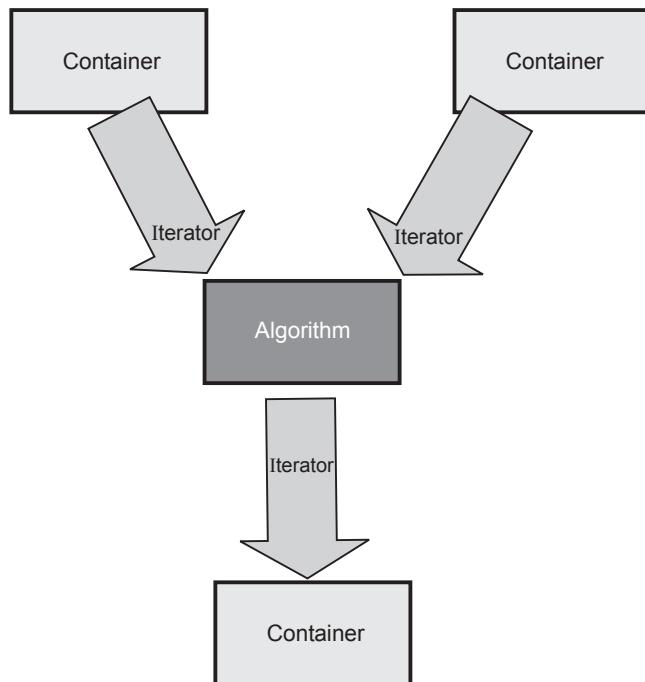


Fig. 6.1.1 Components of standard template library

6.2 STL Components

Container : The container is a collection of objects of different types. These objects store the data. The container can be implemented using template classes. There are two types of containers

Sequence container

Associative container

Iterator : The iterators are basically objects but sometimes they can be pointers and hence iterators specify the positions in container. The iterators are used to traverse the contents of container.

Algorithm : The algorithms are used to process the contents of the containers. The functionalities provided in container are not sufficient to perform complex operations hence the algorithms are used to support more complex operations for the containers.

6.3 Sequence Container

- It consists of all the classes who represent the sequence or linear list.
- For example, **vector** class defines dynamic array. **Deque** is for creating doubly ended queue i.e. we can perform insertion and deletion of elements from both the ends of queue and list provides a linear list. The **list** class defines the collection of elements in which only sequential access to the elements is allowed.
- Let us discuss the implementation of various types of sequence containers.

6.3.1 Vectors

Vector is a most general purpose container. It stores the elements in contiguous memory locations. Any element of vector can be accessed directly using index of it given by subscript operator []. It supports the dynamic array. The dynamic array is an array which can grow or shrink dynamically. The memory allocation for vector is done at the run time.

The vector can be declared as

```
vector <int> v; //It creates a zero length vector
vector<char> v(5); //creates the vector with 5 element character.
vector <double> v2(v1)//creates v2 vector from v1 vector of double type.
vector <char> v(4,'a')//initializes 4 element char vector
```

In the program we have to include header file `<vector>` in the program. The program for vector implementation is as illustrated below -

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10); // create a vector of length 10
    int i;

    // display size of vector
    cout << "The size of vector = " << v.size() << endl;

    //store English alphabets to vector
    for(i=0; i<10; i++)
        v[i] = i + 'A';

    // display contents of vector
```

```

cout << "Elements in the vector are:\n";
for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
cout << "\n\n";

cout << "Inserting more elements to vector..." << endl;
for(i=0; i<5; i++)
    v.push_back(i + 10 + 'A');//vector grows

// display size of vector
cout << "New size of vector = " << v.size() << endl;

// display contents of vector
cout << "Elements in vector are:\n";
for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
cout << "\n\n";

//deleting last five elements of the list
cout << "\n Now deleting the 5 elements from end of the vector..." << endl;
for(i=0;i<5;i++)
    v.pop_back();//vector shrinks
cout << "\n";

//Retaining the original list
cout << "Elements in vector are:\n";
for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
cout << "\n\n";

cout << "The size is now = " << v.size();
cout << "\n\n";
return 0;
}

```

Output

The size of vector = 10
 Elements in the vector are:
 A B C D E F G H I J

Inserting more elements to vector...
 New size of vector = 15
 Elements in vector are:
 A B C D E F G H I J K L M N O

Now deleting the 5 elements from end of the vector...

Elements in vector are:
A B C D E F G H I J

The size is now = 10

Program explanation

In above program we have declared the char vector of length 10 as

```
vector<char> v(10);
```

The vector function *v.size()* is used to obtain the size of the vector. Then we inserted some characters in the vector by *v.push_back* function. By this function we can insert the element at the end of the vector, i.e. we have inserted 5 elements after 'A'+10 elements. In other words we have inserted (K, L, M, N and O after J). Again by using *v.size()* we can obtain the modified size of vector.

Using *v.pop_back()* we have deleted the last five elements and retain the original vector.

Thus the above program of vector is for expanding and shrinking the vector.

Some commonly used vector functions are -

Function	Description
begin()	Returns the first element of the vector.
end()	Returns the last element of the vector.
size()	Returns the size of the vector.
erase()	Erases the given elements.
push_back()	Insert the element in the vector at the end.
pop_back()	Deletes the last element of the vector.
resize()	Modifies the original size of the vector.

6.3.2 Deque

The doubly ended queue(Deque) is a type of data structure in which the element can be inserted from both the front as well as rear end. Similarly the element can be deleted from the front as well as rear end.

Various functions used for performing deque operations are as given in the following table -

Function	Purpose
void push_back(value)	The element can be inserted in the deque using rear end.
void push_front(value)	The element can be inserted in the deque using front end.
void pop_back()	The element can be deleted from the rear end.
void pop_front()	The element can be deleted from the front end.
front()	Pointer is at front element.
back()	Pointer is at back element.
Size_type size()	Returns the total number of elements in deque.

There is no direct function available for displaying the contents of the deque. Hence using object of **iterator** the element of the deque can be displayed.

The syntax for creating deque is

```
deque<object_type> deque_name
```

Following C++ program illustrates various operations on deque -

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
    int item;
    char ans = 'y';
    int choice;
    deque<int> dq;
    deque<int>::iterator i;
    do
    {
        cout << "\n Program for Implementing DEQUE using Sequence Container";
        cout << "\n Main Menu";
        cout << "\n1. Insert from Rear";
        cout << "\n2. Insert from Front";
        cout << "\n3. Delete from Front";
        cout << "\n4. Delete from Rear";
        cout << "\n5. Get the Size of DEQUE";
        cout << "\n6. Display";
        cout << "\n Enter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\n Enter the element to be inserted: ";
            break;
        }
    } while (ans == 'y');
}
```

```

        cin >> item;
        dq.push_back(item);
        break;
    case 2:cout << "\n Enter the element to be inserted: ";
        cin >> item;
        dq.push_front(item);
        break;
    case 3:item = dq.front();
        dq.pop_front();
        cout << "\n The deleted element is: " << item;
        break;
    case 4:item = dq.back();
        dq.pop_back();
        cout << "\n The deleted element is: " << item;
        break;
    case 5:cout << "\n The size of DEQUEU is: " << dq.size();
        break;
    case 6:cout << "Elements of DEQUE are: ";
        for (i = dq.begin(); i != dq.end(); i++)//i is for iterator
            cout << *i << " ";
        }
        cout << "\n Do you want to continue?(y/n): ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');
}

```

Output

Program for Implementing DEQUE using Sequence Container

Main Menu

1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display

Enter your choice: 1

Enter the element to be inserted: 10

Do you want to continue?(y/n): y

Program for Implementing DEQUE using Sequence Container

Main Menu

1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display

Enter your choice: 1

Enter the element to be inserted: 20

```
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 2
Enter the element to be inserted: 30
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 2
Enter the element to be inserted: 40
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 6
Elements of DEQUE are: 40 30 10 20
Do you want to continue?(y/n): y

Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 3
The deleted element is: 40
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
```

```
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 5
The size of DEQUEU is: 3
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 4
The deleted element is: 20
Do you want to continue?(y/n): y
Program for Implementing DEQUE using Sequence Container
Main Menu
1. Insert from Rear
2. Insert from Front
3. Delete from Front
4. Delete from Rear
5. Get the Size of DEQUE
6. Display
Enter your choice: 5
The size of DEQUEU is: 2
Do you want to continue?(y/n):
```

6.3.3 List

List is a collection of elements in which only sequential access to the elements is allowed. This is basically a bidirectional linear list in which we can traverse the elements from left to right and from right to left. As bidirectional traversing is possible the insertion and deletion of elements is efficient.

The header file `<list>` needs to be included in the program.

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> lst; // create an empty list
```

```
int i,n,item;

//storing the elements in the list
cout << "\n How many elements you want to insert?" << endl;
cin >> n;
cout << "\n Enter the Elements in the List" << endl;
for(i=0; i<n; i++)
{
    cin >> item;
    lst.push_back(item);
}
cout << "The size of list is = " << lst.size() << endl;

//displaying the contents of the list
cout << "The contents of the List are: " << endl;
list<int>::iterator ptr = lst.begin();
while(ptr != lst.end())
{
    cout << *ptr << " "; //accessing the contents through iterator
    ptr++;
}
//sorting the contents of the list
lst.sort();

//displaying the sorted list
cout << "\n\n Sorted elements of the List are: " << endl;
ptr = lst.begin();
while(ptr != lst.end())
{
    cout << *ptr << " ";
    ptr++;
}
// Modifying the List
ptr = lst.begin();
while(ptr != lst.end())
{
    *ptr = *ptr + 1000;
    ptr++;
}
//displaying the modified list using iterator
cout << "\nModified elements of the list are: " << endl;
ptr = lst.begin();
while(ptr != lst.end())
{
    cout << *ptr << " ";
    ptr++;
}
```

```
cout<<"\n\n";
return 0
}
```

Output

How many elements you want to insert?

5

Enter the Elements in the list

30

10

20

40

50

The size of list is = 5

The contents of the List are:

30 10 20 40 50

Sorted elements of the list are:

10 20 30 40 50

Modified elements of the list are:

1010 1020 1030 1040 1050

Program Explanation

In above given program, initially the empty list is created by

```
list<int> lst;
```

Then using the *push_back()* function we have inserted the elements in the list

```
for(i=0; i<n; i++)
{
    cin>>item;
    lst.push_back(item);
}
```

One iterator is declared and initialized and using which we are trying to access the contents of the list as follows -

```
list<int>::iterator ptr = lst.begin();
```

The iterator is given by pointer *ptr*. The following code is for accessing the contents of the list.

```
while(ptr != lst.end())
{
    cout << *ptr << " ";
    ptr++;
}
```

The list can be sorted simply by invoking `lst.sort()` function.

Various commonly used functions of list are given in following table -

Function	Description
<code>begin()</code>	Returns the first element of the list.
<code>end()</code>	Returns the last element of the list.
<code>size()</code>	Returns the size of the list.
<code>pop_back()</code>	Removes last element of the list.
<code>pop_front()</code>	Removes the first element of the list.
<code>push_back()</code>	Inserts the specified value at the end of the list.
<code>push_front()</code>	Inserts the specified value at the beginning of the list.
<code>reverse()</code>	Reverses the list.
<code>sort()</code>	Sort the contents of the list.

6.4 Associative Container

The associative container allows efficient retrieval of values based on keys. The key can be usually an integer or string value using which the data can be arranged in ascending or descending order.

There are various types of associative containers - set, multiset, map and multimap.

Let us discuss the implementation of these containers.

6.4.1 Set

- Set is a container that contains **unique elements** in some specific order. We can perform various operations on set such as insertion or deletion of element, searching particular element from the set, displaying of size of the set and so on.
- Following table shows various functions that can be performed on the set -

Function	Purpose
<code>void insert(value)</code>	The element is inserted in the set.
<code>void erase(value)</code>	The element is deleted from the set.
<code>Return_type find(value)</code>	Returns the location of the element to be searched from the set.
<code>Size_type size()</code>	Returns the total number of elements in set.

- The syntax for creating set is

```
set<object_type> set_name
```

- The header file <set> need to be included in the program while using the above mentioned operations.
- Following C++ program illustrates various operations on set -

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    int item,count;
    char ans = 'y';
    int choice;
    set<int> s;
    set<int>::iterator i;
    do
    {
        cout << "\n Program for Implementing SET using Associative Container";
        cout << "\n Main Menu";
        cout << "\n1. Insert an element";
        cout << "\n2. Delete an element";
        cout << "\n3. Get the Size of SET";
        cout << "\n4. Search an element";
        cout << "\n5. Display";
        cout << "\n Enter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\n Enter the element to be inserted: ";
                      cin >> item;
                      s.insert(item);
                      break;
            case 2:cout << "\n Enter the element to be deleted: ";
                      cin >> item;
                      s.erase(item);
                      break;
            case 3:count = s.size();
                      cout << "\n The size of set is: " << count;
                      break;
            case 4:i = s.find(item);
                      if (i != s.end())
                          cout << "Element " << *i << " is present in the set" << endl;
                      else
                          cout << "Element is not present in the set" << endl;
                      break;
        }
    } while (ans == 'y');
}
```

```
case 5:cout << "Elements of SET are: ";
    for (i = s.begin(); i != s.end(); i++)//i is for iterator
        cout << *i << " ";
    }
    cout << "\n Do you want to continue?(y/n): ";
    cin >> ans;
} while (ans == 'y' || ans == 'Y');
}
```

Output

Program for Implementing SET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display

Enter your choice: 1

Enter the element to be inserted: 10

Do you want to continue?(y/n): y

Program for Implementing SET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display

Enter your choice: 1

Enter the element to be inserted: 20

Do you want to continue?(y/n): y

Program for Implementing SET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display

Enter your choice: 1

Enter the element to be inserted: 30

Do you want to continue?(y/n): y

Program for Implementing SET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display

Enter your choice: 5

```

Elements of SET are: 10 20 30
Do you want to continue?(y/n): y
Program for Implementing SET using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display
Enter your choice: 2
Enter the element to be deleted: 20
Do you want to continue?(y/n): y
Program for Implementing SET using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Display
Enter your choice: 5
Elements of SET are: 10 30
Do you want to continue?(y/n):

```

6.4.2 Multi Set

The difference between set and multi-set is the set does not allow duplicating elements but multiset allows duplicating elements.

The syntax for declaring multiset is

```
multiset <object_type> multiset_name
```

Operations using iterator

Function	Purpose
ms.begin()	Returns a bidirectional iterator for the first element.
ms.end()	Returns a bidirectional iterator for the position after the last element.
ms.rbegin()	Returns a reverse iterator for the first element of a reverse iteration
ms.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration

Inserting and Removing Elements of Multisets

Operation	Effect
ms.insert(value)	Inserts some value and returns the position of the new element.
ms.insert(pos, element)	Inserts a copy of element and returns the position of the new element.
ms.insert(beg,end)	Inserts a copy of all elements of the range [beg, end].
ms.erase(element)	Removes all elements with value element and returns the number of removed elements.
ms.erase(pos)	Removes the element at iterator position.
ms.erase(beg,end)	Removes all elements of the range [beg, end].
ms.clear()	Removes all elements.

The implementation of multiset is just similar to set.

Following C++ program shows the implementation of multiset -

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    int item, count;
    char ans = 'y';
    int choice,num;
    multiset<int> ms;
    multiset<int>::iterator i;
    do
    {
        cout << "\n Program for Implementing MULTISET using Associative Container";
        cout << "\n Main Menu";
        cout << "\n1. Insert an element";
        cout << "\n2. Delete an element";
        cout << "\n3. Get the Size of SET";
        cout << "\n4. Search an element";
        cout << "\n5. Count the number of occurrences";
        cout << "\n6. Display";
        cout << "\nEnter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\nEnter the element to be inserted: ";
            cin >> item;
```

```

        ms.insert(item);
        break;
    case 2:cout << "\n Enter the element to be deleted: ";
        cin >> item;
        ms.erase(item);
        break;
    case 3:count = ms.size();
        cout << "\n The size of multiset is: " << count;
        break;
    case 4:cout << "\n Enter the element to be searched: ";
        cin >> item;
        i = ms.find(item);
        if (i != ms.end())
            cout << "Element " << *i << " is present in the multiset\n";
        else
            cout << "Element is not present in the multiset" << endl;
        break;
    case 5:cout << "\n Enter the element for counting its occurrences ";
        cin >> item;
        num = ms.count(item);
        cout << " The " << item << "is present for " << num << "times";
        break;
    case 6:cout << "Elements of MULTISET are: ";
        for (i = ms.begin(); i != ms.end(); i++)//i is for iterator
            cout << *i << " ";
    }
    cout << "\n Do you want to continue?(y/n): ";
    cin >> ans;
} while (ans == 'y' || ans == 'Y');
}

```

Output

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Count the number of occurrences
6. Display

Enter your choice: 1

Enter the element to be inserted: 10

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET

4. Search an element

5. Count the number of occurrences

6. Display

Enter your choice: 1

Enter the element to be inserted: 20

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element

2. Delete an element

3. Get the Size of SET

4. Search an element

5. Count the number of occurrences

6. Display

Enter your choice: 1

Enter the element to be inserted: 30

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element

2. Delete an element

3. Get the Size of SET

4. Search an element

5. Count the number of occurrences

6. Display

Enter your choice: 1

Enter the element to be inserted: 20

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element

2. Delete an element

3. Get the Size of SET

4. Search an element

5. Count the number of occurrences

6. Display

Enter your choice: 1

Enter the element to be inserted: 20

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Count the number of occurrences
6. Display

Enter your choice: 1

Enter the element to be inserted: 20

Do you want to continue?(y/n): y

Program for Implementing MULTISET using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of SET
4. Search an element
5. Count the number of occurrences
6. Display

Enter your choice: 5

Enter the element for counting its occurrences 20

The 20 is present for 4 times

Do you want to continue?(y/n):n

6.4.3 Map

- The map is an associative container in which the elements are stored in the form of **key value** and **mapped value**. For example - {(1, a), (2, b), (3, c)}
- In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ.
- The duplicate values are not allowed in map. Hence it possess one to one relationship.
- The header file **<map>** is included in the program for creating and using the map object.
- The key value and the mapped value are grouped together in member type **value_type** which is a pair type as

```
typedef pair<const Key, T> value_type;
```

- The syntax for creating map is,

```
map<key,value> map_name
```

- Various operations that can be performed on map are -

Function name	Purpose
insert(pair)	The element is inserted into the map.
erase(iterator i)	The element pointed by the iterator is deleted from the map.
void swap(map)	Swaps the contents of the map.
void clear()	Clears the contents.
size_type size()	Returns the size of the container.

- The C++ program is as follows -

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    int count,key;
    char ans = 'y',item;
    int choice;
    map<int,char> m;
    map<int,char>::iterator i;
    do
    {
        cout << "\n Program for Implementing MAP using Associative Container";
        cout << "\n Main Menu";
        cout << "\n1. Insert an element";
        cout << "\n2. Delete an element";
        cout << "\n3. Get the Size of MAP";
        cout << "\n4. Search an element";
        cout << "\n5. Display";
        cout << "\n Enter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\n Enter the element the key: ";
            cin >> key;
            cout << "\n Enter the value to be inserted: ";
            cin >> item;
            m.insert(pair<int,char>(key,item));
            break;
            case 2:cout << "\n Enter the key to be deleted: ";
        }
    } while (ans == 'y');
}
```

```

        cin >> key;
        m.erase(key);
        break;
    case 3:count = m.size();
        cout << "\n The size of set is: " << count;
        break;
    case 4:cout << "\n Enter the Key at for searching the element: ";
        cin >> key;
        if(m.count(key)!=0) // first represents key and second represents value
            cout << "Element " << m.find(key)->second << " is present" << endl;
        else
            cout << "Element is not present" << endl;
        break;
    case 5:cout << "Elements of MAP are: ";
        for (i = m.begin(); i != m.end(); i++)//i is for iterator
            cout << "[" << (*i).first << ", " << (*i).second << "] ";
    }
    cout << "\n Do you want to continue?(y/n): ";
    cin >> ans;
} while (ans == 'y' || ans == 'Y');
}

```

Output

Program for Implementing MAP using Associative Container
Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display

Enter your choice: 1

Enter the element the key: 1

Enter the value to be inserted: a

Do you want to continue?(y/n): y

Program for Implementing MAP using Associative Container
Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display

Enter your choice: 1

Enter the element the key: 2

Enter the value to be inserted: b

Do you want to continue?(y/n): y

Program for Implementing MAP using Associative Container
Main Menu

1. Insert an element

```
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 1
Enter the element the key: 3
Enter the value to be inserted: c
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 1
Enter the element the key: 4
Enter the value to be inserted: d
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 5
Elements of MAP are: [1, a] [2, b] [3, c] [4, d]
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 2
Enter the key to be deleted: 3
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 5
```

```
Elements of MAP are: [1, a] [2, b] [4, d]
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 3
The size of set is: 3
Do you want to continue?(y/n): y
Program for Implementing MAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MAP
4. Search an element
5. Display
Enter your choice: 4
Enter the Key at for searching the element: 2
Element b is present
Do you want to continue?(y/n): n
```

6.4.4 Multimap

- It is associative container which is used for fast retrieval of values using the keys. Thus similar to maps multimap also contain values in the form of key-value pair.
- The difference between map and multimap is that multimap allows the duplicate values. That means multiple values can be associated with a single key.
- Thus multimap possess the one to many relationship.
- The header file <map> is used for using multimap objects in your C++ program.
- The implementation of map and multimap is very much similar. But while finding the element with unique key and duplicate values the function **equal_range** is used.
- The syntax of this function is as follows -

```
pair<iterator, iterator> equal_range(const KeyType &key)
```

- This function returns a pair of iterators representing the range of elements in the multimap with the specified key. This function is ideal for iterating over all elements with the same key in a for loop.

The C++ code for the implementation of multimap is as follows -

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    int count, key;
    char ans = 'y', item;
    int choice;
    multimap<int, char> mmp;
    multimap<int, char>::iterator i;
    do
    {
        cout << "\n Program for Implementing MULTIMAP using Associative Container";
        cout << "\n Main Menu";
        cout << "\n1. Insert an element";
        cout << "\n2. Delete an element";
        cout << "\n3. Get the Size of MULTIMAP";
        cout << "\n4. Search an element";
        cout << "\n5. Display";
        cout << "\nEnter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\nEnter the element the key: ";
                      cin >> key;
                      cout << "\nEnter the value to be inserted: ";
                      cin >> item;
                      mmp.insert(pair<int, char>(key, item));
                      break;
            case 2:cout << "\nEnter the key to be deleted: ";
                      cin >> key;
                      mmp.erase(key);
                      break;
            case 3:count = mmp.size();
                      cout << "\nThe size of set is: " << count;
                      break;
            case 4:cout << "\nEnter the Key at for searching the element: ";
                      cin >> key;
                      multimap<int, char>::iterator k;//used for finding duplicate values
                      for (k = mmp.equal_range(key).first; k !
                           = mmp.equal_range(key).second; ++k)
                          cout << ' ' << (*k).second;
                      break;
            case 5:cout << "Elements of MULTIMAP are: ";
                      for (i = mmp.begin(); i != mmp.end(); i++)//i is for iterator
                          cout << "[" << (*i).first << ", " << (*i).second << "] ";
        }
    } while (ans != 'n');
}
```

```
    }
    cout << "\n Do you want to continue?(y/n): ";
    cin >> ans;
} while (ans == 'y' || ans == 'Y');
}
```

Output

Program for Implementing MULTIMAP using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display

Enter your choice: 1

Enter the element the key: 10

Enter the value to be inserted: a

Do you want to continue?(y/n): y

Program for Implementing MULTIMAP using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display

Enter your choice: 1

Enter the element the key: 20

Enter the value to be inserted: b

Do you want to continue?(y/n): y

Program for Implementing MULTIMAP using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display

Enter your choice: 1

Enter the element the key: 20

Enter the value to be inserted: c

Do you want to continue?(y/n): y

Program for Implementing MULTIMAP using Associative Container

Main Menu

1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display

Enter your choice: 1

```
Enter the element the key: 30
Enter the value to be inserted: d
Do you want to continue?(y/n): y
Program for Implementing MULTIMAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display
Enter your choice: 5
Elements of MULTIMAP are: [10, a] [20, b] [20, c] [30, d]
Do you want to continue?(y/n): y

Program for Implementing MULTIMAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display
Enter your choice: 4

Enter the Key at for searching the element: 20
b c
Do you want to continue?(y/n): y

Program for Implementing MULTIMAP using Associative Container
Main Menu
1. Insert an element
2. Delete an element
3. Get the Size of MULTIMAP
4. Search an element
5. Display
Enter your choice: 3
The size of set is: 4
Do you want to continue?(y/n):n
```

6.5 Container Adapter

- There are some classes that are derived from the sequence containers. These are sometimes known as **derived container or container adapters**.
- Examples of them are - stack, queue, and priority queue.
- Let us discuss the implementation one container adapter - stack

6.5.1 Stack

The stack is a **LIFO** data structure. That means the element inserted at the last gets popped off first. The basic operations that can be implemented on stack are -

- Push
- Pop

Before popping the element it is necessary to check whether the stack is empty or not. Hence **stack** empty operation is also an important operation.

Stack is a derived container class which is basically derived from the sequence container **deque**.

Following is a list of operations that are supported by the stack derived container class.

Function	Meaning
push(item)	This function helps to push an item onto the stack.
pop()	This function pops the topmost element from the stack.
size()	This function returns the size of the stack.
top()	This function returns value which is at the top of the stack.

For using these functions in the program we must include the header file `<stack>` at the top. Following is a simple C++ program in which the derived container class **stack** is used. Various operations are also performed on this stack using the inbuilt functions of container class library.

```
#include<iostream>
#include<stack> //header file stack must be included
using namespace std;
int main()
{
    stack<int> s; //object is created for stack class
//Note that this class handles the int values
    int item;
    char ans;
    int choice;
    do
    {
        cout<<"\n Main Menu";
        cout<<"\n 1.Push ";
        cout<<"\n 2.Pop ";
        cout<<"\n 3.Display ";
```

```

cout<<"\n Enter your choice ";
cin>>choice;
switch(choice)
{

    case 1: cout<<"\n Enter the element to be pushed ";
    cin>>item;
    s.push(item);//pushing the element onto the stack
    cout<<"\n Item is pushed!!!";
    break;
    case 2: if(!s.empty())//checking stack is empty or not
    {
        s.pop();//popping the element from the stack
        cout<<"\n Popped an item!!!";
    }
    else
        cout<<"\n stack empty!!!";
    break;
    case 3:if(!s.empty())
    {
        cout<<"Top element of the stack is "<<s.top();
    }
    else
        cout<<"\n The stack is empty!!!";
    break;
}
cout<<"\n Do you want to continue? ";
cin>>ans;
}while(ans=='y');
return 0;
}

```

Displaying the
stack top element



Output

Main Menu
 1.Push
 2.Pop
 3.Display
 Enter your choice 1
 Enter the element to be pushed 10

Item is pushed!!!
 Do you want to continue? y

Main Menu
 1.Push
 2.Pop
 3.Display

Enter your choice 1

Enter the element to be pushed 20

Item is pushed!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 1

Enter the element to be pushed 30

Item is pushed!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 3

Top element of the stack is 30

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 2

Popped an item!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 3

Top element of the stack is 20

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 2

Popped an item!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 3

Top element of the stack is 10

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 2

Popped an item!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

Enter your choice 3

The stack is empty!!!

Do you want to continue? y

Main Menu

1.Push

2.Pop

3.Display

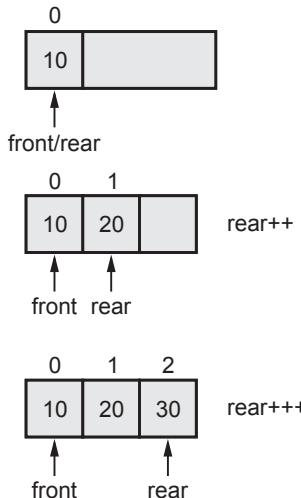
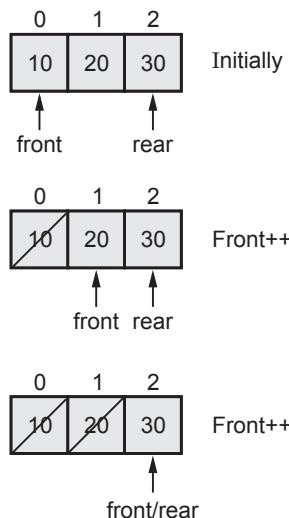
Enter your choice 2

stack empty!!!

Do you want to continue? n

6.5.2 Queue

The **queue** is a data structure in which the insertion of element is from the rear end and deletion of the element is front end.

Inserting 10, 20, 30 in a queue**Deleting 10, 20 from the queue**

The queue can be implemented using STL. The header file `<queue>` need to be inserted for this purpose.

Following C++ program shows the implementation of queue using STL

```
#include<iostream>
#include<queue> //header file queue must be included
using namespace std;
int main()
{
    queue<int> q; //object is created for stack class
```

```
//Note that this class handles the int values
int item;
char ans;
int choice;
do
{
    cout << "\n Main Menu";
    cout << "\n 1.Insert element ";
    cout << "\n 2.Delete element ";
    cout << "\n 3.Display Rear element";
    cout << "\n 4.Display Front element";
    cout << "\n 5.Size of queue";
    cout << "\n Enter your choice ";
    cin >> choice;
    switch (choice)
    {

        case 1: cout << "\n Enter the element to be inserted: ";
            cin >> item;
            q.push(item);//inserting the element onto the queue
            cout << "\n Item is inserted!!!";
            break;
        case 2: item = q.front();
            q.pop();//Deleting element from queue
            cout << "\n Deleted item is" << item;
            break;
        case 3:cout << "The element at rear end of queue is " << q.back();
            break;
        case 4:cout << "The element at front end of queue is " << q.front();
            break;
        case 5:cout << "\n Size of queue = " << q.size();
            break;
    }
    cout << "\n Do you want to continue? ";
    cin >> ans;
} while (ans == 'y');
return 0;
}
```

Output

Main Menu
1.Insert element
2.Delete element
3.Display Rear element
4.Display Front element
5.Size of queue
Enter your choice 1

Enter the element to be inserted : 10

Item is inserted!!!

Do you want to continue ? y

Main Menu

- 1.Insert element
- 2.Delete element
- 3.Display Rear element
- 4.Display Front element
- 5.Size of queue

Enter your choice 1

Enter the element to be inserted : 20

Item is inserted!!!

Do you want to continue ? y

Main Menu

- 1.Insert element
- 2.Delete element
- 3.Display Rear element
- 4.Display Front element
- 5.Size of queue

Enter your choice 1

Enter the element to be inserted : 30

Item is inserted!!!

Do you want to continue ? y

Main Menu

- 1.Insert element
- 2.Delete element
- 3.Display Rear element
- 4.Display Front element
- 5.Size of queue

Enter your choice 1

Enter the element to be inserted : 40

Item is inserted!!!

Do you want to continue ? y

Main Menu

- 1.Insert element
- 2.Delete element

```
3.Display Rear element  
4.Display Front element  
5.Size of queue  
Enter your choice 5
```

```
Size of queue = 4  
Do you want to continue ? y
```

```
Main Menu  
1.Insert element  
2.Delete element  
3.Display Rear element  
4.Display Front element  
5.Size of queue  
Enter your choice 2
```

```
Deleted item is10  
Do you want to continue ? y
```

```
Main Menu  
1.Insert element  
2.Delete element  
3.Display Rear element  
4.Display Front element  
5.Size of queue  
Enter your choice 3  
The element at rear end of queue is 40  
Do you want to continue ? y
```

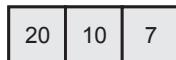
```
Main Menu  
1.Insert element  
2.Delete element  
3.Display Rear element  
4.Display Front element  
5.Size of queue  
Enter your choice 4  
The element at front end of queue is 20  
Do you want to continue ?n
```

6.5.3 Priority Queues

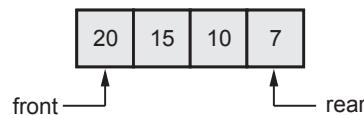
A priority queue is just like a normal queue data structure except that each element inserted is associated with a "priority". It supports the usual push(), pop(), top() etc operations, but is specifically designed so that its first element is always the greatest of the elements it contains, according to some strict weak ordering condition.

Priority queue is a queue in which priority is associated with every element. The first element (to be deleted first) is always greatest element.

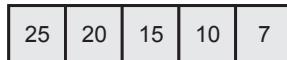
For example - Consider following priority queue



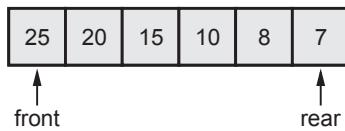
Insert 15



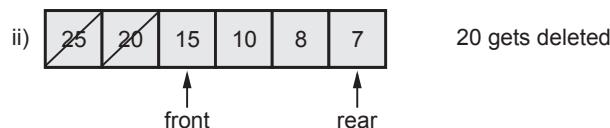
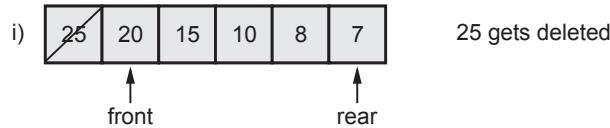
Insert 25



Insert 8



While deleting



```
#include<iostream>
#include<queue> //header file queue must be included
using namespace std;
int main()
{
    priority_queue<int> pq;
    int item;
```

```
char ans;
int choice;
do
{
    cout << "\n Main Menu";
    cout << "\n 1.Insert element ";
    cout << "\n 2.Delete element ";
    cout << "\n 3.Size of queue";
    cout << "\n 4.Display of queue";
    cout << "\n Enter your choice ";
    cin >> choice;
    switch (choice)
    {

        case 1: cout << "\n Enter the element to be inserted: ";
            cin >> item;
            pq.push(item);//inserting the element onto the queue
            cout << "\n Item is inserted!!!!";
            break;
        case 2: item = pq.top();
            pq.pop();//Deleting element from queue
            cout << "\n Deleted item is" << item;
            break;
        case 3:cout << "\n Size of queue = " << pq.size();
            break;
        case 4:while(!pq.empty())
        {
            cout << pq.top() << " ";
            pq.pop();
        }
    }
    cout << "\n Do you want to continue? ";
    cin >> ans;
} while (ans == 'y');
return 0;
}
```

Output

Main Menu
1.Insert element
2.Delete element
3.Size of queue
4.Display of queue
Enter your choice 1

Enter the element to be inserted : 10

Item is inserted!!!

```
Do you want to continue ? y
```

```
Main Menu  
1.Insert element  
2.Delete element  
3.Size of queue  
4.Display of queue  
Enter your choice 1
```

```
Enter the element to be inserted : 20
```

```
Item is inserted!!!  
Do you want to continue ? y
```

```
Main Menu  
1.Insert element  
2.Delete element  
3.Size of queue  
4.Display of queue  
Enter your choice 1
```

```
Enter the element to be inserted : 30
```

```
Item is inserted!!!  
Do you want to continue ? y
```

```
Main Menu  
1.Insert element  
2.Delete element  
3.Size of queue  
4.Display of queue  
Enter your choice 4  
30 20 10  
Do you want to continue ?
```

6.6 Algorithms

The algorithms are used to process the contents of the containers. The functionalities provided in container are not sufficient to perform complex operations hence the algorithms are used to support more complex operations for the containers. Using algorithms the reusability can be achieved in STL.

In order to access the STL algorithms we have to write `<algorithm>` in the program.

Various categories of algorithm are -

- 1. Sorting algorithms** - These algorithms contain the functionalities related to sorting of the list.

- 2. Mutating sequence algorithms** - These algorithms modify the contents of the container. For example copy() operation will modify the contents of the container.
- 3. Non mutating sequence algorithms** - These algorithms do not modify the contents of the container as they work. For instance count() will simply count the occurrences in the container.
- 4. Numerical algorithms** - These algorithms are useful for performing some computations. For instance sum of all the elements can be obtained by the function accumulate().

We will summarize various algorithms and supporting functionalities in following tables -

Sorting algorithm

Sr. No.	Function	Description
1	sort()	Using quick sort the elements are sorted.
2	stable_sort()	Using stable sorting method the elements are sorted.
3	merge()	This function is used for merging the elements.
4	sort_heap()	Performs sorting on already created heap.
5	min()	It finds the minimum element.
6	max()	It finds the maximum element.
7	binary_search()	It performs the binary search on the sorted elements to search for particular element.

Mutating algorithm

Sr. No.	Function	Description
1	copy()	Copies the sequence of elements.
2	copy_backward()	Copies the sequence of elements from end of the list, i.e. copies the list in backward direction.
3	reverse()	This function is used to reverse the given sequence of elements.
4	unique()	It finds the adjacent duplicate elements and removes them.

Nonmutating algorithm

Sr. No.	Function	Description
1	find()	It will find the position of desired element.
2	count()	This function counts the number of elements in the given sequence.
3	equal()	It checks whether the two sequences are equal or not. If two sequences are matching then it returns true.
4	search()	This operation is used for searching the desired element from the given sequence.

Numerical algorithm

Sr. No.	Function	Description
1	accumulate()	Successive elements are summerized and a sum of all the elements in a given sequence is obtained.
2	inner_product()	It performs the product operation on a pair of sequences.
3	partial_sum()	It obtains the sequence by summing the pair of sequences.
4	adjacent_difference()	It produces a sequence from another sequence.

6.6.1 Searching Algorithm

There are various functions available for searching an element from the list such as find, search, binary search . Let us discuss the implementation of them.

Note that : The header file<algorithm>should be inserted in the C++ program to make use of above algorithms.

Example of find()

```
#include <iostream>
#include <algorithm> //for find()
using namespace std;
int a[] = { 10,20,30,40,50 };
int key;
int main()
{
    int* ptr;
    for (int i = 0; i < 5; i++)
        cout << " " << a[i];
    cout << "\n Enter the element to be searched from array: ";
    cin >> key;
```

```

ptr = find(a, a + 5, key);
cout << "\n The elements is present at index: "<<(ptr-a);
return 0;
}

```

Output

```

10 20 30 40 50
Enter the element to be searched from array: 30
The elements is present at index: 2

```

Example of search

```

#include <iostream>
#include <algorithm> //for search()
using namespace std;
int a[] = { 10,20,30,40,50,60,70,80 };
int b[] = { 30,40,50 };
int key;
int main()
{
    int* ptr;
    cout << "\nArray a[8]= ";
    for (int i = 0; i < 8; i++)
        cout << " " << a[i];
    cout << "\nArray b[3]= ";
    for (int j = 0; j < 3; j++)
        cout << " " << b[j];
    ptr = search(a, a + 8, b, b + 3);
    if (ptr == a + 8)
        cout << "\n The pattern is not present";
    else
        cout << "\n The pattern is present from index: "<<(ptr-a);
    return 0;
}

```

Output

```

Array a[8]= 10 20 30 40 50 60 70 80
Array b[3]= 30 40 50
The pattern is present from index: 2

```

Example of Binary Search

```

#include <iostream>
#include <algorithm>
using namespace std;
int a[] = { 10,20,30,40,50,60,70,80 };
int key;
char ans='y';
int main()
{

```

```

cout << "\nArray a[8] = ";
for (int i = 0; i < 8; i++)
    cout << " " << a[i];
do
{
    cout << "\n Enter the element to be searched: ";
    cin >> key;
    if (binary_search(a, a + 8, key))
        cout << "\n The " << key << " is present";
    else
        cout << "\n The " << key << " is not present";
    cout << "\n Do you want to continue?(y/n) ";
    cin >> ans;
} while (ans == 'y');
return 0;
}

```

Output

Array a[8] = 10 20 30 40 50 60 70 80
Enter the element to be searched : 40

The 40 is present
Do you want to continue ? (y / n) y

Enter the element to be searched : 99

The 99 is not present
Do you want to continue ? (y / n)n

Example of find_if

```

#include <iostream>
#include <algorithm>
#include<vector>
using namespace std;
int a[] = { 10,20,30,40,50,60,70,80 };
int key;
char ans='y';
bool isEven(int val)
{
    if (val % 2 == 0)
        return true;
    else
        return false;
}
int main()
{
    vector<int>::iterator it;

```

```
cout << "\nArray a[8] = ";
for (int i = 0; i < 8; i++)
    cout << " " << a[i];
if(find_if(a, a + 8, isEven))
    cout << "\n There are even numbers present in the list";
else
    cout << "\n There are not any even numbers present in the list";

return 0;
}
```

Output

```
Array a[8] = 10 20 30 40 50 60 70 80
There are even numbers present in the list
```

6.6.2 Sorting Algorithm

We will write a simple program for sorting the elements of an array using sorting algorithm.

```
#include <iostream>
#include <algorithm> // keyword algorithm included
#define SIZE 10
using namespace std;
int main()
{
    int n, item;
    int array[SIZE], i;
    cout << "How Many Elements You Want to Enter";
    cin >> n;

    // setting the range for sorting
    int *Limit = array + n;

    cout << "Enter The Numbers";
    for (i = 0; i < n; ++i)
    {
        cin >> item;
        array[i] = item;
    }

    // calling the function from algorithm
    sort(array, Limit);
    // displaying the sorted list of elements
    cout << "\n The sorted list is ..." << endl;
    for (i = 0; i < n; ++i)
```

```

cout << array[i] << '\t';
cout << endl;
return 0;
}

```

Output

How Many Elements You Want to Enter 7

Enter The Numbers

```

4
3
1
2
7
6
5

```

The sorted list is ...

```

1   2   3   4   5   6   7

```

In above program the keyword `<algorithm>` is used and the sort is a function that is supported by the sorting algorithm. By this function the quick sort is performed over the range of the array from first element to the last element of the array. Hence we set the range by a *Limit* variable(from 0th position of array to N).

6.6.3 Min Max Algorithm

There are various functions available for finding min and max values. For instance - **min()**, **max()**, **minmax()** and **minmax_element()**

The syntax are -

- `min(element1,element2)` - returns minimum element
- `max(element1,element2)` - returns maximum element
- `minmax(object type1,object type2)` - returns a pair in which first element is minimum element and the second element is maximum element.
- `Minmax_element(iterator first, iterator last, compare_function)` - returns a pair of iterator where first element of the pair points to the smallest element in the range [first,last] and second element of the pair points to the largest element in the range [first,last].

The header file `<algorithm>` need to be inserted in the program to work with above mentioned algorithms

Example of min()

```

#include <iostream>
#include <algorithm>
using namespace std;

```

```
int main()
{
    cout << "\n min(20,10)= " << min(20, 10);
    cout << "\n min('t','z')= " << min('t', 'z');
    cout << "\n min(88.99,100.44)= " << min(88.99, 100.44);
    return 0;
}
```

Output

```
min(20,10)= 10
min('t','z')= t
min(88.99,100.44)= 88.99
```

Example of max()

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    cout << "\n max(20,10)= " << max(20, 10);
    cout << "\n max('t','z')= " << max('t', 'z');
    cout << "\n max(88.99,100.44)= " << max(88.99, 100.44);
    return 0;
}
```

Output

```
max(20,10)= 20
max('t','z')= z
max(88.99,100.44)= 100.44
```

Example of minmax()

- The **minmax()** is a unique function to retrieve min and max values from two values or from an entire array.
- With the keyword **auto** we don't need to tell what is the type of each element.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    auto a = { 10,20,30,40,50 };
    auto result = minmax(a);
    cout << "\n Using minmax(10,20,30,40,50)\n";
    cout << "\n The minimum element =" << result.first;
    cout << "\n The maximum element =" << result.second;
    return 0;
}
```

Output

Using minmax(10,20,30,40,50)
 The minimum element = 10
 The maximum element = 50

Example of minmax_element

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;

int main()
{
    vector<int> a = { 10,5,20,25,35,40,30 };

    auto result = minmax_element(a.begin(), a.end());
    cout << "\nmin value is " << *result.first;
    cout << "\nmax value is " << *result.second;
    return 0;
}
```

Output

min value is 5
 max value is 40

6.7 Set Operations

Various set operations are union, intersection, difference and symmetric difference.
 For example

Consider set

$$\begin{aligned} A &= \{10,20,30,40\} \\ B &= \{10,30,50,60\} \\ A \cup B &= \{10,20,30,40,50,60\} \\ A \cap B &= \{10,30\} \\ A-B &= \{20,40\} \\ B-A &= \{ 50,60\} \end{aligned}$$

1. Union operation

Using **set_union** method the union of two sets can be computed.

Syntax

```
OutputIterator set_union(  
    InputIterator1 First1,  
    InputIterator1 Last1,  
    InputIterator2 First2,  
    InputIterator2 Last2,  
    OutputIterator Result,  
)
```

where **First1**,**First** are the input iterators addressing the positions of the first element in the first and second source of elements.

Last1 and **Last2** are the input iterators addressing the positions of the one past the last element in the first and second source of elements.

Result is an output iterator storing the union of two source of elements

2. Intersection operation

Using **set_intersection** method the intersection of two sets can be computed.

Syntax

```
OutputIterator set_intersection(  
    InputIterator1 First1,  
    InputIterator1 Last1,  
    InputIterator2 First2,  
    InputIterator2 Last2,  
    OutputIterator Result,  
)
```

where **First1**,**First2** are the input iterators addressing the positions of the first element in the first and second source of elements.

Last1 and **Last2** are the input iterators addressing the positions of the one past the last element in the first and second source of elements.

Result is an output iterator storing the intersection of two source of elements

3. Difference operation

Using **set_difference** method the difference of two sets can be computed.

Syntax

```
OutputIterator set_difference(  
    InputIterator1 First1,  
    InputIterator1 Last1,  
    InputIterator2 First2,  
    InputIterator2 Last2,  
    OutputIterator Result,  
)
```

where **First1,First2** are the input iterators addressing the positions of the first element in the first and second source of elements.

Last1 and **Last2** are the input iterators addressing the positions of the one past the last element in the first and second source of elements.

Result is an output iterator storing the difference of two source of elements

4. Symmetric Difference

Using `set_symmetric_difference`, method the symmetric difference of two sets can be computed.

Syntax

```
OutputIterator set_symmetric_difference (
    InputIterator1 First1,
    InputIterator1 Last1,
    InputIterator2 First2,
    InputIterator2 Last2,
    OutputIterator Result,
);
```

where **First1,First2** are the input iterators addressing the positions of the first element in the first and second source of elements.

Last1 and **Last2** are the input iterators addressing the positions of the one past the last element in the first and second source of elements.

Result is an output iterator storing the symmetric difference of two source of elements.

Let us see the implementation of the set operations

C++ Program

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    int A[] = { 10,20,30,40 };
    int B[] = { 50,60,30,10 };
    vector<int> v(10);
    vector<int>::iterator it;

    sort(A, A + 4);
    sort(B, B + 4);
    cout << "\n Set A = ";
    cout << "{ ";
    for (int j = 0; j < 4; j++)
        v[j] = A[j];
    cout << " } \n Set B = ";
    cout << "{ ";
    for (int j = 0; j < 4; j++)
        v[j] = B[j];
    cout << " } \n Symmetric Difference = ";
    cout << "{ ";
    for (int j = 0; j < 4; j++)
        if (v[j] != v[j + 1])
            cout << v[j] << " ";
    cout << " } \n";}
```

```

        cout << " " << A[j];
cout << "}";
cout << "\n Set B = ";
cout << "{ ";
for (int j = 0; j < 4; j++)
    cout << " " << B[j];
cout << " }";

cout << "\n\t The Union of two Sets is... \n";
it = set_union(A, A + 4, B, B + 4, v.begin());

v.resize(it - v.begin());

cout << "The union has " << (v.size()) << " elements:\n";
cout << "{ ";
for (it = v.begin(); it != v.end(); ++it)
    cout << ' ' << *it;
cout << " }";
cout << "\n\n \tThe intersection of two Sets is... \n";
it = set_intersection(A, A + 4, B, B + 4, v.begin());
v.resize(it - v.begin());

cout << "The intersection has " << (v.size()) << " elements:\n";
cout << "{ ";
for (it = v.begin(); it != v.end(); ++it)
    cout << ' ' << *it;
cout << " }";

cout << "\n\t The difference of two Sets(A-B) is... \n";
it = set_difference(A, A + 4, B, B + 4, v.begin());
v.resize(it - v.begin());
cout << "The difference has " << (v.size()) << " elements:\n";
cout << "{ ";
for (it = v.begin(); it != v.end(); ++it)
    cout << ' ' << *it;
cout << " }";
return 0;
}

```

Output

Set A = { 10 20 30 40 }

Set B = { 10 30 50 60 }

The Union of two Sets is...

The union has 6 elements:

{ 10 20 30 40 50 60 }

The intersection of two Sets is...

The intersection has 2 elements :

{ 10 30 }

The difference of two Sets(A - B) is...

The difference has 2 elements :

{ 20 40 }

6.8 Heap Sort

Definition : Heap is a **complete binary tree** or a almost complete binary tree in which every **parent node** be either **greater or lesser than** its child nodes.

A **Max heap** is a tree in which value of each node is greater than or equal to the value of its children nodes.

For example :

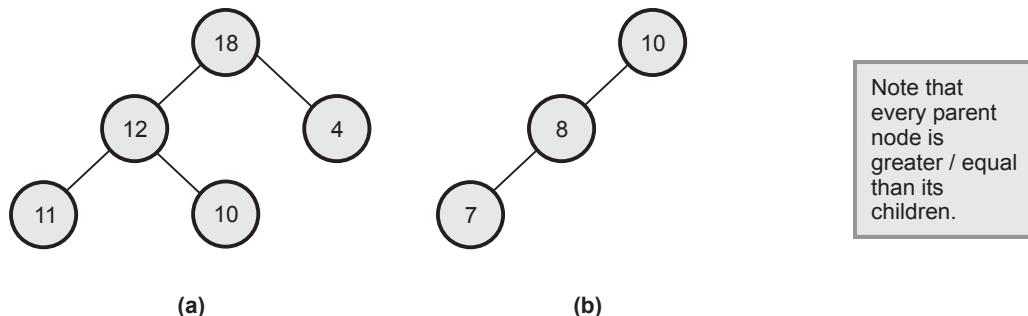


Fig. 6.8.1 Max heap

A **Min heap** is a tree in which value of each node is less than or equal to value of its children nodes.

For example :

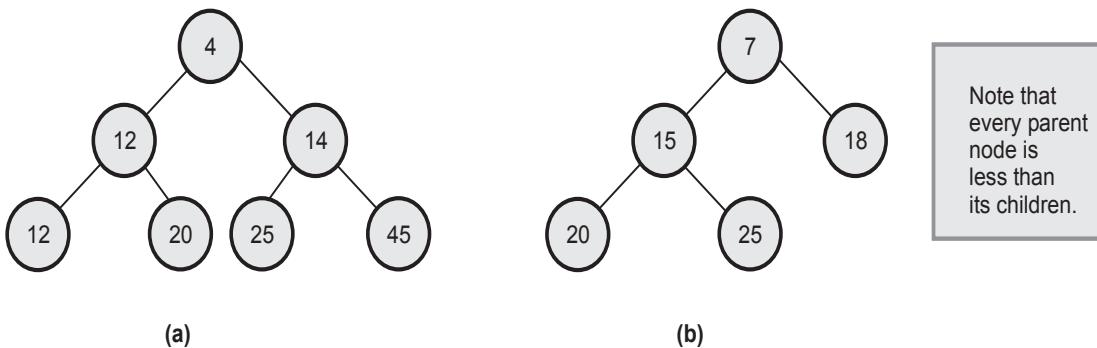
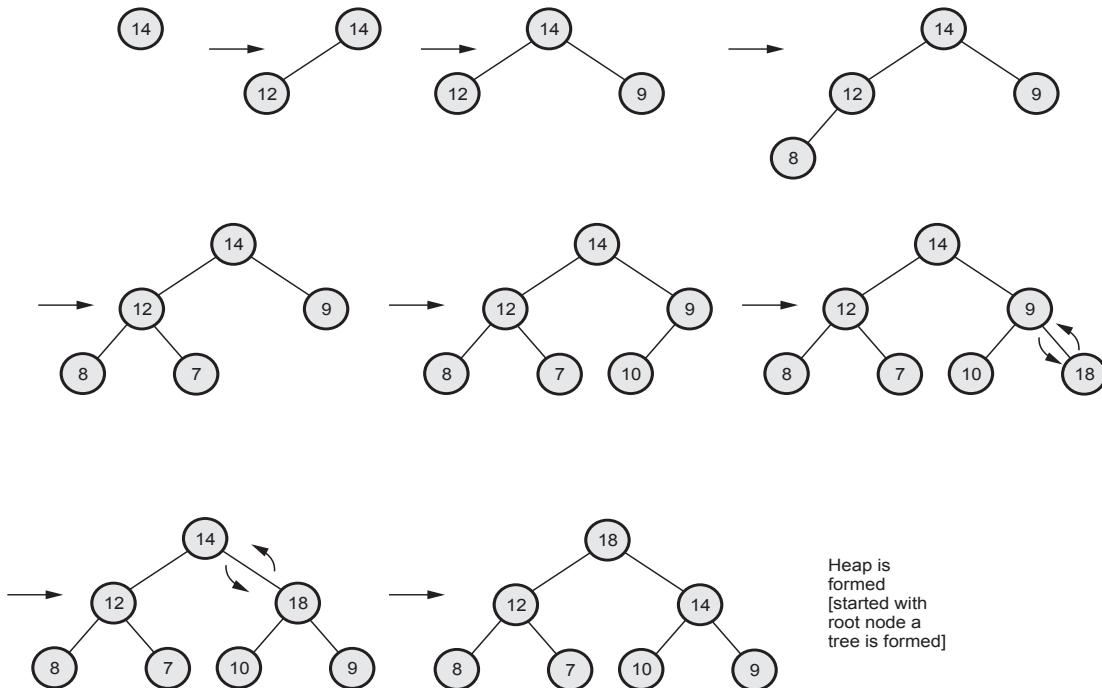


Fig. 6.8.2 Min heap

We can construct heap using top down approach with repeated insertion of element.

After applying Heap sort technique we should get the list as follows -



After applying heap sort technique the sorted list for Max heap will be

18 14 12 10 9 8 7

After applying heap sort technique the sorted list for Min heap will be

7 8 9 10 12 14 18

The **sort_heap** function which sorts the heap in ascending order.

Various operations that can be used for heap sort are

<code>make_heap(begin,end)</code>	Make heap from range
<code>push_heap(element)</code>	Push element into heap range
<code>pop_heap(begin,end)</code>	Pop element from heap range
<code>sort_heap(begin,end)</code>	Sort the heap

In the following C++ program we are performing heaps sort

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
```

```

{
    int a[] = { 14,10,67,29,57,12,32, };
    vector<int> v(a, a + 7);
    cout << " List Before heap operation is... :\n";
    for (unsigned i = 0; i < v.size(); i++)
        cout << ' ' << v[i];
    cout << "\n\t\t Heapifying....";
    make_heap(v.begin(), v.end());
    cout << "\n\n The root of max heap : " << v.front() << '\n';
    cout << " List after heap operation is... \n";
    for (unsigned i = 0; i < v.size(); i++)
        cout << ' ' << v[i];
    cout << "\n\t\t Sorting....";
    sort_heap(v.begin(), v.end());
    cout << "\n sorted List is... \n";
    for (unsigned i = 0; i < v.size(); i++)
        cout << ' ' << v[i];
    cout << "\n-----\n";
    v.push_back(100);
    push_heap(v.begin(), v.end());
    cout << "\nmax heap after inserting 100, the root is: " << v.front() << '\n';
    sort_heap(v.begin(), v.end());
    cout << "\n sorted List is... \n";
    for (unsigned i = 0; i < v.size(); i++)
        cout << ' ' << v[i];
    cout << "\n-----\n";
    pop_heap(v.begin(), v.end());
    v.pop_back();
    cout << "\nAfter deleting one node from max heap, the root is : " << v.front() << '\n';
    sort_heap(v.begin(), v.end());
    cout << "\n sorted List is... \n";
    for (unsigned i = 0; i < v.size(); i++)
        cout << ' ' << v[i];

    return 0;
}

```

Output

List Before heap operation is... :

14 10 67 29 57 12 32

Heapifying....

The root of max heap : 67

List after heap operation is...

67 57 32 29 10 12 14

Sorting....

sorted List is...

10 12 14 29 32 57 67

```
max heap after inserting 100, the root is: 100
sorted List is...
10 12 29 32 57 67 14 100
```

```
After deleting one node from max heap, the root is : 100
sorted List is...
12 14 32 57 67 29 100
```

6.9 Iterators- Input, Output, Forward, Bidirectional and Random Access

The iterators are basically objects but sometimes they can be pointers and hence iterators specify the positions in container. The iterators are used to traverse the contents of container. There are five types of iterators.

Iterator	Description
Random access	Elements can be stored or retrieved randomly.
Forward	The elements can be stored or retrieved but only forward moving is allowed.
Input	Elements retrieving is allowed with forward moving.
Output	Elements storing is allowed with forward moving.
Bidirectional	Store and retrieve the elements and forward/backward moving is allowed.

Levels of functionalities of different iterators is as shown below.

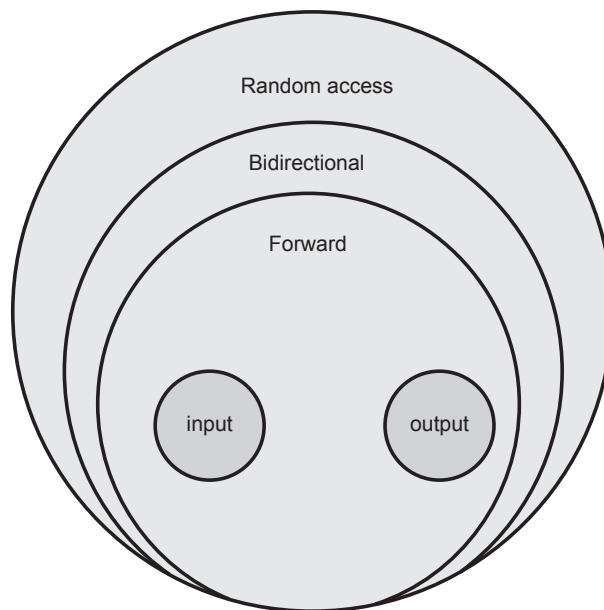


Fig. 6.9.1 Types of iterator

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    int arr[4] = { 1, 2, 3, 4 }, *ptr = arr;
    set<int, greater<int> > s;
    set<int, greater<int> > :: const_iterator it;
    while (ptr != arr + 4)
        s.insert(*ptr++);
    cout << "The numbers below 5 : " << endl;
    for (it = s.begin(); it != s.end(); ++it)
        cout << *it << 't'; // Using iterator 'it' traversing array
    cout << endl;
    return 0;
}
```

Output

The numbers below 5 :

4 3 2 1

The above given program is a simple example in which the iterator for the set container is used. The iterator is given to be variable *it* which traverse through the range from *s.begin()* to *s.end()*. The iterator is basically a pointer which navigates the contents of the container.

- The operators used for iterating through the container are enlisted in the following table

Operator	Purpose
<code>++</code>	Make the iterator step forward to the next element.
<code>==</code> and <code>!=</code>	Return whether two iterators represent the same position or not.
<code>=</code>	Assigns an iterator

- The most commonly used functions for iterating through container are -

Operator	Purpose
<code>begin()</code>	returns an iterator representing the beginning of the elements in the container.
<code>end()</code>	returns an iterator representing the element just past the end of the elements.
<code>cbegin()</code>	returns a const (read-only) iterator representing the beginning of the elements in the container.
<code>cend()</code>	returns a const (read-only) iterator representing the element just past the end of the elements.

- Iterator can be specified in two ways
 - **container::iterator** provides a read/write iterator
 - **container::const_iterator** provides a read-only iterator

Implementation

Example 1 : Following program demonstrates how to use iterator for iterating the list of integers.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst;
    for (int i = 1; i <= 5;i++)
        lst.push_back(i);
    // iterate over all elements and print, separated by space
    cout << "\n Iterating through list...\n";
    list<int>::const_iterator it;
    for (it = lst.begin(); it != lst.end(); ++it)
        cout << *it << ' ';
    cout << endl;
    return 0;
}
```

Output

Iterating through list...

1 2 3 4 5

Example 2 : Following program demonstrates how to use iterator for iterating the **set** of integers.

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    set<int> lst;
    for (int i = 1; i <= 5;i++)
        lst.insert(i);
    // iterate over all elements and print, separated by space
    cout << "\n Iterating through set...\n";
    set<int>::const_iterator it;
```

```
for (it = lst.begin(); it != lst.end(); ++it)
    cout << *it << ' ';
cout << endl;
return 0;
}
```

Output

Iterating through set...

1 2 3 4 5

6.10 Object Oriented Programming - A Road Map to Future

- Being an object oriented programming C++ is a backbone of many programming paradigm and logic design
- C++ play an important role for **Object Oriented Modeling and Design(OOMD)**. The Unified Modeling Language (UML) take full advantages of all the features offered by C++.
- There are certain features such as polymorphism, overloading of operators, templates, STL, Runtime Type Identification(RTTI), dynamic type casting, Strings and I/O streams with the support for international character set and localization - and so on that has a great support for generation of runtime efficient applications.
- C++ is a natural choice for robotic programming because **robotics** require high performance code. For handling low level motor controlling routines, the C++ language is useful.



Notes

Object Oriented Programming - Laboratory

Group B

- Experiment 1** Create user defined exception to check the following conditions and throw the exception if the criterion does not meet. a. User has age between 18 and 55 b. User stays has income between ₹ 50,000 - ₹ 1,00,000 per month c. User stays in Pune/ Mumbai/ Bangalore / Chennai d. User has 4-wheeler accept age, income, city, vehicle from the user and check for the conditions mentioned above. If any of the condition not met then throw the exception..... L - 2
- Experiment 2** Write a C++ program that creates an output file, writes information to it, closes the file and open it again as an input file and read the information from the file

..... L - 5

- Experiment 3** Write a function template selection sort. Write a program that inputs, sorts and outputs an integer array and a float array. L - 6

Group C

- Experiment 4** Write C++ program using STL for sorting and searching with user defined records such as person record(Name, DOB, telephone number), item record (Item code, name, cost, quantity) using vector container..... L - 8
- Experiment 5** Write a program in C++ to use map associative container. The keys will be the names of states and the values will be the populations of the states. When the program runs, the user is prompted to type the name of a state. The program then looks in the map, using the state name as an index and returns the population of the state.....L - 11

Group B

Experiment 1 Create user defined exception to check the following conditions and throw the exception if the criterion does not meet. a. User has age between 18 and 55 b. User stays has income between ₹ 50,000 - ₹ 1,00,000 per month c. User stays in Pune/ Mumbai/ Bangalore / Chennai d. User has 4-wheeler accept age, income, city, vehicle from the user and check for the conditions mentioned above. If any of the condition not met then throw the exception.

C++ Program

```
#include<iostream>
using namespace std;
#include<cstring>
class My_Exception
{
public:
    int age;
    double income;
    char city[20];
    bool fourWheeler;
    char msg[50];
    My_Exception() { strcpy(msg, ""); age = 0; strcpy(city, ""); fourWheeler = 0; }
    My_Exception(char s[50], int i)
    {
        strcpy(msg, s);
        age = i;
    }
    My_Exception(char s[50], double i)
    {
        strcpy(msg, s);
        income = i;
    }
    My_Exception(char s[50], char *i)
    {
        strcpy(msg, s);
        strcpy(city, i);
    }
    My_Exception(char s[50], bool i)
    {
        strcpy(msg, s);
        fourWheeler = i;
    }
};
```

```
int Mystrcmp(char str1[20],char str2[20])//Used defined function for string
                                // comparison
{
    int i, j;
    int flag = 0;
    j = 0;
    for (i = 0; str1[i] != '\0'; i++)
    {
        if (str1[i] == str2[j])//each character from str1 and str2 is
        // compared
            j++;
        else
        {
            flag = 1; //when mismatch is found set flag to 1
            break;
        }
    }
    if (flag == 1) //flag 1 means mismatching
        return 1;
    else
        return 0; // flag 0 means both the strings are equal
}
void main()
{
    int Age;
    double Income;
    char City[20] = {};
    char c[4][20] = { "Pune", "Mumbai", "Banglore", "Chennai" };
    bool FourWheeler=0;
    int Mystrcmp(char City[20],char c[20]);
    try
    {
        cout << "Enter age : ";
        cin >> Age;
        if ((Age<18) | | (Age>55))
            throw My_Exception("Age Criterion does not meet", Age);//user defined
exception
        else
            cout << "Age: " << Age << endl;

        cout << "Enter Income : ";
        cin >> Income;
        if ((Income<50000) | | (Income>100000))
            throw My_Exception("Income Criterion does not meet", Age);
        else
            cout << "Income: " << Income << endl;
```

```
cout << "Enter 1 if person has Fourwheeler otherwise 0 : ";
cin >> FourWheeler;
if (!FourWheeler)
    throw My_Exception("FourWheeler Criterion does not meet",
FourWheeler);
else
    cout << "Fourwheeler: " << FourWheeler << endl;
cout << "Enter City Name : ";
cin >> City;

if ((Mystrcmp(City,c[0]) == 1) && (Mystrcmp(City, c[1]) == 1) &&
(Mystrcmp(City, c[2]) == 1) && (Mystrcmp(City, c[3]) == 1))
    throw My_Exception("City Criterion does not meet", City);
else
    cout << "City: " << City << endl;
cout << endl;

}
catch (My_Exception obj)
{
    cout << "Error!!! "<< obj.msg << endl;
}
}
```

Output(Run1)

Enter age : 15
Error!!!Age Criterion does not meet

Output(Run2)

Enter age : 25
Age : 25
Enter Income : 4000
Error!!!Income Criterion does not meet

Output(Run3)

Enter age : 25
Age : 25
Enter Income : 55000
Income : 55000
Enter 1 if person has Fourwheeler otherwise 0 : 0
Error!!!FourWheeler Criterion does not meet

Output(Run4)

Enter age : 25
Age : 25
Enter Income : 55000
Income : 55000
Enter 1 if person has Fourwheeler otherwise 0 : 1
Fourwheeler : 1

```
Enter City Name : Delhi
Error!!!City Criterion does not meet
Enter age : 25
Age : 25
Enter Income : 55000
Income : 55000
Enter 1 if person has Fourwheeler otherwise 0 : 1
Fourwheeler : 1
Enter City Name : Pune
City : Pune
```

Output(Run5)

Experiment 2 Write a C++ program that creates an output file, writes information to it, closes the file and open it again as an input file and read the information from the file.

C++ Program

```
#include <iostream>
#include <fstream>
using namespace std;
#include <process.h>
const int MAX = 10;
int array1[MAX] = { 10,20,30,40,50 };
int array2[MAX];
int main()
{
    ofstream os; //create output stream
    os.open("d:\\test.dat", ios::trunc | ios::binary); //Opening file
    if (!os)
    {
        cerr << "Could not open output file\\n"; //Error Handling
        exit(1);
    }
    cout << "Writing the contents to the file...\\n\\n";
    os.write((char*)&array1,sizeof(array1)); //writing 'array1' to file
    if (!os)
    {
        cerr << "Could not write to file\\n"; //Error handling
        exit(1);
    }
    os.close(); //close the file
    ifstream is; //create input stream for reading the contents from file
    is.open("d:\\test.dat", ios::binary);
    if (!is)
    {
        cerr << "Could not open input file\\n"; //Error Handling
        exit(1);
    }
    cout << "Reading the contents from the file ...\\n";
```

```

is.read((char*)&array2,sizeof(array2)); //reading the contents in
                                         //another array 'array2'
if (!is)
{
    cerr << "Could not read from file\n"; //Error Handling
    exit(1);
}
for (int j = 0; j < MAX; j++) //check data
    cout << " " << array2[j];
return 0;
}

```

Output

Writing the contents to the file...

Reading the contents from the file ...

10 20 30 40 50 0 0 0 0 0

Experiment 3 Write a function template selection sort. Write a program that inputs, sorts and outputs an integer array and a float array.

C++ Program

```

#include<iostream>
using namespace std;
#define Size 10
int n;
template <class T>
void selection(T A[Size])
{
    int i, j, Min;
    T temp;
    for (i = 0; i <= n - 2; i++)
    {
        Min = i;
        for (j = i + 1; j <= n - 1; j++)
        {
            if (A[j] < A[Min])
                Min = j;

        }
        temp = A[i];
        A[i] = A[Min];
        A[Min] = temp;
    }
    cout << "\n The sorted List is ... \n";
    for (i = 0; i < n; i++)
        cout << "\t" << A[i];
}
int main()

```

```
{  
    int i, A[Size];  
    float B[Size];  
    cout << "\n\t\t Selection Sort\n ";  
    cout << "\n\t Handling Integer elements ";  
    cout << "\n How many elements are there ? ";  
    cin >> n;  
    cout << "\n Enter the integer elements\n ";  
    for (i = 0; i < n; i++)  
        cin >> A[i];  
    selection(A);  
    cout << "\n\t Handling Float elements ";  
    cout << "\n How many elements are there ? ";  
    cin >> n;  
    cout << "\n Enter the float elements\n ";  
    for (i = 0; i < n; i++)  
        cin >> B[i];  
    selection(B);  
    cout << "\n";  
    return 0;  
}
```

Output

Selection Sort
Handling Integer elements
How many elements are there ? 5
Enter the integer elements
30
50
40
20
10

The sorted List is ...
10 20 30 40 50
Handling Float elements
How many elements are there ? 5
Enter the float elements
44.44
22.22
11.11
55.55
33.33
The sorted List is ...
11.11 22.22 33.33 44.44 55.55

Group C

Experiment 4 Write C++ program using STL for sorting and searching with user defined records such as person record(Name, DOB, telephone number), item record (Item code, name, cost, quantity) using vector container.

C++ Program

```
#include <iostream>
#include<cstring>
#include<cstdlib>
#include<algorithm>
#include <vector>
using namespace std;
typedef struct rec
{
    char name[20];
    char BirthDt[20];
    char phone[11];
}node;
node temp;
vector<node> rec; // create an empty vector
vector<node>::iterator ptr;

bool compare(node &r1,node &r2)
{
    if(strcmp(r1.name,r2.name)<0)
        return true;
    else
        return false;
}
void Create()
{
    int n,i;
    cout <<"\n How many elements you want to insert?"<<endl;
    cin>>n;
    cout<<"\n Enter the Elements in the database"<<endl;
    for(i=0; i<n; i++)
    {
        cout<<"\n Name: ";
        cin>>temp.name;
        cout<<"\n Birth Date(dd-mm-yy): ";
        cin>>temp.BirthDt;
        cout<<"\n Phone: ";
        cin>>temp.phone;
        rec.push_back(temp);
    }
}
```

```
}

void Display()
{
    cout<<"\n\tThe contents of the database are: ";
    cout<<"\n-----";
    cout<<"\n Name           Birth Date       Phone";
    cout<<"\n-----";
    for(ptr=rec.begin();ptr!=rec.end();ptr++)//accessing the contents thru iterator
    {
        cout<<"\n";
        cout<<"    "<<(*ptr).name;
        cout<<"          "<<(*ptr).BirthDt;
        cout<<"          "<<(*ptr).phone;
    }
}

void Searching()
{
    char key[20];
    int flag=0;
    cout<<"\n Enter the name which you want to search ";
    cin>>key;
    for(ptr=rec.begin();ptr!=rec.end();ptr++)
    {
        if(strcmp((*ptr).name,key)==0)
        {
            flag=1;
            break;
        }
        else
            flag=0;
    }
    if(flag==1)
        cout<<"\n The desired element is present in the database ";
    else
        cout<<"\n The desired element is not present in the database ";
}
void Sorting()
{
    sort(rec.begin(),rec.end(),compare);
    cout <<"\n\n Record is Sorted!!! "<<endl;
}

int main()
{
    char ans='y';
    int choice;
    cout<<"\n Program for Searching and sorting ";
```

```
do
{
    cout<<"\n Main Menu ";
    cout<<"\n 1. Create a database ";
    cout<<"\n 2. Display a database ";
    cout<<"\n 3. Search particular element";
    cout<<"\n 4. Sort the database(based on name)";
    cout<<"\n Enter your choice: ";
    cin>>choice;
    switch(choice)
    {
        case 1:Create();
                  break;
        case 2:Display();
                  break;
        case 3:Searching();
                  break;
        case 4:Sorting();
                  Display();
                  break;
    }
    cout<<"\n Do you want to go back to Main Menu? ";
    cin>>ans;
}while(ans=='y');
return 0;
}
```

Output

Program for Searching and sorting
Main Menu
1. Create a database
2. Display a database
3. Search particular element
4. Sort the database(based on name)
Enter your choice: 1
How many elements you want to insert?
3
Enter the Elements in the database
Name: BBB
Birth Date(dd-mm-yy): 01-01-01
Phone: 1111
Name: CCC
Birth Date(dd-mm-yy): 11-11-19
Phone: 2222
Name: AAA
Birth Date(dd-mm-yy): 05-10-05
Phone: 3333
Do you want to go back to Main Menu? y

Main Menu

1. Create a database
2. Display a database
3. Search particular element
4. Sort the database(based on name)

Enter your choice: 2

The contents of the database are:

Name	Birth Date	Phone
BBB	01-01-01	1111
CCC	11-11-19	2222
AAA	05-10-05	3333

Do you want to go back to Main Menu? y

Main Menu

1. Create a database
2. Display a database
3. Search particular element
4. Sort the database(based on name)

Enter your choice: 3

Enter the name which you want to search CCC

The desired element is present in the database

Do you want to go back to Main Menu? y

Main Menu

1. Create a database
2. Display a database
3. Search particular element
4. Sort the database(based on name)

Enter your choice: 4

Record is Sorted!!!

The contents of the database are:

Name	Birth Date	Phone
AAA	05-10-05	3333
BBB	01-01-01	1111
CCC	11-11-19	2222

Do you want to go back to Main Menu?

Experiment 5 Write a program in C++ to use map associative container. The keys will be the names of states and the values will be the populations of the states. When the program runs, the user is prompted to type the name of a state. The program then looks in the map, using the state name as an index and returns the population of the state.

C++ Program

```
#include <iostream>
#include <map>
```

```
#include<string>
using namespace std;
int main()
{
    string country;
    int population;
    char ans = 'y';
    int choice;
    map<string,int> m;
    map<string,int>::iterator i;
    do
    {
        cout << "\n Main Menu";
        cout << "\n1. Insert an element";
        cout << "\n2. Display";
        cout << "\n3. Search an state";

        cout << "\n Enter your choice: ";
        cin >> choice;
        switch (choice)
        {
            case 1:cout << "\n Enter the name of state: ";
                      cin >> country;
                      cout << "\n Enter the population(in Cr): ";
                      cin >> population;
                      m.insert(pair<string,int>(country,population));
                      break;
            case 2:cout << "State and Populations are: ";
                      for (i = m.begin(); i != m.end(); i++) //i is for iterator
                          cout << "[" << (*i).first << ", " << (*i).second << "] ";
                      break;
            case 3:cout << "\n Enter the name of state for searching its population: ";
                      cin >> country;
                      if(m.count(country)!=0) // first represents key and second represents
                                              value
                          cout << "Population is " << m.find(country)->second << "Cr";
                      else
                          cout << "State is not present in the list" << endl;
                      break;
        }
        cout << "\n Do you want to continue?(y/n): ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');
    return 0;
}
```

Output

Main Menu

1. Insert an element
 2. Display
 3. Search an state
- Enter your choice: 1

Enter the name of state: China

Enter the population(in Cr): 143

Do you want to continue?(y/n): y

Main Menu

1. Insert an element
 2. Display
 3. Search an state
- Enter your choice: 1

Enter the name of state: India

Enter the population(in Cr): 130

Do you want to continue?(y/n): y

Main Menu

1. Insert an element
 2. Display
 3. Search an state
- Enter your choice: 1

Enter the name of state: USA

Enter the population(in Cr): 33

Do you want to continue?(y/n): y

Main Menu

1. Insert an element
 2. Display
 3. Search an state
- Enter your choice: 1

Enter the name of state: Indonesia

Enter the population(in Cr): 27

Do you want to continue?(y/n): y

Main Menu

- 1. Insert an element
- 2. Display
- 3. Search an state

Enter your choice: 1

Enter the name of state: Brazil

Enter the population(in Cr): 21

Do you want to continue?(y/n): y

Main Menu

- 1. Insert an element
- 2. Display
- 3. Search an state

Enter your choice: 2

Do you want to continue?(y/n): y

Main Menu

- 1. Insert an element
- 2. Display
- 3. Search an state

Enter your choice: 2

State and Populations are: [Brazil, 21] [China, 143] [India, 130] [Indonesia, 27] [USA, 33]

Do you want to continue?(y/n): y

Main Menu

- 1. Insert an element
- 2. Display
- 3. Search an state

Enter your choice: 3

Enter the name of state for searching its population: India

Population is 130 Cr

Do you want to continue?(y/n): n



SOLVED MODEL QUESTION PAPER

(As Per 2019 Pattern)

Object Oriented Programming

S.E. (Computer) Sem - I (End Sem)

Time : 2 ½ Hours]

[Total Marks : 70

Instructions to the candidates :

- 1) Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.
- 2) Neat diagrams must be drawn wherever necessary.
- 3) Figures to the right indicate full marks.
- 4) Make suitable assumptions if necessary.

Q.1 (a) Explain early binding and late binding. (Refer section 3.2) [6]

**(b) Write a C++ program for overloading a unary minus operator.
(Refer example 3.5.1) [6]**

(c) Explain explicit and mutable keywords. (Refer section 3.8) [6]

OR

**Q.2 (a) Differentiate between compile time and runtime polymorphism.
(Refer example 3.10.1) [4]**

(b) Explain the concept of function overloading with example. (Refer section 3.9.1) [6]

(c) Write a C++ program by illustrating pointer to base class. (Refer section 3.11) [8]

Q.3 (a) Write and explain purpose of various member functions used to handle stream errors. (Refer section 4.4) [6]

(b) Write a C++ program that reads a file and counts the number of sentences, words, and characters present in it. (Refer example 4.5.2) [6]

**(c) Explain the class object that can be used to read to or write from the file.
(Refer section 4.5.6) [5]**

OR

Q.4 (a) Explain the format flags with their meaning. (Refer section 4.7) [6]

(b) Explain I/O Manipulators. (Refer section 4.7.1) [6]

(c) Explain seekp and seekg function with illustrative example. (Refer section 4.8) [5]

Q.5 (a) Explain the concept of function template in detail. (Refer section 5.14) [6]

(b) Write a C++ program for finding the minimum value from an array using function template. (Refer example 5.17.1) [6]

(c) Explain the typename and export keywords. (Refer section 5.21) [6]

OR

Q.6 (a) Differentiate between error and exception. (Refer section 5.2) [6]

(b) Write a C++ program to handle divide by zero error. (Refer section 5.3) [6]

(c) How to handle multiple exceptions occurred in a program ? (Refer section 5.4) [6]

Q.7 (a) Explain the deque class using sequence container in C++. (Refer section 6.3.2) [5]

(b) Write various operations used in Multi-set container. (Refer section 6.4.2) [6]

(c) Explain how to implement stack operations in STL. (Refer section 6.5.1) [6]

OR

Q.8 (a) Explain various types of iterators. (Refer section 6.9) [8]

(b) What are various searching algorithms in STL ? How will you implement it ?

(Refer section 6.6.1) [9]



Notes

Notes

Notes

Notes