

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



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

<b>Part III : Functions</b>
-----------------------------

1.25 Functions.....	1 - 47
---------------------	--------

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

**Laboratory**

**(L - 1) to (L - 16)**



# 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	..... <b>Dec.-17,</b> ..... Marks 6
1.21	Class and Data Abstraction	
1.22	Class Scope and Accessing Class Members .	
1.23	Access Specifiers	..... <b>Dec.-19,</b> ..... Marks 2
1.24	Separating Interface from Implementation	
1.25	Functions	..... <b>May-17,</b> ..... Marks 8
1.26	Accessing Function and Utility Function	
1.27	Constructors	..... <b>Dec.-17,</b> ..... Marks 6
1.28	Destructor	..... <b>May-19,</b> ..... Marks 6
1.29	Objects and Memory Requirements	
1.30	Static Members : Variable and Functions . . .	<b>May-17, Dec.-19,</b> ..... Marks 4
1.31	Inline Function	..... <b>Dec.-16, 18, 19, May-19,</b> ..... Marks 6
1.32	Friend Function	..... <b>Dec.-16, 18, May-18, 19,</b> ..... 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 losing of an important information.

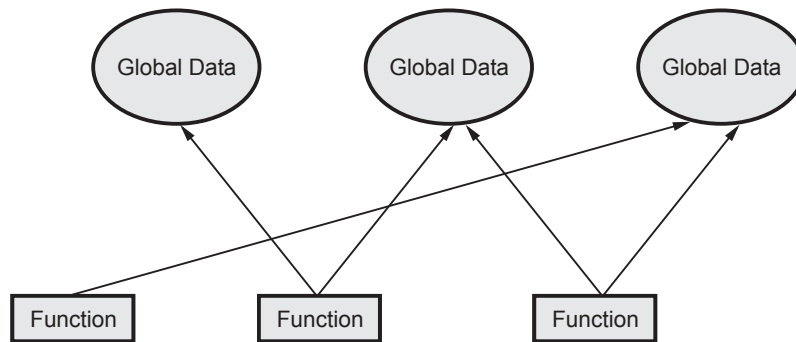


Fig. 1.2.1 Procedural paradigm

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.

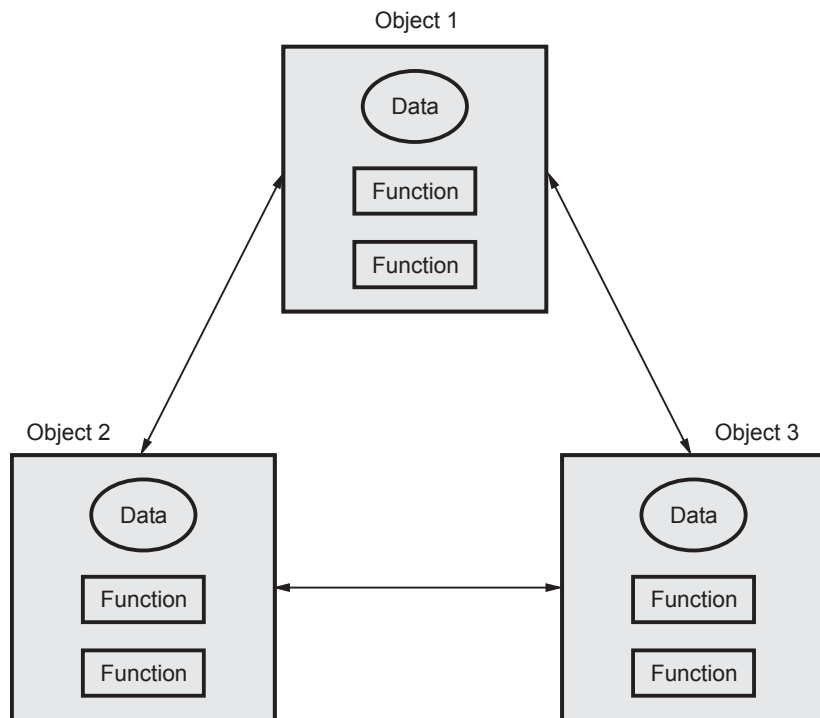


Fig. 1.4.1 Object oriented programming paradigm

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

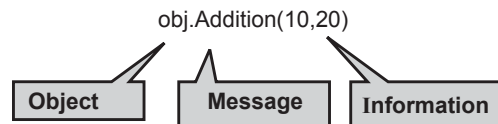


Fig. 1.5.1 Message

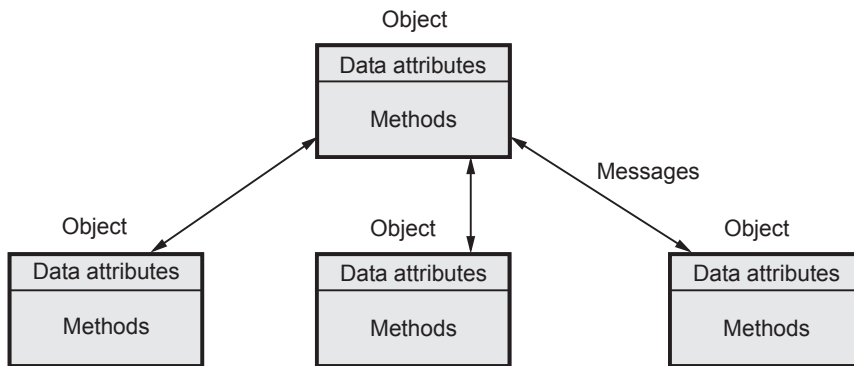


Fig. 1.5.2 Methods and messages

### 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.

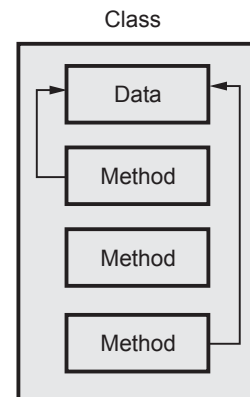


Fig. 1.5.3 Concept of encapsulation

### 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.

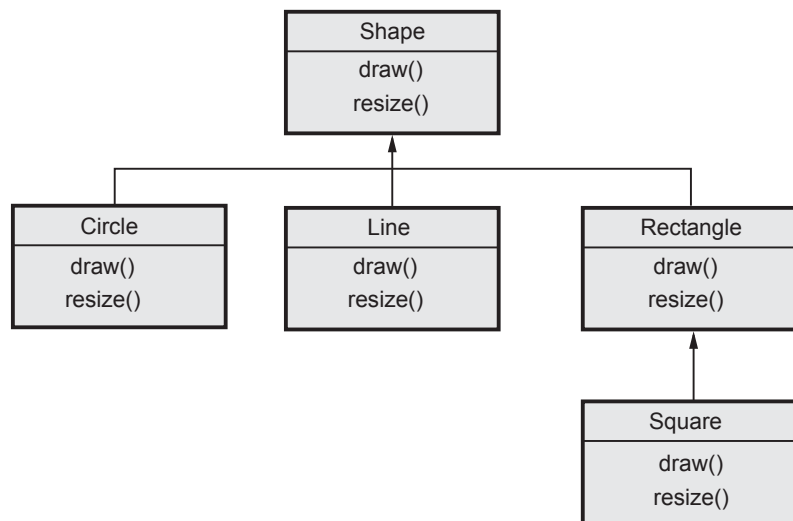


Fig. 1.5.4 Hierarchical structure of inheritance

### 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 -

1. Using **inheritance** the redundant code can be eliminated and the existing classes can be used.
2. The standard working **modules** can be created using object oriented programming. These modules can then communicate to each other to accomplish certain task.
3. Due to **data hiding** property, important data can be kept away from unauthorized access.
4. It is possible to create **multiple objects** for a given class.
5. For **upgrading the system** from small scale to large scale is possible due to object oriented feature.
6. Due to **data centered nature** of object oriented programming most of the details of the application model can be captured.
7. **Message passing technique** in object oriented programming allows the objects to communicate to the external systems.
8. **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 -

1. 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.
2. 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.
3. 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 Stroustrup** 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 <b>scanf</b> and <b>printf</b> statements.	The input and output is done using <b>cin</b> and <b>cout</b> statements.
4.	The I/O operations are supported by <b>stdio.h</b> header file.	The I/O operations are supported by <b>iostream.h</b> 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 printf and scanf functions.	The format specifier is not required in <b>cin</b> and <b>cout</b> functions.

## 1.9 C++ Programming Basics

### Structure of C++

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

Include file section
Class declaration section
Function definition section
Main function definition

**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  $x = r \cdot \cos(\text{angle})$  and  $y = r \cdot \sin(\text{angle})$ ).

**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.

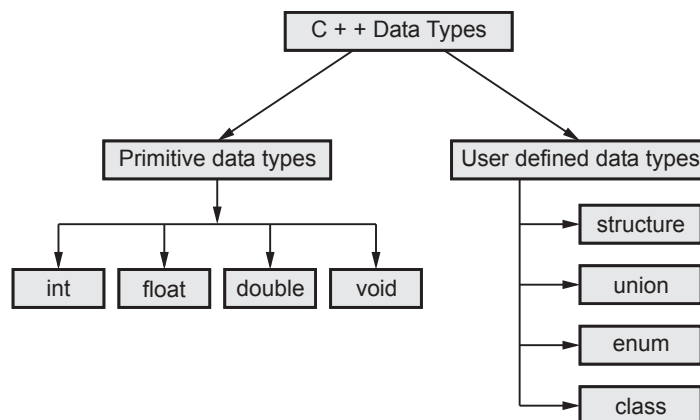


Fig. 1.10.1



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 <b>struct</b> has to be used.
4.	<b>Example :</b> int a [20];	<b>Example :</b> 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;
void main(void)
{
    cout<<"\n Enter the roll number: ";
    cin>>stud1.roll_no;
    cout << "\n Enter the name: ";
    cin>>stud1.name;
    cout<<"\n Enter the marks: ";
    cin>>stud1.marks;
}

```

Structure tag

Using dot. we are accessing member roll\_no

```

    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;
}

```

Using dot operator each member is printed

### Output

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&lt;&lt;"a is larger than b";     cout&lt;&lt;"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&lt;b)     cout&lt;&lt;"a is smaller than b"; else if(a&lt;c)     cout&lt;&lt;"a is smaller than b" else     cout&lt;&lt;"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;

	<pre>         }         case 2: cout&lt;&lt;"You have selected 2";                 break;         case 3: cout&lt;&lt;"You have selected 3";                 break;         default: cout&lt;&lt;"good bye";         } </pre>
--	---

## 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 -

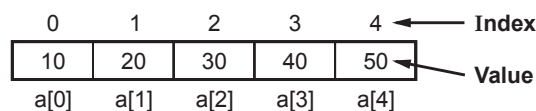


Fig. 1.17.1

### 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 ...\n";
        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 :**

int a[10][10]  
 rows ——— ↑      ↑ ——— columns

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 ...\n";
    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 ...

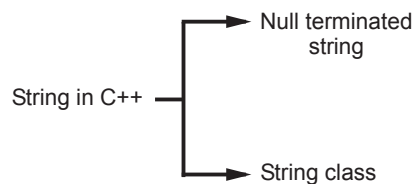
```
2 3 4
6 7 8
10 11 12
```

## 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<<"\nYou 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<<"\nYou 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	<code>strcpy(s1, s2);</code>	Copies string s2 into string s1.
2	<code>strcat(s1, s2);</code>	Concatenates string s2 onto the end of string s1.
3	<code>strlen(s1);</code>	Returns the length of string s1.
4	<code>strcmp(s1, s2);</code>	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	<code>strchr(s1, ch);</code>	Returns a pointer to the first occurrence of character ch in string s1.
6	<code>strstr(s1, s2);</code>	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 <b>public</b> .	By default the members of class are <b>private</b> .
2.	The structure <b>can not be inherited</b> .	The class can be <b>inherited</b> .
3.	The structures <b>do not require constructors</b> .	The classes <b>require constructors</b> for initializing the objects.
4.	A structure contains <b>only data</b> members.	A class contains the <b>data</b> as well as the <b>function</b> 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;
    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;
}

```

**Data members** are declared as **private**.

**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();
    return 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;
}
```

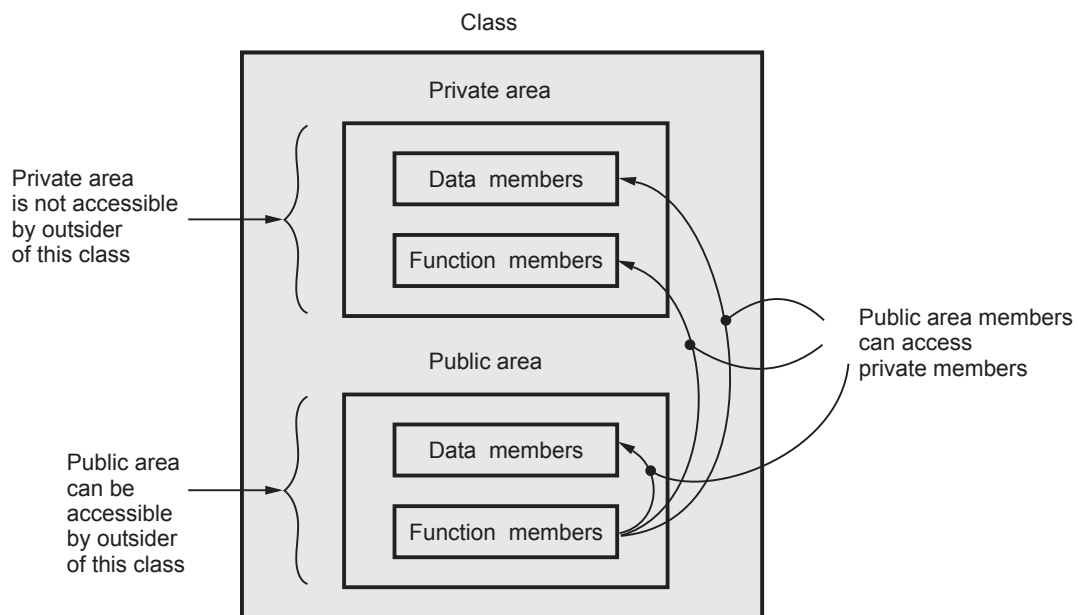
```
}  
void Test::Display()  
{ cout < "The sum is:" < c < "\n";}  
  
int main()  
{  
    Test obj;  
    obj.Addition(10,20);  
    obj.Display();  
    return 0;  
}
```

**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 by giving the name of the function. The syntax for this is

Name\_of\_function(parameter<sub>1</sub>,parameter<sub>2</sub>,...parameter<sub>n</sub>);

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;
}
```

Constructor is defined. Note that name of the constructor is similar to the name of the class. Purpose of constructor is to initialize the variables

object is created and values are initialized. When object gets created the compiler invokes the constructor **image()**

#### 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 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()
{
    class Test obj1,obj2(100);
    obj1.display();
    cout<<"\n";
    obj2.display();
    return 0;
}
```

#### 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

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

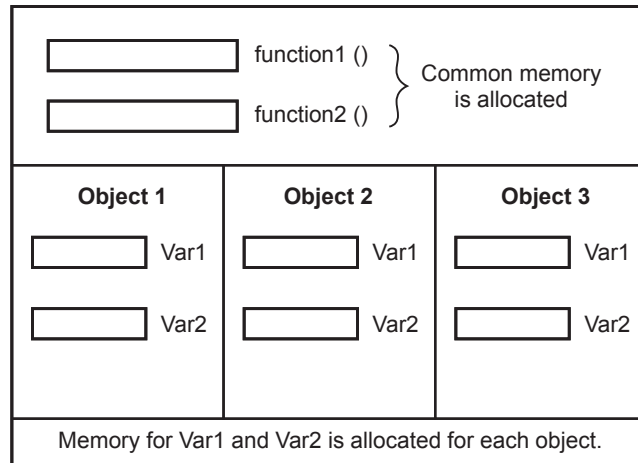
**SPPU : May-19, Marks 6**

## 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;
};
```

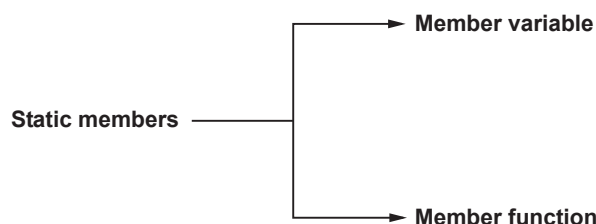
Test obj;

For **obj** size will be  $\text{sizeof}(a) + \text{sizeof}(b) + \text{sizeof}(d) = 4 + 2 + 1 = 7$  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
count = 0
count = 0
count = 0
count = 0
```

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



- 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	..... <b>Dec.-16, 19,</b> ..... Marks 2
2.20	Pointers Vs Arrays	
2.21	Accessing Arrays using Pointers	
2.22	Pointer Arithmetic	..... <b>Dec.-18,</b> ..... Marks 4
2.23	Arrays of Pointers	
2.24	Function Pointers	..... <b>Dec.-17, May-18, 19,</b> ..... 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 << "\n Enter the value of x";
    cin >> x;
    cout << "\n Enter 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

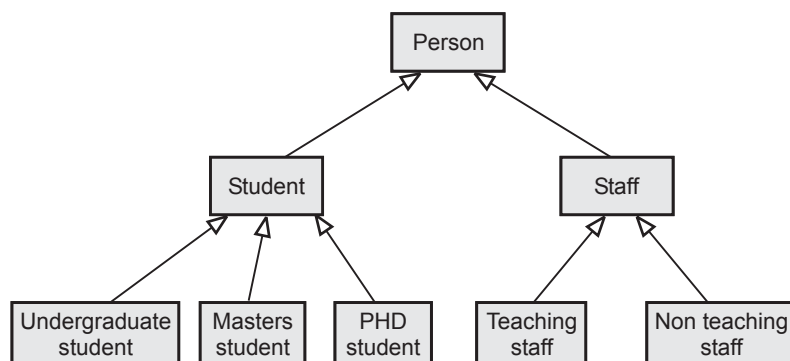
C(100-10) = 90
I'm print_msg() in class B
```

**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**

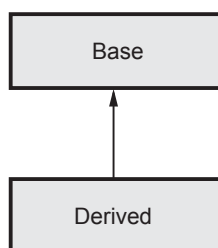
Various types of inheritance are -

- 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

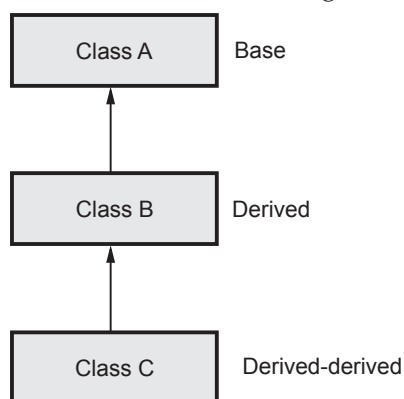
y = 20

**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;
```

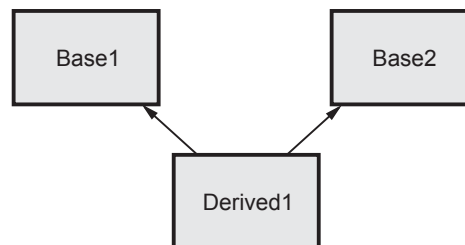
And now using this *obj* we can access the member of any class.

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**

#### C++ Programs

```
#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.

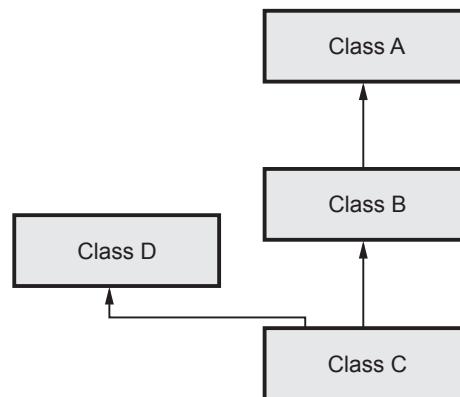


Fig. 2.9.4 Hybrid inheritance

### 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;
}
```

```
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 -

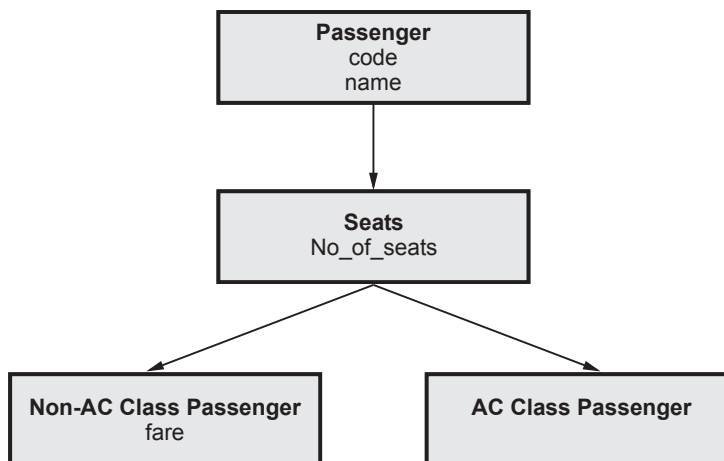


Fig. 2.9.5 Hierarchical inheritance

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<<"\n Enter 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

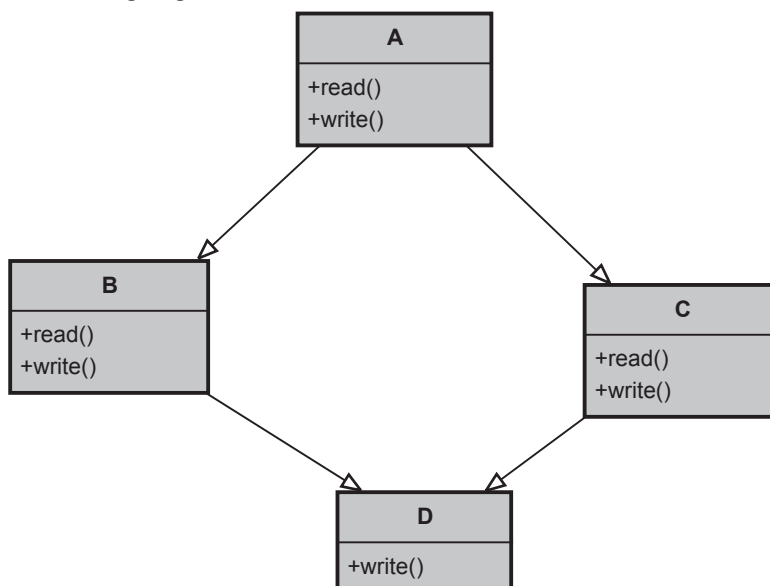
1. Write short notes on types of inheritance with respect to :  
(i) Single (ii) Multiple (iii) Hierarchical  
**SPPU : Dec.-19, Marks 6**
2. Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors  
**SPPU : Dec.-16, Marks 4**

## 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.

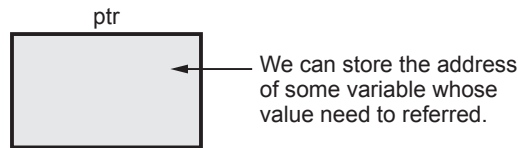


Fig. 2.15.1 Pointer variable

## 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 -

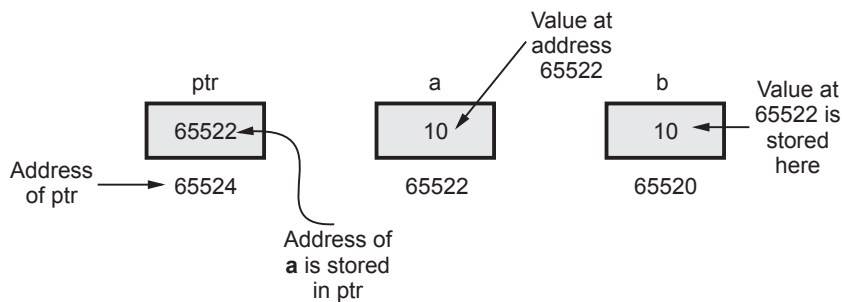


Fig. 2.16.1

**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;

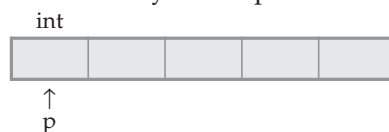
**For example :**

```
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



### 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.

<code>ptr1 &lt; ptr2</code>	The relational operations are possible on pointer variable while comparing two pointers.
<code>ptr1 &gt; ptr2</code>	
<code>ptr1 == ptr2</code>	
<code>ptr1 &lt;= ptr2</code>	
<code>ptr1 &gt;= ptr2</code>	
<code>ptr1 != ptr2</code>	

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

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

Now consider

```
int *ptr;
```

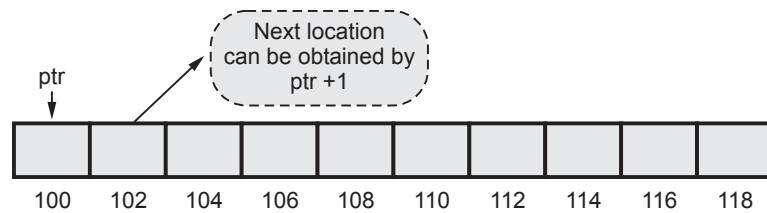
This is a pointer variable which is of integer type. This allocates the memory of 2 bytes for each such variable. If there are 10 integer pointers then total 20 bytes of memory block will be reserved. Consider a block in memory consisting of ten integers in a row. That is, 20 bytes of memory are set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

Now, let's say we point our integer pointer **ptr** at the first of these integers. Furthermore let's 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					
1	25	-17	4	45	10
0	1	2	3	4	5

`my_array [0]` is 1 or it is `(my_array + 0)`

`my_array [1]` is 25 or it is `(my_array + 1)`

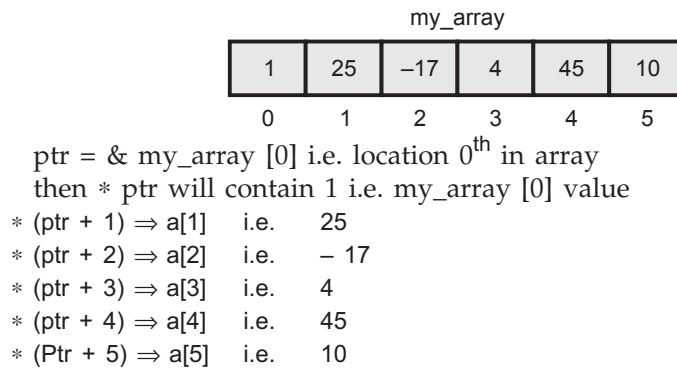
`my_array [2]` is - 17  $\Rightarrow$  `(my_array + 2)`

`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**



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

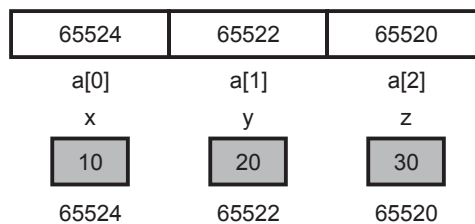
### Review Question

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

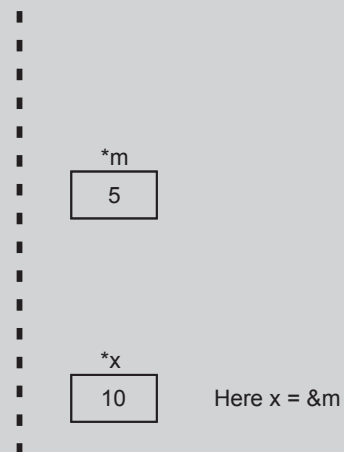
A pointer variable can be passed as an argument to the function. This method of parameter passing is called **call by reference**. Following C++ illustrates how to pass pointer as an argument to the function

```
/******
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

```

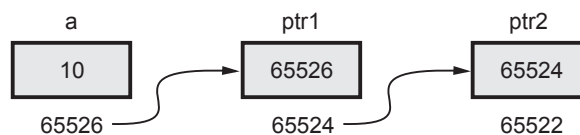


Fig. 2.25.1 Pointer to Pointer

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 *bloddgrp = 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++;
                        PersonClass::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

