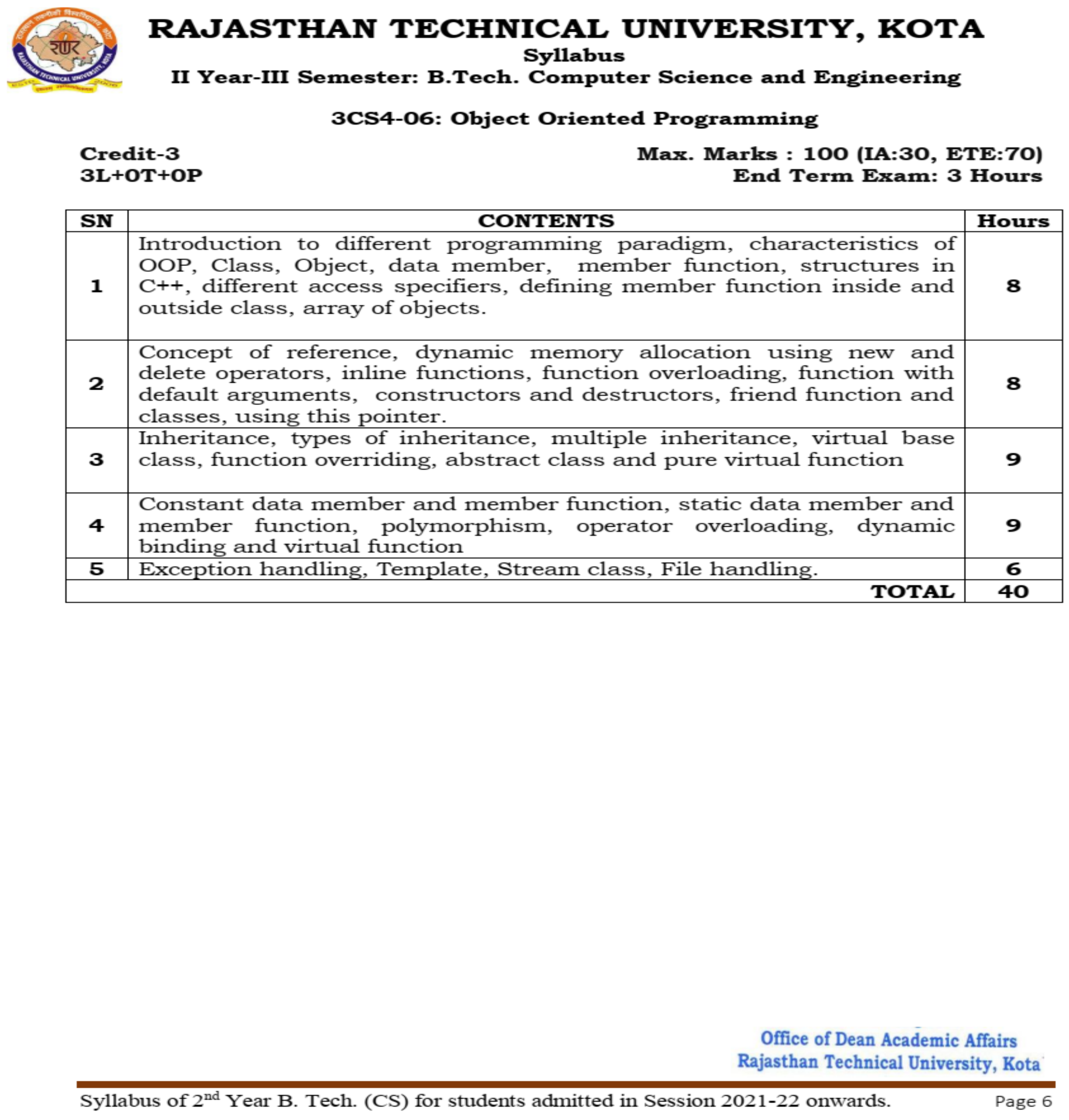
RTU Syllabus



RAJASTHAN TECHNICAL UNIVERSITY, KOTA
Syllabus
II Year-III Semester: B.Tech. Computer Science and Engineering

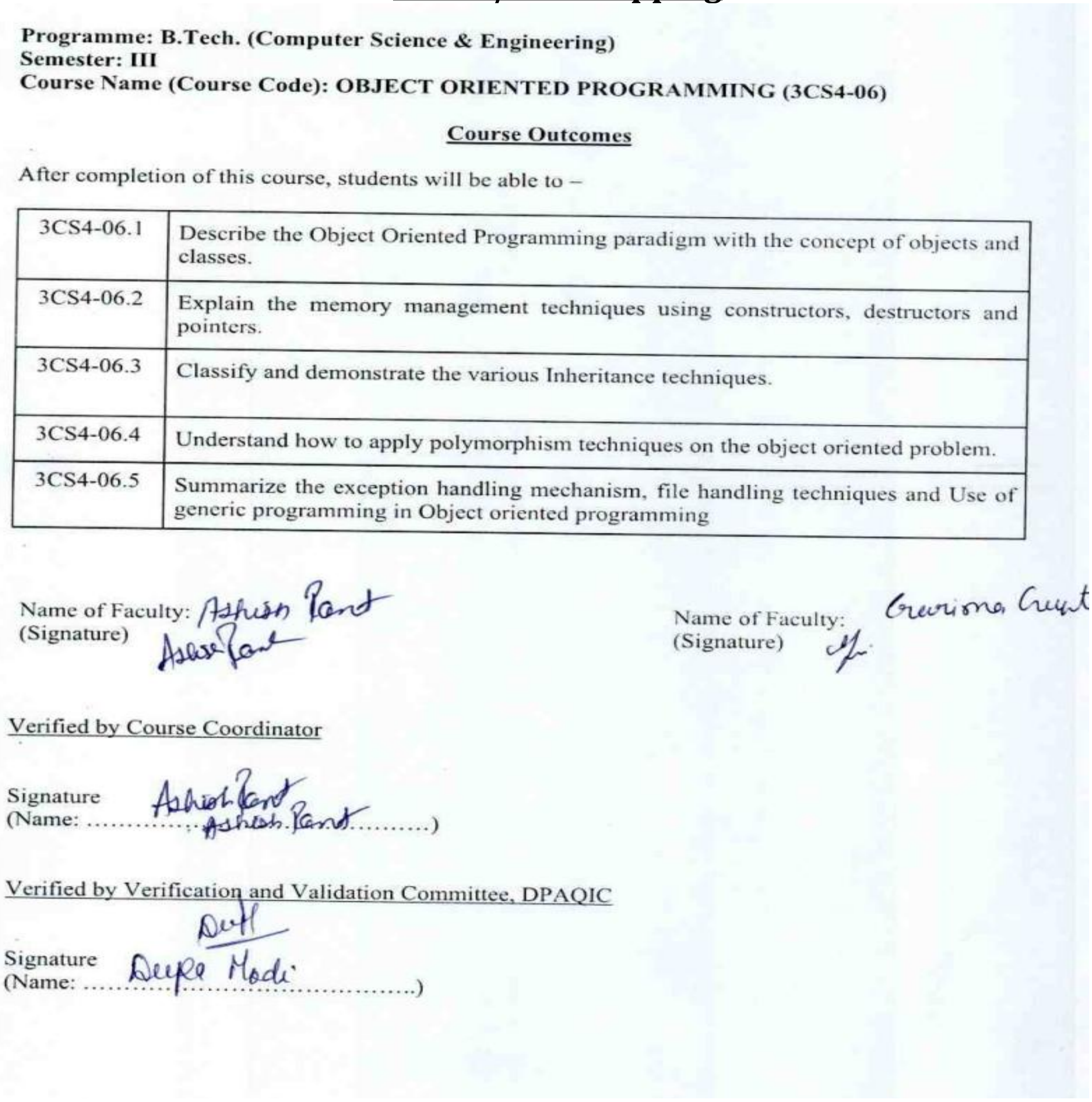
3CS4-06: Object Oriented Programming

Credit-3 **Max. Marks : 100 (IA:30, ETE:70)**
3L+0T+0P **End Term Exam: 3 Hours**

SN	CONTENTS	Hours
1	Introduction to different programming paradigm, characteristics of OOP, Class, Object, data member, member function, structures in C++, different access specifiers, defining member function inside and outside class, array of objects.	8
2	Concept of reference, dynamic memory allocation using new and delete operators, inline functions, function overloading, function with default arguments, constructors and destructors, friend function and classes, using this pointer.	8
3	Inheritance, types of inheritance, multiple inheritance, virtual base class, function overriding, abstract class and pure virtual function	9
4	Constant data member and member function, static data member and member function, polymorphism, operator overloading, dynamic binding and virtual function	9
5	Exception handling, Template, Stream class, File handling.	6
	TOTAL	**40**

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

CO-PO/PSO Mapping



Programme: B.Tech. (Computer Science & Engineering)
Semester: III
Course Name (Course Code): OBJECT ORIENTED PROGRAMMING (3CS4-06)

Course Outcomes

After completion of this course, students will be able to –

3CS4-06.1	Describe the Object Oriented Programming paradigm with the concept of objects and classes.
3CS4-06.2	Explain the memory management techniques using constructors, destructors and pointers.
3CS4-06.3	Classify and demonstrate the various Inheritance techniques.
3CS4-06.4	Understand how to apply polymorphism techniques on the object oriented problem.
3CS4-06.5	Summarize the exception handling mechanism, file handling techniques and Use of generic programming in Object oriented programming

Name of Faculty: Ashish Pant
(Signature) [illegible]

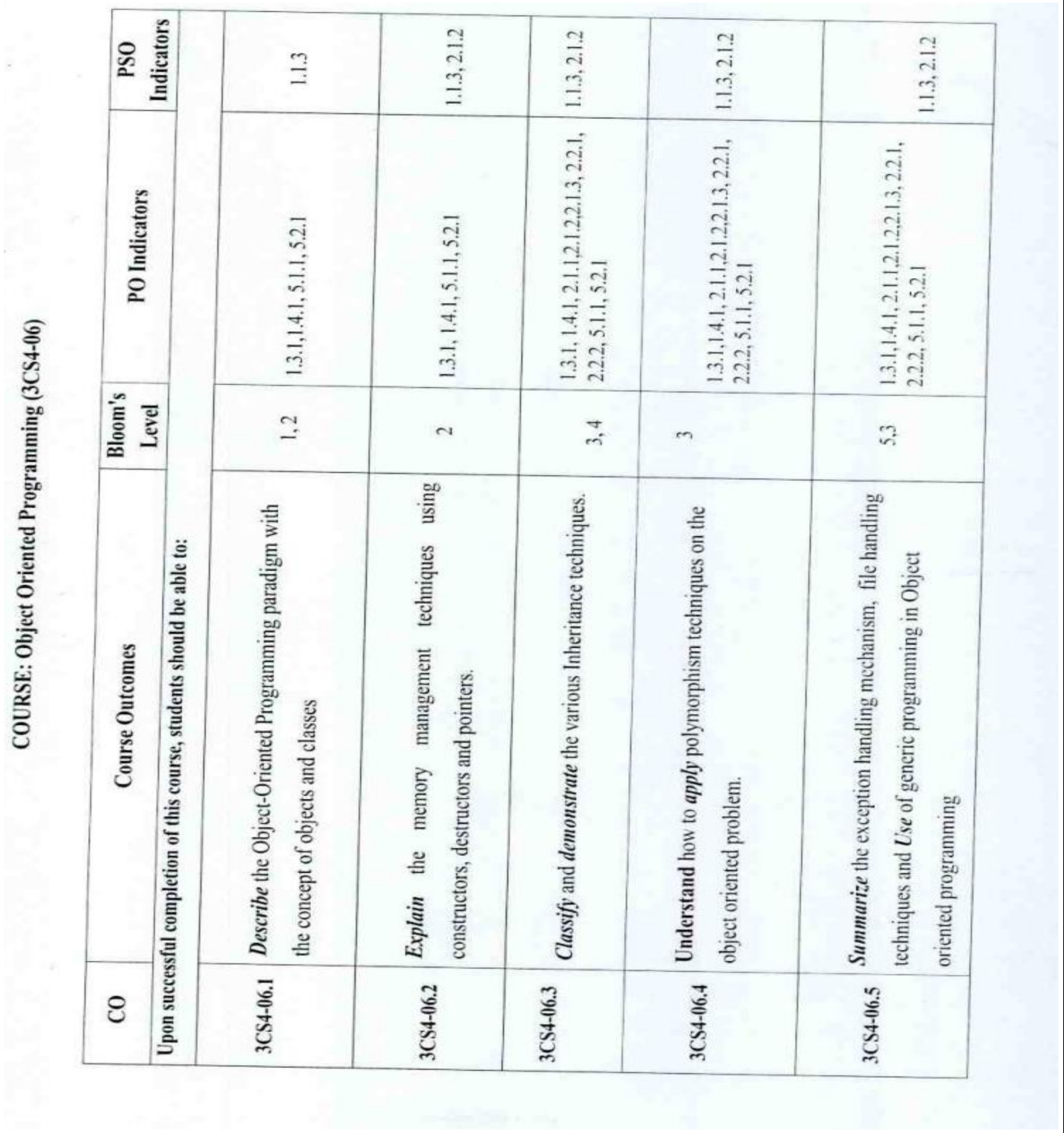
Name of Faculty: Garima Gupta
(Signature) [illegible]

Verified by Course Coordinator

Signature [illegible]
(Name: ....Ashish Pant.........)

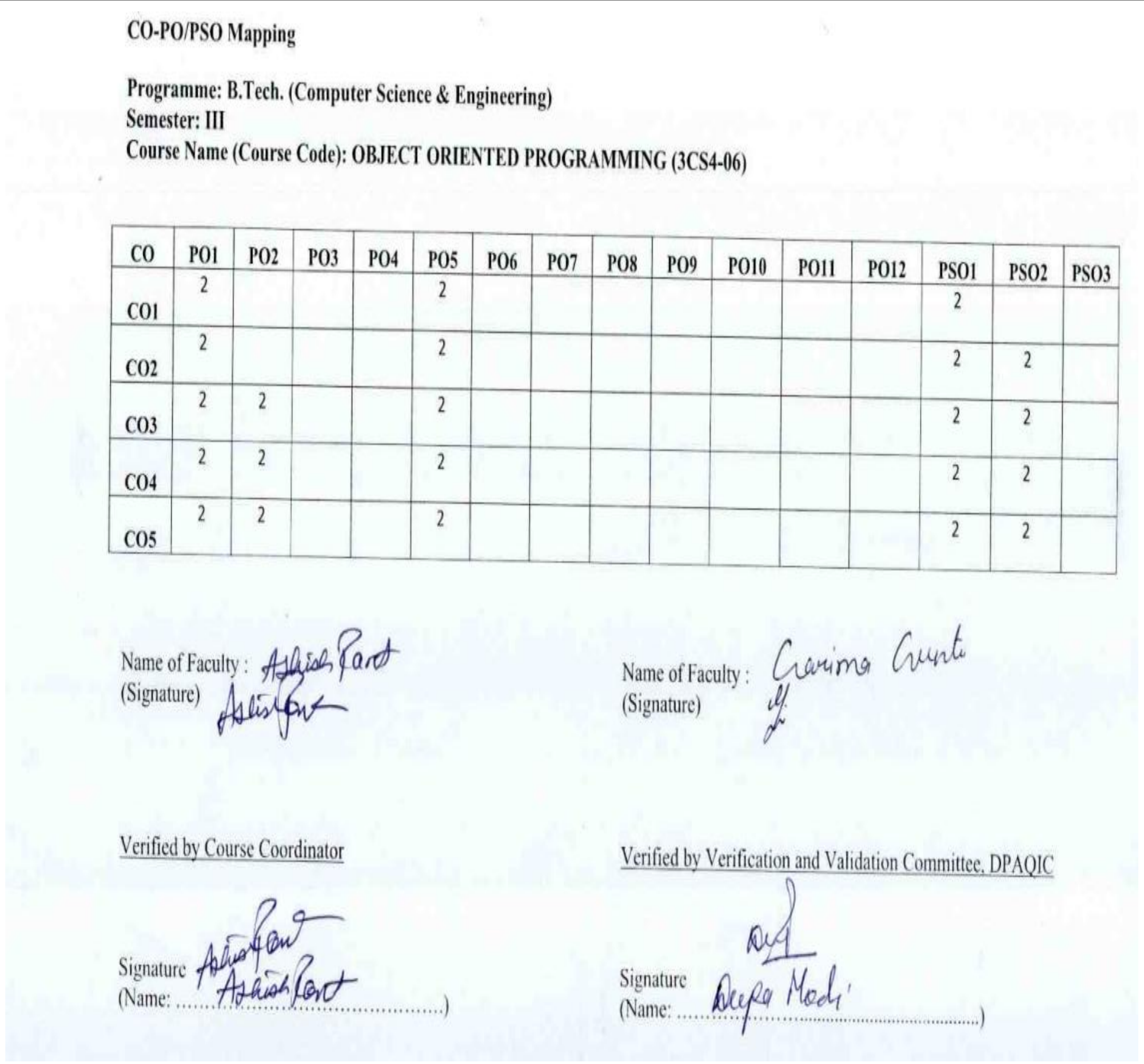
Verified by Verification and Validation Committee, DPAQIC

Signature [illegible]
(Name: ....Deepa Modi...............................)



COURSE: Object Oriented Programming (3CS4-06)

CO	Course Outcomes	Bloom's Level	PO Indicators	PSO Indicators
Upon successful completion of this course, students should be able to:				
3CS4-06.1	***Describe*** the Object-Oriented Programming paradigm with the concept of objects and classes	1, 2	1.3.1,1.4.1, 5.1.1, 5.2.1	1.1.3
3CS4-06.2	***Explain*** the memory management techniques using constructors, destructors and pointers.	2	1.3.1, 1.4.1, 5.1.1, 5.2.1	1.1.3, 2.1.2
3CS4-06.3	***Classify*** and ***demonstrate*** the various Inheritance techniques.	3, 4	1.3.1, 1.4.1, 2.1.1,2.1.2,2.1.3, 2.2.1, 2.2.2, 5.1.1, 5.2.1	1.1.3, 2.1.2
3CS4-06.4	**Understand** how to ***apply*** polymorphism techniques on the object oriented problem.	3	1.3.1,1.4.1, 2.1.1,2.1.2,2.1.3, 2.2.1, 2.2.2, 5.1.1, 5.2.1	1.1.3, 2.1.2
3CS4-06.5	***Summarize*** the exception handling mechanism, file handling techniques and ***Use*** of generic programming in Object oriented programming	5,3	1.3.1,1.4.1, 2.1.1,2.1.2,2.1.3, 2.2.1, 2.2.2, 5.1.1, 5.2.1	1.1.3, 2.1.2



CO-PO/PSO Mapping

Programme: B.Tech. (Computer Science & Engineering)
Semester: III
Course Name (Course Code): OBJECT ORIENTED PROGRAMMING (3CS4-06)

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	2				2								2		
CO2	2				2								2	2	
CO3	2	2			2								2	2	
CO4	2	2			2								2	2	
CO5	2	2			2								2	2	

Name of Faculty : Ashish Pant
(Signature) [illegible]

Name of Faculty : Garima Gupta
(Signature) [illegible]

Verified by Course Coordinator

Signature [illegible]
(Name: Ashish Pant)

Verified by Verification and Validation Committee, DPAQIC

Signature [illegible]
(Name: Deepa Modi)

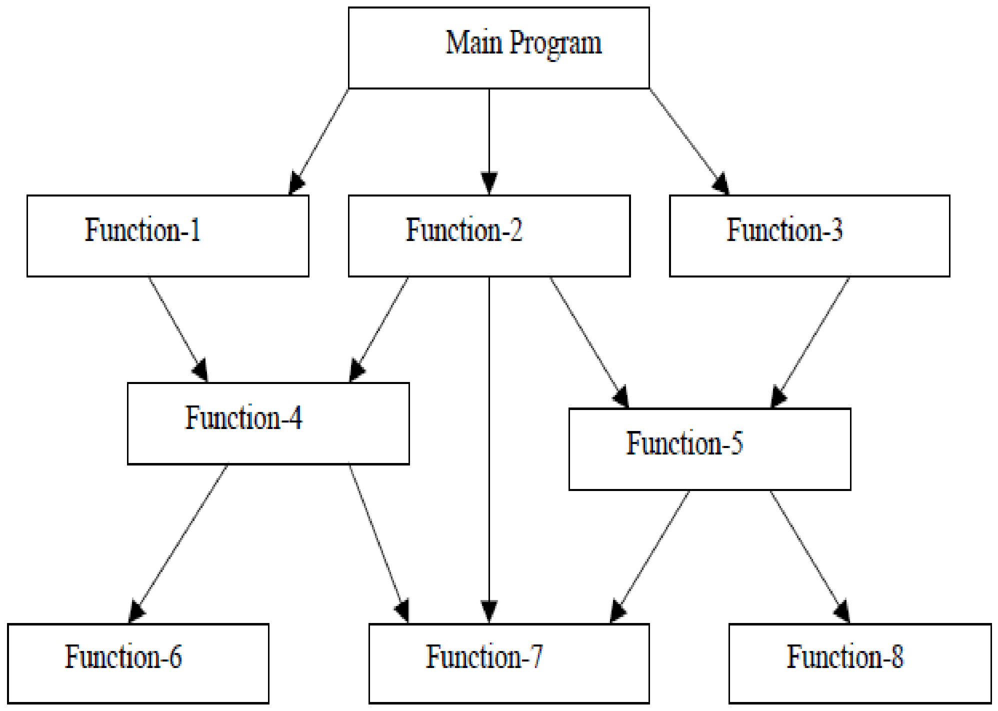
Course Notes

UNIT - I

Introduction to different programming paradigms

1) Procedure Oriented Programming (POP) Paradigm

- Conventional programming, using high level languages such as COBOL, FORTRAN & C is known as procedure oriented programming.
- In the procedure oriented approach, the problem is viewed as a sequence of things to be done such as reading , calculating and printing.
- Primary focus is on ***functions*** which are written to accomplish these tasks.
- Procedure oriented programming basically consists of writing a list of instruction (or actions) for the computer to follow and organizing these instructions into groups known as ***functions***.
- Flowcharts are used to organize the actions to represent the flow of control from one instruction to another.
- In multi-function program, many important data items are placed as ***global*** so that they may be accessed by all the functions. Each function may have its own ***local*** data.
- It follows ***top down approach*** in program design.
- It is also known as **structured** programming.

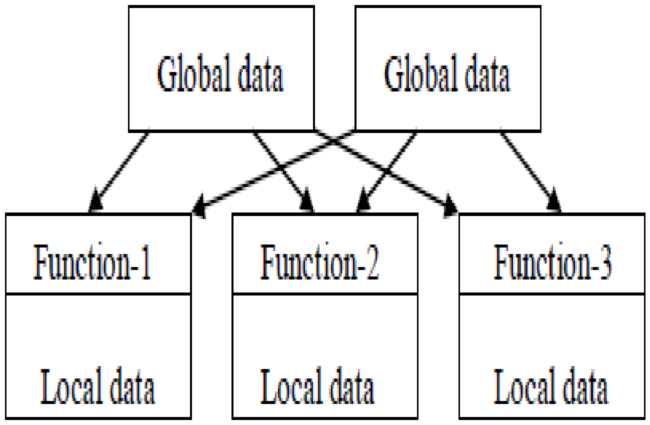


Structure of procedure oriented programs

Disadvantages of procedure oriented programming

➢ Global data is vulnerable to change by functions.

- It does not model real world problems very well as functions are action oriented and do not really correspond to the elements of the problem.
- No data hiding.



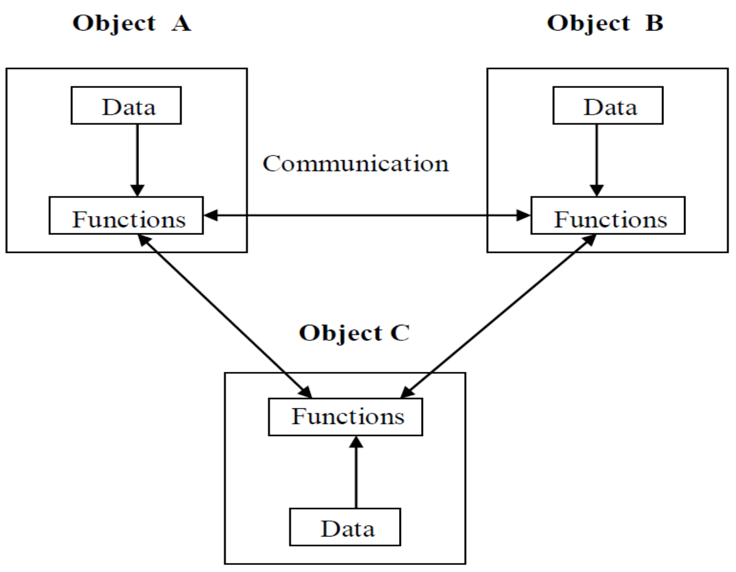
Relationship of data and functions in procedural programming.

Characteristics (features) of procedure-oriented programming

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

2) Object Oriented Programming (OOP) Paradigm

- Object-oriented programming (OOP) is a programming paradigm based upon ***objects*** (having both ***data*** and ***functions***) that aims to incorporate the advantages of ***modularity*** and ***reusability.***
- It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. Hence it treats data as a critical element in the program development and does not allow it to flow freely around the system.
- OOP allows decomposition of a problem into a number of entities called ***objects*** and then builds ***data*** and ***function*** around these objects.
- The data of an object can be accessed only by the function associated with that object.
- Function of one object can access the function of other objects.
- It follows ***bottom up approach*** in program design.



Organization of data and functions in OOP

Characteristics (features) of object-oriented programming

• Emphasis is on data rather than procedure.
• Programs are divided into what are known as objects.
• Data structures are designed such that they characterize the objects.
• Functions that operate on the data of an object are tied together in the data structure.
• Data is hidden and cannot be accessed by external functions.
• Objects may communicate with each other through functions.
• New data and functions can be easily added whenever necessary.
• Follows ***bottom up*** approach in program design.

Basic Concepts of Object Oriented Programming

- **Object**
- **Class**
- **Encapsulation**
- **Data Abstraction**
- **Inheritance**
- **Polymorphism**
- **Dynamic binding**
- **Message passing**

Object

Objects are the basic run time and real world entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Objects take up space in the memory and have an associated address.

```
OBJECTS: STUDENT

DATA
        Name
        Date-of-birth
        Marks

FUNCTIONS
        Total
        Average
        Display
        .........
```

Representing an object

- ✓ Object is considered to be a partitioned area of computer memory that stores **data (properties, attributes, fields, states, characteristics, features, or variables having some values)** and set of **functions (operations, code, behavior, procedures, methods)** that can access and manipulate that data.
- ✓ An object has a state in which all of its attributes have values.
- ✓ An object is an **instance** of a class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- ✓ Object is the **variable** of type class.
- ✓ An ***attribute*** of an object is a name-value pair associated with the object.
- ✓ An ***instance variable*** defines the attributes of the object. Instance variables' types and names are defined in the class, but their values are set and changed in the object.
- ✓ Object is a runtime entity; it is created at runtime.
- ✓ Class specifies the information and objects contains the information that the class specifies.

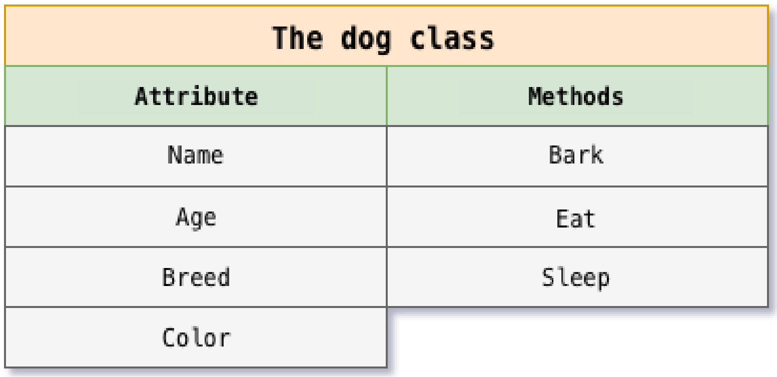
Class

- A class is a blueprint(or template) of an object and can be defined as collection of objects of similar type.
- Class is a **user defined data type** and objects are the variables of type ***class***.
- For e.g. mango, apple, and orange are objects of class fruit.
- If fruit has been defined as class, then the statement

  fruit mango;

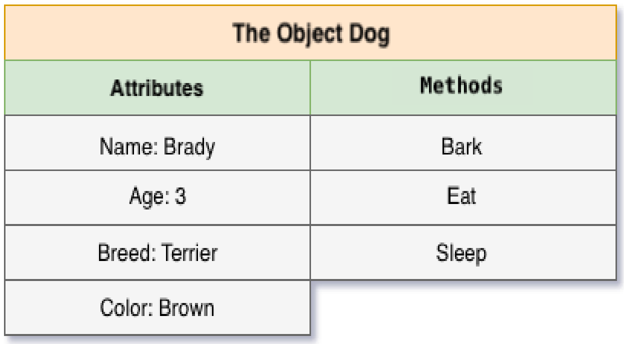
  will create an object mango belonging to the class fruit.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- Classes have logical existence while Objects have a physical, real world existence.
- A Class is a description of a group of objects with common properties (attributes), behavior (operations)

- A class serves as a **template** for all objects whose type is that class.
- A class is a user defined data-type which binds data and functions which can be accessed and used by creating an instance of that class.
- **Data members** or **member variables :** These are the variables which define the **data** (attributes/properties/fields/states/characteristics/features) the objects in a Class should have.
  Member functions : Functions (or methods/operations/code) used to manipulate these variables (data) and which defines the **behavior** of the objects in a class.
  The data members and member functions within a class are called **members of the class** or **class members.**
- Data members and member functions define the properties and behavior of the objects in a class.
- **Member variables** which are non-static and associated with objects are called **instance variables** and member variables associated with class are called **class variables or static member variables** or **static data members.**
- Objects of class holds separate copies of data members. We can create as many objects of a class as we need.
- All the instances or objects share the attributes(data) and the behavior(function) of the class. But the values of those attributes, i.e. the state are unique for each object.

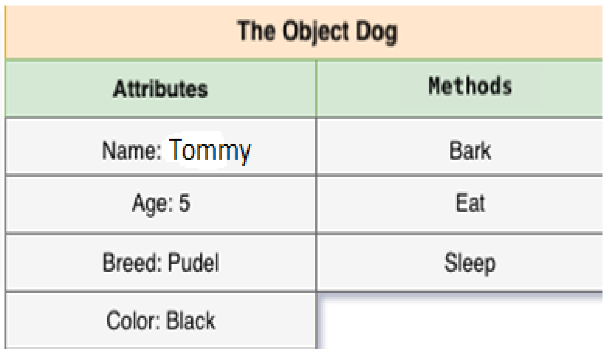


The dog class	
Attribute	**Methods**
Name	Bark
Age	Eat
Breed	Sleep
Color	

object : dog1 **object : dog2**



The Object Dog	
Attributes	**Methods**
Name: Brady	Bark
Age: 3	Eat
Breed: Terrier	Sleep
Color: Brown	

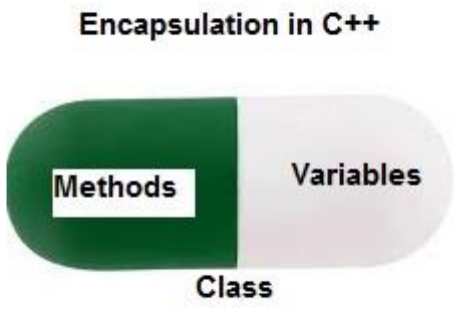


The Object Dog	
Attributes	**Methods**
Name: Tommy	Bark
Age: 5	Eat
Breed: Pudel	Sleep
Color: Black	

Example of Class and objects in OOP

Encapsulation

- The wrapping up of data and function into a single unit (called class) is known as ***encapsulation***.
- Class is the best example of encapsulation as the data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- Encapsulation leads to ***data hiding*** or ***information hiding*** as it provides insulation of the data from direct access by the program.
- Encapsulation can be implemented using class and **access specifiers** (modifiers) i.e. private, protected and public.
- Encapsulation can be achieved by declaring all the variables (data members) in the class as **private** and writing **public** functions (member functions) in the class to get and display the values of variables.



- Data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

*Note : Keywords **private, public, protected** are called **access specifiers** (access modifiers/visibility labels/access labels/visibility modifiers).

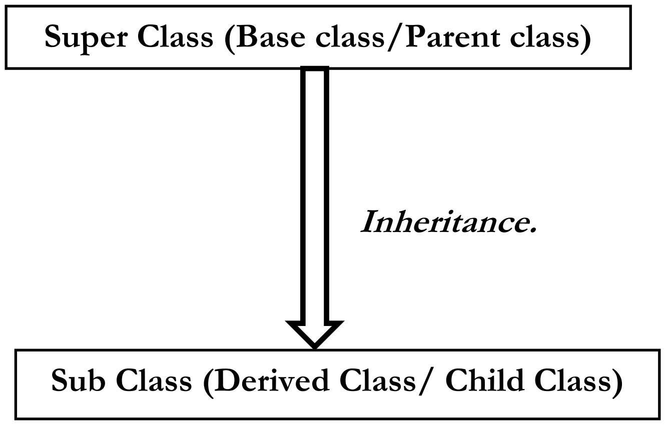
Data Abstraction

- Abstraction refers to the act of representing essential features and hiding the background (implementation) details or explanation.
- Classes use the concept of abstraction and are defined as a list of abstract attributes and function operate on these attributes. Class helps us to group data members and member functions using available access specifiers. A class can decide which class members will be visible to outside world and which is not.
  - ➢ Members declared as public in a class, can be accessed from anywhere in the program.
  - ➢ Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.
- Public member functions can be called from outside the class using object of the class without knowing the implementation details.
- One more type of abstraction can be C++ header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

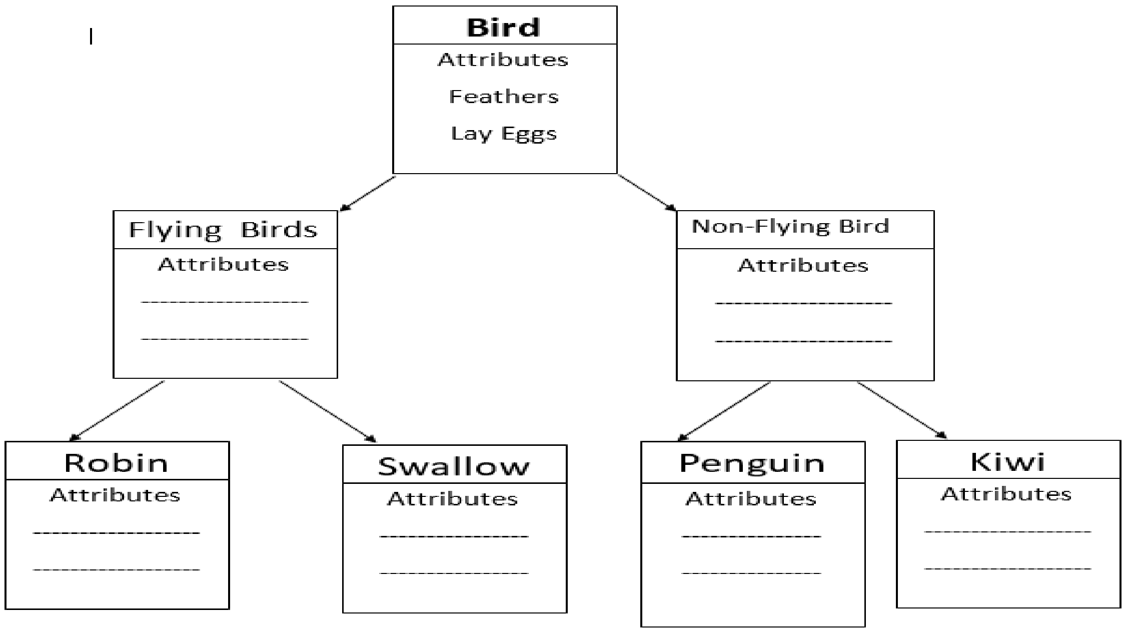
- Since the classes use the concept of data abstraction they are known as ***Abstract Data Types (ADT).***

Inheritance

- Inheritance is the process by which objects of one class acquire the properties and behavior (functionality) of objects of another class.
- **Sub Class:** The class that inherits properties from another class is called Sub class **(Derived Class/Child Class)**
- **Super Class:** The class whose properties are inherited by sub class is called Super class **(Base Class/Parent Class)**


Super Class (Base class/Parent class)
Inheritance.
Sub Class (Derived Class/ Child Class)

- Inheritance supports the concept of ***hierarchical classification***. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'.
- Each derived class shares common characteristics with the class from which it is derived.



Property Inheritance

- The derived class automatically acquires (inherits) properties and behavior of base class and has its own unique properties and behavior.
- The main advantage of inheritance is that it provides ***code reusability.*** This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. Hence programmer is able to reuse a class i.e. when child class inherits the properties and functionality of parent class, the programmer need not to write the same code again in child class. Thus, we can eliminate redundant code and extend the use of existing classes.
- Each subclass defines only those features which are unique to it, other features are inherited from super class. Without use of classification (inheritance) each class would have to explicitly include all of it features.

Polymorphism

Polymorphism, means the ability to take more than on form. Types of polymorphism:

1. **Compile time polymorphism** or **static polymorphism.**
   a) **Operator overloading :** The process of making an operator to exhibit different behaviors in different instances is known as ***operator overloading.*** An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition, i.e. **+ (plus)** operator. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.
   For e.g the plus(+) operator is overloaded as:

   16 **+** 4 = 20
   "xyz" **+** "sqr" = "xyzsqr"

   The operator << behaves as bit-wise left shift operator and output operator. Also, the operator >> behaves as bit-wise right shift operator and input operator. Thus operators >> and << can be overloaded.

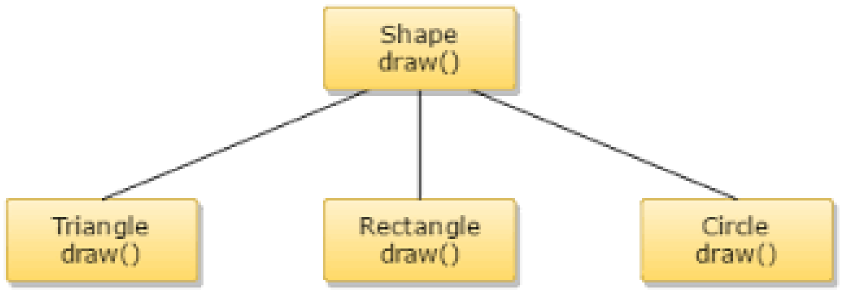
   b) **Function overloading :** Using a single function name to perform different type of task is known as function overloading. Functions to be overloaded must have the same name but have different arguments (either a different number of arguments or different type of arguments or different order of arguements).

```
int test( ) { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

   Here, all 4 functions are overloaded functions. Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. All versions of overloaded functions should be in same class or scope.

   c) **Constructor overloading :** We can have more than one constructor in a class with same name, but different arguments.

2. **Run time polymorphism or Dynamic Polymorphism**
   It is achieved by **function overriding** with inheritance**,** using **virtual function**. When child class inherits a function , which is already present in the parent class and then overrides it, that is defines it to provide its own specific implementation, then this is called function (method) overriding.
   For example, child classes Triangle, Rectangle, and Circle inherits draw() function from parent class and each class overrides draw() function to provide its own specific implementation. It is also known as **late binding** or **dynamic binding.**



Polymorphism

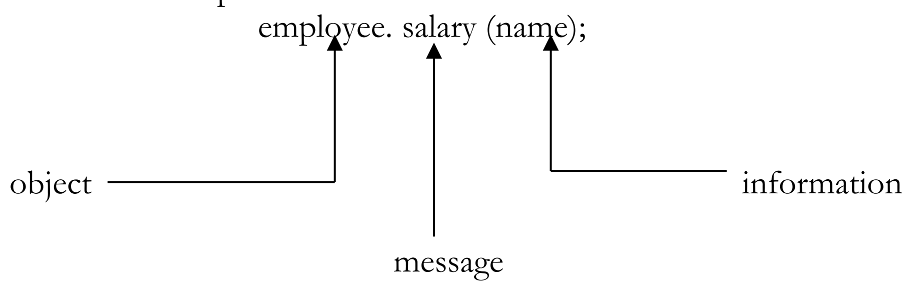
Dynamic binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.
- It provides run time polymorphism and is associated with inheritance.
- Consider the function “draw” in above figure. By inheritance, every object will have this function. Its algorithm is, however, unique to each object and so the draw function will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.
- It is also called **late binding** or **run time binding.**
- It is implemented by using **virtual function** and **function overriding.**

Message Passing

- An object-oriented program consists of a set of objects that communicate with each other.
- Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.
- A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results.

- ***Message passing*** involves specifying the name of object, the name of the function (message) and the information to be sent. Example:



- Object has a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Introduction to C++

- C++ is an object-oriented programming language.
- It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's.
- Therefore, C++ is an extension of C with a major addition of the class, object and object oriented programming features.
- C++ is a superset of C. Almost all programs are also C++ programs.
- The most important features that C++ adds on to C are classes and objects, inheritance, data encapsulation, data abstraction, and polymorphism(function overloading and operator overloading).
- The idea of C++ comes from the C increment operator **++**, thereby suggesting that C++ is an **incremented** version of C.

A Simple C++ Program

```
#include<iostream>

int main()
{
    int a,b;
    std::cout<<"Enter two numbers\n";
    std::cin>>a>>b;
    int sum;
    sum=a+b;
    std::cout<<"Here is the output:"<<std::endl;
    std::cout<<"The sum of "<<a<<" and "<<b<<" is "<<sum;
    return 0;
}
```

- Data is transferred to/from output/input device in the form of a sequence of bytes called stream.
- The #include directive, #include <iostream> instructs the compiler to include the contents of the header file **iostream** (input output stream) enclosed within angular brackets into the source file or program.
- The **iostream** file contains code that allows a **C++** program to display **output** on the screen and read **input** from the keyboard.
- Stream objects **cin** and **cout** defined in iostream header file.
- The identifier **cin (character input** ,pronounced 'C in'**)** is a predefined **object** in C++, known as **standard input stream** , is used to read the input from the standard input device i.e. keyboard.

The object **cin** is connected to the standard input device i.e. keyboard.
The "**stream extraction"** operator **">>"**, also known as **"get from"** operator or **"input"** operator is used with object **cin** to read data from the standard input device i.e. keyboard.
The **extraction operator >>** extracts (or takes) the value (entered using keyboard) from object **cin** in its left, and assigns it to the variable on its right.

- The identifier **cout (character output** ,pronounced 'C out'**)** is a predefined **object** in C++, known as **standard output stream** , is used to produce or display output in the standard output device i.e. screen.
  The object **cout** is connected to the standard output device i.e. screen.
  The "**stream insertion"** operator **"<<"**, also known as **"put to"** operator or **"output"** operator is used with object **cout** to display data on standard output device i.e. screen.
  The **insertion operator <<** inserts (sends) the contents of variable or data on its right to the cout object on its left, which is then displayed on screen.
- A namespace is a special area inside which something is defined. It is a declarative region that provides a scope to the identifiers (the **names** of types, functions, variables, etc.) inside it. **std** is the standard namespace in which cin and cout and other C++ standard library classes, objects and functions are defined. The **::** symbol is an operator called the **scope resolution operator**. The identifier to the left of the **::** symbol identifies the namespace that the name to the right of the :: symbol is contained within.
- If we don't wish to add prefix **std::** , again and again before cin or cout and other elements of standard c++ libraries which are defined in **std** we can include using directive at starting and then simply write cin and cout in our program, not std::cin or std::cout
  using and **namespace** are keywords in C++

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

int main()
{
    int a,b;
    cout<<"Enter two numbers\n";
    cin>>a>>b;
    int sum;
    sum=a+b;
    cout<<"Here is the output:"<<endl;
    cout<<"The sum of "<<a<<" and "<<b<<" is "<<sum;
    return 0;
}
```

- Multiple use of << or >> operators in one statement is called ***cascading.***
- In input statement **cin>>a>>b,** the values are entered from left to right. That is if we input two values from keyboard 10 and 20, 10 will be assigned to a and 20 to b.

- **Line feed operator : endl**(end line) **manipulator**, is an object defined in std namespace(std::endl) which is used to insert new line, as "\n" but the it is different from "\n" as endl flushes the output stream but "\n" does not.
- Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Tokens

The smallest individual units in program are known as **tokens.** C++ has the following tokens.

i. Keywords
ii. Identifiers
iii. Constants
iv. Strings
v. Operators

Keywords

They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements.
A list of 32 Keywords in C++ Language which are also available in C language are given below.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

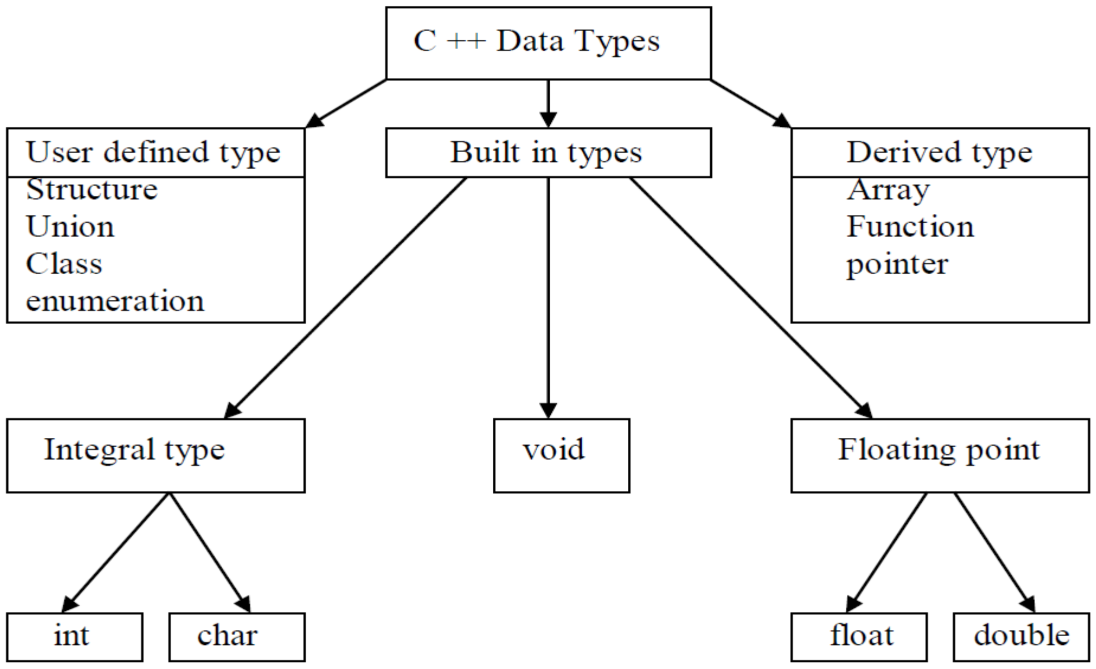
Identifiers

Identifiers refers to the name of variable , functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers.
The following rules are common for both C and C++.

1. Only alphabetic chars, digits and underscore are permitted.
2. The name can't start with a digit.
3. Upper case and lower case letters are distinct.
4. A declared keyword can't be used as a variable name.

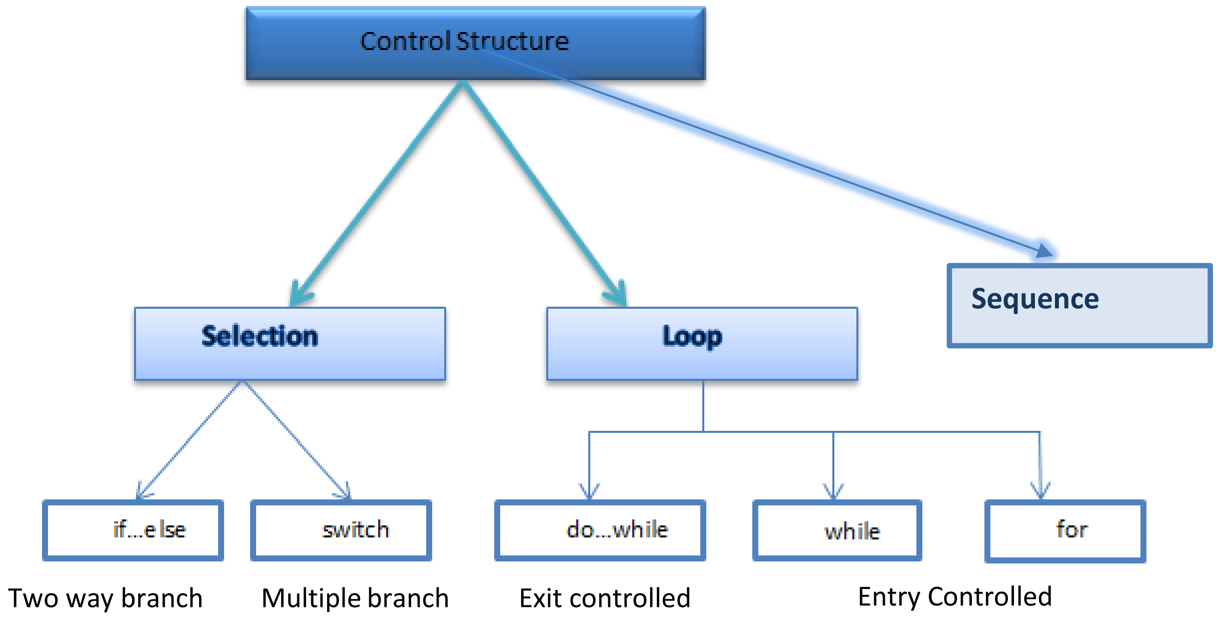
In C the maximum length of a variable is 32 chars but in C++ there is no bar.

Basic Data Types in C++



- In C++, the size of the character array should be one larger than the number of characters in the string.
- **const** is a qualifier (keywords which are used to modify the properties of a variable, i.e to make it read only)
  const int size = 10 ; char name [size] ; would be illegal in C but valid in C++
  const size=10; Means const int size =10;
- size of character constant is equivalent to size of integer(4 bytes) in C and size of char (1 byte) in C++.

Control Structures in C++



- Sequence structure (straight line)
- Selection structures (branching)
- Loop structures (iteration or repetition)

Scope Resolution in C++

In C++, scope resolution operator is :: . It is used for following purposes.

1) To access a global variable when there is a local variable with same name:

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

int x=10;     // Global x
         //this a is defined in global namespace

       //which means, its scope is global. It exists everywhere

namespace N

{

     int x=20;  //it is defined in a non-global namespace called `N`

             //outside N it doesn't exist.

}
```

```
int main()
{
  int x = 30; // Local x
  cout << "Value of global x is " << ::x; //if no namespace is mentioned before ::
                                           it denotes global namespace, i.e global
                                           scope.

  cout << "\nValue x in namespace N is " <<N::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Output: Value of global x is 10

Value x in namespace N is 20

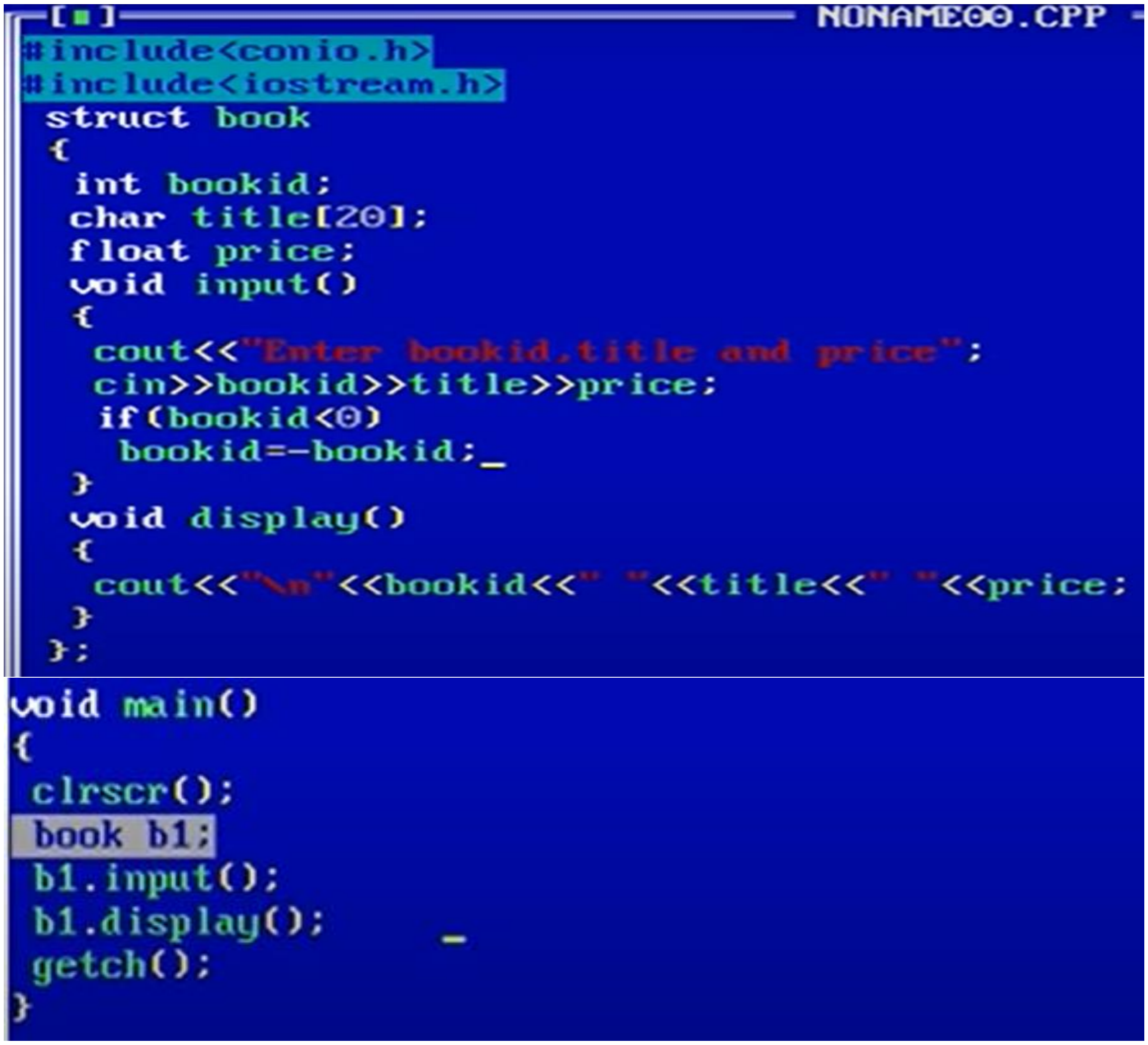
Value of local x is 30

Structure in C++

Differences between C structures and C++ structures

C Structure	C++ Structure
Structures in C, cannot have member functions inside structures. (No encapsulation)	Structures in C++ can hold member functions with member variables.
We cannot initialize the structure data directly in C.	We can directly initialize structure data in C++.
In C, we have to write 'struct' keyword to declare structure type variables.	In C++, we do not need to use 'struct' keyword for declaring variables.
C structures cannot have static members.	C++ structures can have static members.
The data hiding feature is not available in C structures.	The data hiding feature is present in C++ structures.
C structures does not have access modifiers.	C++ structures have access specifiers.

A simple C++ structure program



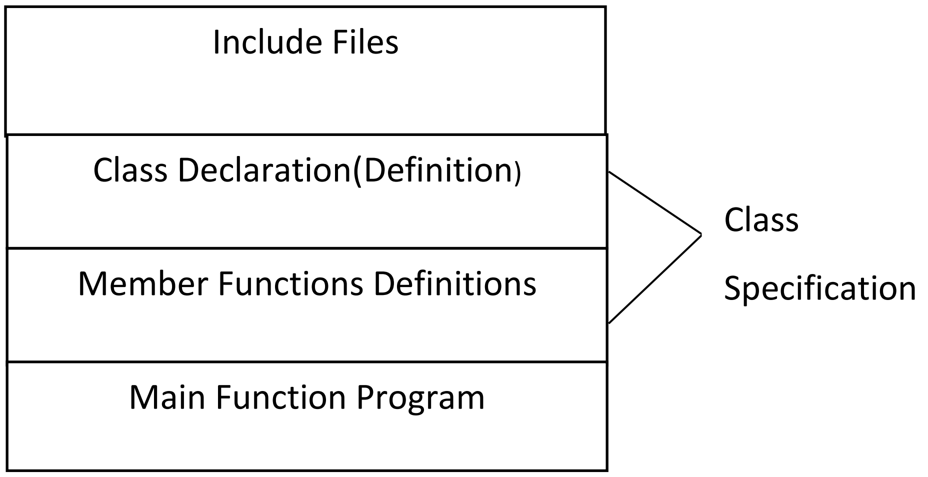
```
#include<conio.h>
#include<iostream.h>
 struct book
 {
  int bookid;
  char title[20];
  float price;
  void input()
  {
   cout<<"Enter bookid,title and price";
   cin>>bookid>>title>>price;
   if(bookid<0)
    bookid=-bookid;
  }
  void display()
  {
   cout<<"\n"<<bookid<<" "<<title<<" "<<price;
  }
 };
void main()
{
 clrscr();
 book b1;
 b1.input();
 b1.display();
 getch();
}
```

Structure vs class in C++

- First difference between a **struct** and **class** in C++ is the default accessibility of member variables and functions. In a **struct** they are public; in a **class** they are private
- Another difference is, when deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.
- In C++, a class defined with the class keyword has private members and base classes by default. A structure is a class defined with the **struct** keyword. Its members and base classes are **public** by default.

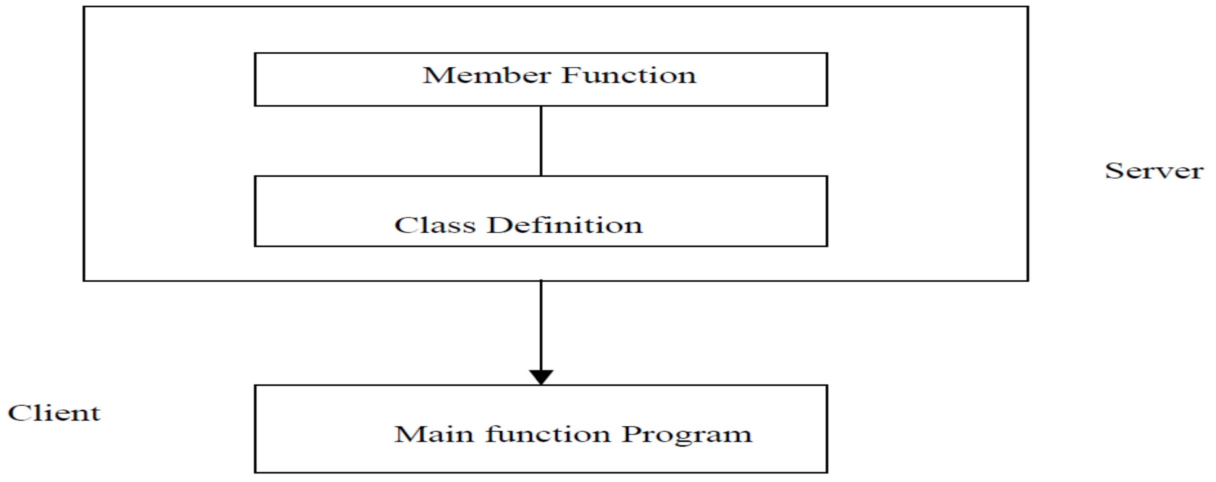
Classes and Objects in C++

Structure of C++ Program



- It is common practice to organize a program into three separate files.
- The class declarations/definition(data member declaration & member functions declaration) are placed in a header file.
- The member function definitions are kept in another header file.
- Finally , the main program that uses the class is placed in a third file which "includes" the previous other files as well any other files required
- This approach enables the programmer to separate abstract specification of the interface(class definition) from the implementation details(member functions definition).
- This approach is based on the concept of **client-server model** as shown in figure below. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

The client-server model



Specifying a Class (Creating a Class)

- A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use.
- While defining a class, we are creating a new ***abstract data type*** that can be treated like any other built in data type.
- A class specification has two parts:
  1. Class declaration(definition)
  2. Class function definitions.
- The **Class declaration** describes the type and scope of its members.
- The **class function definition** describes how the class functions are implemented.
- The general form of a class declaration(definition) is :

```
class class-name
{
        private: //optional
                variable declarations;
                function declaration ;
        protected:
                variable declarations;
                function declaration ;
        public:
                variable declarations;
                function declaration;
};
```

- The keyword class specifies, that what follows is an abstract data of type **class_name.**
- The **body** of a class is enclosed within braces & terminated by semicolon. The **class body** contains the declaration of variables and declaration(and/or definition) of functions.
- The class function definition can be done in two ways:
  1. Declaring member function inside the class and defining it outside the class definition. Such members functions are called **external member functions.**
  2. Defining the member function directly inside the class (replacing the function declaration by actual definition inside the class definition).
- The variables declared inside the class are known as **data members** or **member variables** and functions declared(or defined) inside class are called **member functions.**
- The variables and functions are collectively called **class members** or **members of the class.**
- A non-member function always appears outside of a class and cannot access private and protected members of class. (Friend function is exception)
- The data members are usually declared as private and member functions as public.
- The binding of data and functions into a single class-type variable is referred to as **encapsulation.**

- There are two types of class members, static members and non-static members. Non static members are called instance members as they are associated with class instances(objects) and static members are declared as static and associated with the class.
- Non static data members (member variables) are called instance data members or instance variables and static data members are called class variables.
- Non static member functions are called instance member functions and static member functions are called class member functions.
- If nothing is specified members of class are instance members.

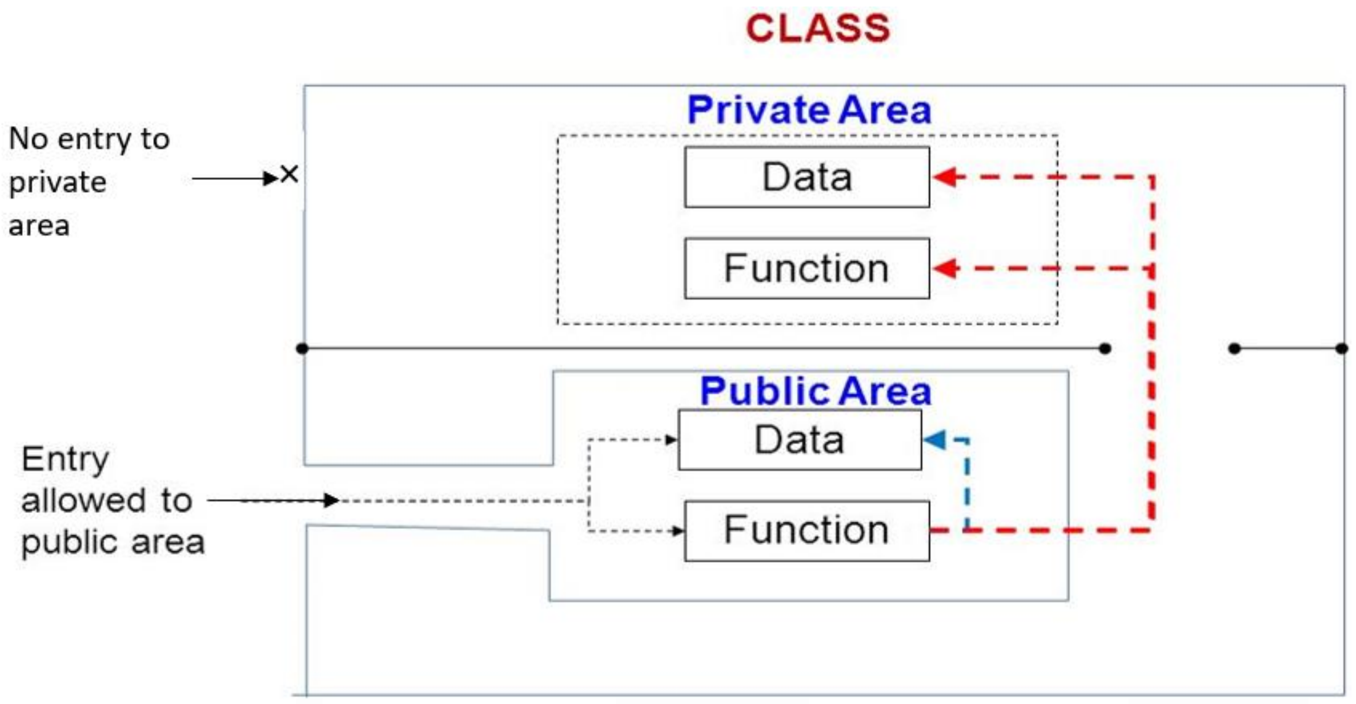
Access Specifiers

```
class class-name
{
                                    //private by default
            variable declarations;
            function1 declaration ; // visible to (accessible by) member functions within its class.
            function2 definition

      protected:
             variable declarations; // visible to (accessible by) member functions within its class & any
             function3 declaration ;  //class immediately derived from it
             function4 definition
      public:
            variable declarations;  //visible to (accessed from) outside the class also.
            function5 declaration;
            function6 definition
};
```

- The keywords **private, public,** and **protected** are used to set the accessibility of the class members These keywords are called **access specifiers** (access modifiers/visibility labels/access labels/visibility modifiers).
- There are three access specifiers available in C++ **private, public, protected**
- Class members are grouped under three labeled sections, namely, private, protected, and public to denote which of the members are private and which of them are public and protected.
- Private access modifier is used to implement an important feature of Object-Oriented Programming known as **data hiding.**
- If we do not specify any access modifiers for class members then by default the access modifier for the class members will be **private**.
- **public :** The data members and member functions declared public can be accessed from outside class also i.e. the public members of a class can be accessed from anywhere in the program.
- **private :** The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions are allowed to access the private data members or private functions of a

class. Friend function or friend class can also access private members of the class in which it is declared as friend.

- **protected :** The class members declared as protected are only accessible by member functions within its class & any class immediately derived from it. Friend function or friend class can also access protected members of the class in which it is declared as friend.



Data hiding in classes.

A simple Class Example

Class declaration is done as:

```
class item
{
        int number; //number and cost are instance variables as they are non-static
        float cost;
    public :
        void getdata(int, float); //assigns values passed to number and cost
        void putdata( );          //displays values of data members i.e. number and cost
                                  //Both functions are instance member function as they are non-static
};
```

Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition. (External Member Function)
- Inside the class definition.

Outside the Class Definition (External Member Function)

- Member functions that are declared inside a class have to be defined separately outside the class.
- The difference between an external member function and normal function is that the external member function has **membership identity label (class-name : :)**, in the header. This label tells the compiler which class the function belongs to.
- The general form of external member function definition is :

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

- **: :** is called scope resolution operator. Scope of the function is restricted to class-name specified in header line.
- The above declared member functions getdata( ) and putdata( ) can be defined outside class as follows:

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

void item : : putdata( )
{
        cout << "Number : " << number << "\n ";
        cout << "Cost       : " << cost << "\n ";

}
```

Inside the Class Definition

- The member function can be directly defined inside the class without declaration.
- If there are small functions they can be defined inside the class. When a member function is defined inside a class it behaves as **inline functions** and all the restrictions and limitation that apply to an inline function are also applicable here.

We can define above putdata( ) function inside the class as follows:

```
class item
{
        int number;
        float cost;
    public:
        void getdata(int a, float b);           //declaration of external member function.
        void putdata( )                         //definition inside the class
        {
                cout << "Number : " << number << "\n ";
                cout << "Cost      : " << cost << "\n ";
         }
};
```

Characteristics of member functions

- Several different classes can have same function name. The 'membership label' will resolve their scope.
- Member functions can access the private members (data members) of the class. A non-member function cannot do so (friend function is an exception)
- A member function can call another member function directly, without using the dot operator.

Creating Objects

- Class variables of type class are known as **objects.** An **Object** is an instance of a class.
- item x; // memory for x is created, object x is declared (or defined)

  creates a variable **x** of type **item,** i.e. object x of class item is created. **x** is called object of type **item.**
- The necessary memory space is allocated to an object at this stage.

- Private members a class can be accessed only through the member functions of that class. They are not allowed to be accessed directly by any object or function outside the class.
  x **.** cost = 500   //error, cost is private

Accessing Class Members

- The public members of class can be directly accessed by object of the same class , using the dot('.') operator **(Class Member Access Operator)**. Format for calling a public member function:

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

  x . getdata(100, 250.50);   //assigns 100 and 250.50 to number and cost (data members) of object x in the implementation
  x.putdata( );   //will display value of data members (i.e. number and cost) of object x

public data member (variable) can be assigned value as :
object-name . data-member-name = value;

- Objects communicate by sending and receiving messages. This is achieved through member functions. For example,

  x **.** putdata( );
  sends a message to the object x requesting it to display its contents.
- An object has some state at particular time. State of an object is the collection of values of all instance variables of that object.
- Behavior of an object is collection of all instance member functions associated with that object.
- When an object calls its public member function by dot operator, the member function can directly access private data members associated with that object (and set, change, input, use, display, their values). Hence object cannot access its data members (as they are kept private) directly by itself but indirectly through its member functions.
- If object's state has to be changed it can be changed by its behavior (member function) only(e.g getdata( ))

Memory allocation for objects

- There are two parts of memory in which an object can be stored:

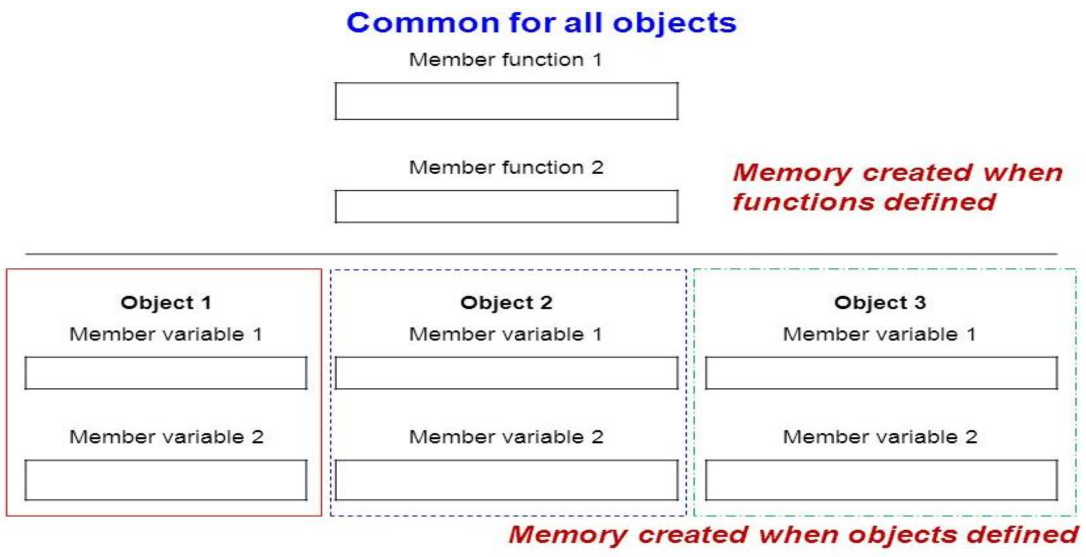
1. **stack** – Memory from the stack is used by all the members which are declared inside blocks/functions. Note that the main is also a function. This memory is allocated(reserved) automatically by compiler when object is created. Stack based **objects** are implicitly managed by **C++ compiler.** Such memory allocation is called compile time or static memory allocation. Note that memory for object is only reserved by compiler as it knows the size at compile time. Actual memory allocation always takes place at run time.
2. **heap** – This memory is unused and can be used to manually allocate the memory at runtime. Such memory allocation is called dynamic memory allocation or run time memory allocation.

- The scope of the object created inside a block or a function is limited to the block in which it is created.

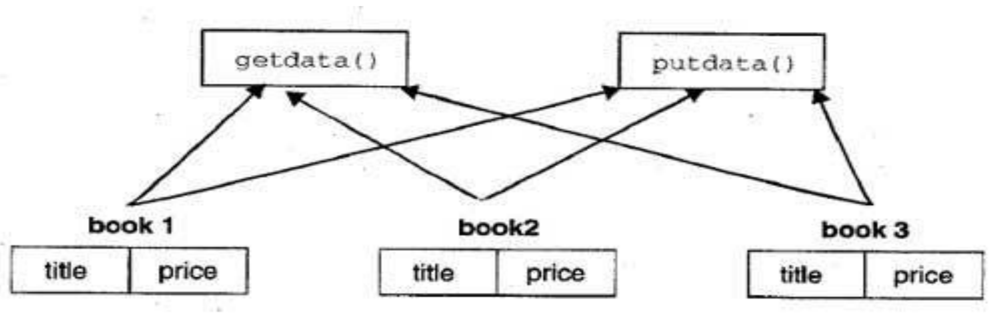
✓ The object created inside the block will be stored in the **stack** and Object is destroyed (destructor will be called automatically) and removed from the stack when the function/block exits.
✓ But if we create the object manually at runtime i.e. by **dynamic memory allocation (using new operator)** then the object will be stored on the **heap**. In this case, we need to explicitly destroy the object using **delete** operator which will call the destructor to deallocate memory.

- The class specification provides only a **template** and does not create any memory space for the objects.
- Object take up space in memory and have an associated address.
- The member functions for all the objects of same class are common but values of data members will be different for different objects.

- Memory space for objects (data members of objects) is allocated when they are declared(created), not when the class is created(specified).
- Memory space for member functions is created only once, when they are defined as part of class creation(specification). When objects are created, no separate space is created for its member functions, since all objects of a class use the same member functions. All objects of same class share the common memory space for member functions.
- Only space for member variables is allocated separately for each object during its creation. Separate memory location for each object is needed, because member variables will hold different values for different objects.

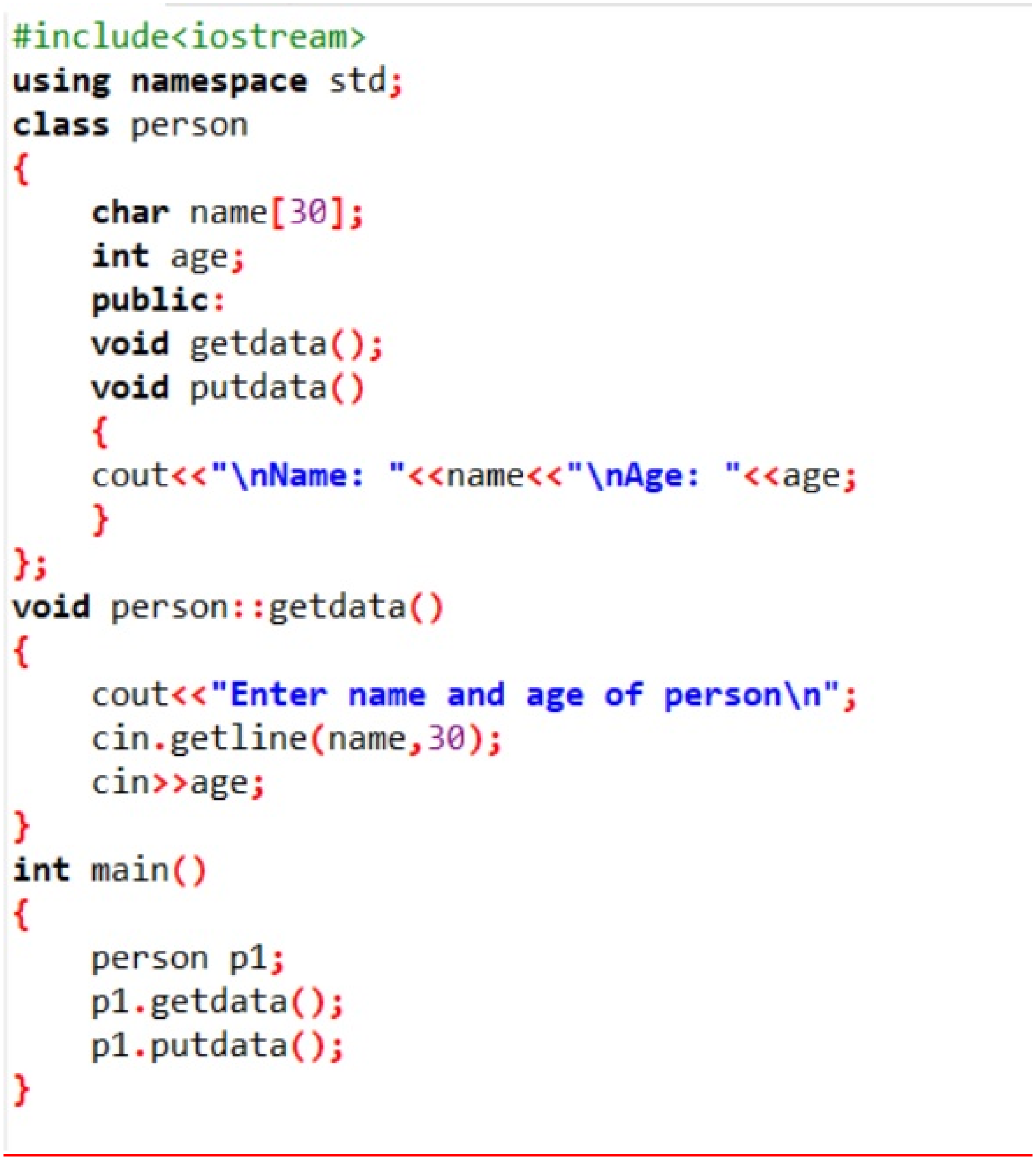


Memory allocation for objects

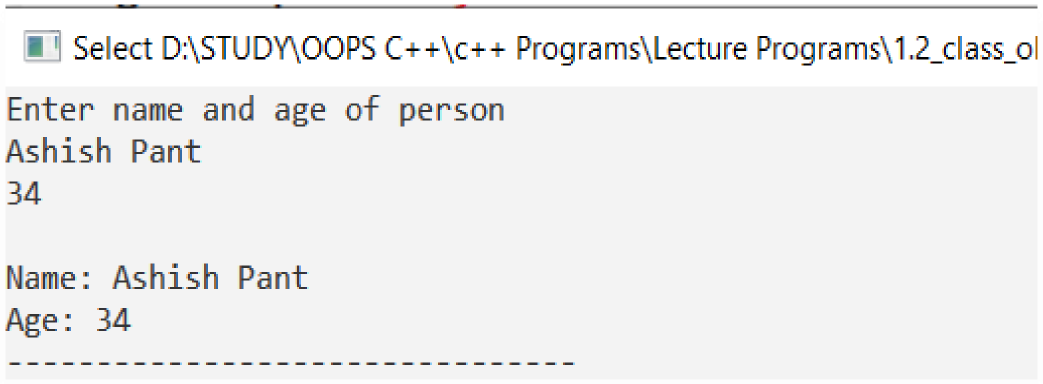


Memory Allocation for the Objects of the Class book

A simple C++ program to show the use of class and objects



```
#include<iostream>
using namespace std;
class person
{
    char name[30];
    int age;
    public:
    void getdata();
    void putdata()
    {
    cout<<"\nName: "<<name<<"\nAge: "<<age;
    }
};
void person::getdata()
{
    cout<<"Enter name and age of person\n";
    cin.getline(name,30);
    cin>>age;
}
int main()
{
    person p1;
    p1.getdata();
    p1.putdata();
}
```



Select D:\STUDY\OOPS C++\c++ Programs\Lecture Programs\1.2_class_ol

```
Enter name and age of person
Ashish Pant
34

Name: Ashish Pant
Age: 34
--------------------------------
```

Array of Objects

- An array of objects is an array of type class whose elements are the objects of that class.
- Array of objects is the collection of objects of same class.
- The syntax for declaring an array of objects is
  `class_name array_name [size] ; //size is total objects to be created.`
- The ith element(object) of array is : `array_name[ i ]`
- i^{th} element (object) can access member function as : `array_name[ i ].putdata( );`
  This statement requests the object array_name[i] to call member function putdata( ) which will display the values of its data members.

UNIT - II

Function Overloading

- Overloading refers to the use of the same thing for different purposes
- We can use the same func functions that perform a variety of different tasks.
- Function overloading is used to implement compile time polymorphism using static binding by compiler.
- Using the concepts of function overloading , a family of functions with one function name but with different argument lists (number of arguments and/or type of arguments and/or order of arguments must be different) in the functions call.
- int test( ) { }
  int test(int a) { }
  float test(double a) { }
  int test(int a, double b) { }
  Here, all 4 functions are overloaded functions
- Overloaded functions may or may not have different return types but they must have different arguments.
- All versions of overloaded functions should be in same class or scope.
- Static Polymorphism is also known as compile time binding or early binding. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.
- Two overloaded functions must not have the same **signature**(number, order and type of its formal parameters.)
- The main advantage of function overloading is to the improve the **code readability** and allows **code reusability**
- For example, an overloaded add( ) function handles different types of data as shown below.

```
//Declaration

int add(int a, int b); //prototype 1
int add (int a, int b, int c); //prototype 2
double add(double x, double y); //prototype 3
double add(int p , double q); //prototype 4
double add(double p , int q); //prototype 5
//function call

cout<<add(5, 10); //uses prototype 1
cout<<add(15, 10.0); //uses prototype 4
cout<<add(12.5, 7.5); //uses prototype 3
cout<<add(5,10, 15); //uses prototype 2
cout<<add(0.75, 5); //uses prototype 5
```

- When a function with name fun (suppose) is called, the compiler looks for all functions having name fun. All these versions of functions having same name are the candidates. Compiler binds the function call correctly to one of the function candidate's definition (early binding) with the help of three rules

- **Exact Match :** The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.

- **Type Promotion :** If an exact match is not found the compiler uses the integral promotions to the actual arguments such as :

  char to **int**
  float to **double**

  to find a match.

- **Type Conversion :** Any primitive type of actual argument be converted to any other primitive data type to find a match. If the conversion is possible to have multiple matches, then the compiler will give error message.

  Example:
  ```
  void f1(float n)
  {
  }
  void f1(int x)
  {
  }
  ```
  A function call such as :
  ```
  f1(10.5);
  ```
  will cause an error because 10.5 will be treated as double, double argument can be converted to either float or integer .There by creating an ambiguous situation as to which version of f1( )should be used.

  ✓ If the compiler cannot choose a function amongst two or more overloaded functions, the situation is -" **Ambiguity** in Function Overloading".
  ✓ As per C++ standard, floating point literals (compile time constants) are treated as double unless explicitly specified by a suffix ***f***.
  ✓ We can simply tell the compiler that the literal is a float and NOT double by providing **suffix f**. e.g. f1(10.5f); will run without error and call the float version.

Function with Default Arguments(Parameters)

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
- C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a ***default value*** to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared.

- The compiler looks at the prototype (declaration) to see how many arguments a function uses and alerts the program for possible default values.
- Example of prototype (function declaration) with default values:

```
float amount (float principle, int time ,float rate=0.15);

                                              or

float amount (float, int ,float =0.15);
```

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

  passes the value of 5000 to **principle** and 7 to **time** and then lets the function use default value of 0.15 for **rate** .

- Default arguments are overwritten when calling function provides values for them.

  `value=amount(5000,5,0.12);` //no missing argument.

  passes an explicit value of 0.12 to **rate.**
- We must add default from right to left so that only trailing(rightmost) arguments have default values. Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. This means, we cannot provide a default to a particular argument in the middle of an argument list.
- Example:-

```
int mul(int i, int j=5,int k=10);        //legal
int mul(int i=0,int j,int k=10);         //illegal
int mul(int i=5,int j);                  //illegal
int mul(int i=2,int j=5,int k=10);       //legal (all arguments are default arguments)
```

- If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.

Inline Functions

- Each time a function is called, there is calling overhead (executing a series of instructions for task such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.
- Due to such calling overheads, for small functions switching time (time needed to make the function call.) is lot more than the actual execution time of the function code.
- One solution to this problem is to use macro definition or **macro**, which is expanded by the preprocessor before compilation. The main drawback is that they are not really functions, and

the usual error checking does not occur during compilation time. Also, macro can't access the class's data members.

- **Inline functions** in C++ provides solution to above problem.
- Inline function is a function that is expanded in line when it is called. That is, the compiler replaces all function calls with the corresponding function code (definition).
- The inline functions are a C++ enhancement feature to increase the execution time of a program by reducing function call overhead.
- Inline function may increase efficiency if it is small. Inline functions make a program run faster because the overhead of a function call and return is eliminated.
- To make a function **inline** , the keyword 'inline' is prefixed to the function definition.
- The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function body
}
```

- Usually, the functions are made inline when they are small enough to be defined in one or two lines.

  Example :

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

- All inline functions must be defined before they are called.
- Inline keyword only sends a request, not a command, to the compiler. The compiler may ignore this request if the function is too long or too complicated, and then compile the function as a normal function.
- Some of the situation where inline expansion may not work are:
  - ✓ For functions with **loop**, **switch** or a **goto**.
  - ✓ For functions not returning values (void return type), if a return statement exists.
  - ✓ If functions contain **static** variables.
  - ✓ If **inline** functions are recursive.

Inline function and classes

Member functions defined inside the class definition

When a function is **defined inside a class,** it is treated as inline function i.e. it implicitly becomes an inline function. Therefore, all the restrictions and limitations that apply to an inline function are also applicable to all the functions defined inside class. So, we should define only small functions inside class. Large functions should be defined outside the class.

```
class myclass
{
public:
   int square(int a) // this function is automatically inline
    {

        return(a*a);
    }
};
```

Making an outside(external) member function inline

- We can write the prototype (declaration) of function inside the class and specify it as an inline in the function definition outside the class. Hence an outside member function can be explicitly made inline by using the qualifier **inline** in the header (function declarator) line of the function definition.

```
class myclass
{

  public:
    int square(int a); // declare the function

};

inline int myclass::square(int s) // use inline qualifier
{

        return(a*a);

}
```

Disadvantages of inline functions

- Speed benefits of inline functions diminish as the function grows in size. If function definition is too long, overhead of function call becomes small compared to the execution of the function, and the benefits of line functions may be lost.
  In such cases normal functions must be used. Thus, inline functions restrict the function to be small in size.
- Inline functions make a program run faster but inline function makes the program to take up more memory space (size of binary executable file will be large) because the statements that define the inline functions are reproduced at each point where the function is called. So, there is always time-space trade-off when inline functions are used.

Concept of Reference (Reference Variables)

- C++interfaces a new kind of variable known as the reference variable.
- Reference variable is an internal pointer. (Reference is actually an address)
- A references variable provides an alias (alternative name, nickname) for a previously defined variable.
- For example, if we make the variable **sum** a reference to the variable **total**, then sum and total can be used interchangeably to represent the variable.
- Declaration of reference variable is preceded by & symbol.
- A reference variable is created as follows:

```
datatype & reference-name = variable-name
```

- Here '&' is not read as 'address-of' operator but it means 'reference to'.
- Once a **reference** is initialized with a **variable**, either the **variable** name or the **reference** name may be used to **refer to** the **variable.**
- Example:

```
int total = 100;
int &sum = total;     // sum is reference to total
                     //  internally treated as : int * sum = &total;   by the compiler
```

- `cout<<sum;`
  and
  `cout<<total;`
  both statements print the value 100.
- `total = total + 10;`
  will change the value of both total and sum to 110.
- `sum = 0;`
  will assign 0 to both the variables sum and total.
- Example :
  int n[10];
  int& x = n[10];
  char& a = '\n';

Rules for reference variables

- A reference variable must be initialized at the time of declaration.
- Reference variable can be initialized with already declared variables only.
- References cannot be NULL.
- A reference variable cannot refer to a constant value.

```
    int& a = 10 ;  //compilation error
```

- Data type of reference variable and variable referenced must be same.
- Reference variables cannot be updated (reset). Once a reference is declared and initialized, it cannot be modified later to reference another variable. This means a reference variable cannot refer to more than one variable.

- There shall be no references to references, no arrays of references, and no pointers to references.
- A variable can have more than one references i.e. a variable can be referenced by more than one variable.
- Reference variable has same memory location as the variable it references.
- Internally, reference is treated as address(pointer) by compiler with above rules and restrictions.

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are

✓ We cannot have NULL references.
✓ A pointer can be declared as void but a reference can never be void
✓ Once a reference is initialized to a variable, it cannot be changed to refer to another variable. Pointers can be pointed to another variable at any time.
✓ A reference must be initialized when it is created. Pointers can be initialized at any time.

Function Calling Mechanisms (Parameter/Argument Passing Techniques)

1. **Call by Value (or Pass by Value) – Default**
2. **Call by Address/pointers (or Pass by Address/Pointers)**
3. **Call by Reference (or Pass by reference) – Only in C++**

1. Call by Value (or Pass by Value)

- In this parameter passing method, values of actual parameters in a calling function (caller) are copied to the formal parameters in the called function (callee), and the two types of parameters are stored in different memory locations.
- So, any changes made inside called function (callee) are not reflected in actual parameters (arguments) of calling function (caller).
- Scope of modification is reflected only in called function. The formal parameters are stored in local data area of the callee.

Example of call by value

```
#include <iostream>
using namespace std;
void fun(int x)
{
   x=20;
}
int main()
{
   int a =10;
```

```
    fun(a);

    cout<<"Value of a: "<<a<<endl;
    return 0;
}
```

Output

```
Value of a: 10
```

2. Call by Address/Pointers (or Pass by Address/Pointers)

- In this method addresses of the actual arguments are copied and then assigned to the corresponding formal parameters.
- Formal and actual arguments both points to the same data (because they contain the same address). As a result, any changes made by called function are also reflected in actual arguments in calling function.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly, we need to declare the function parameters as pointer types.
- In call by address, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.
-

Example of call by address (pointer)

```
#include <iostream>
using namespace std;
void fun(int *b)
{
    *b=20;
}
int main()
{
    int a =10;

    fun(&a);    //&a indicates pointer to a

    cout<<"Value of a: "<<a<<endl;

    return 0;
}
```

```
Output
Value of a: 20
```

3. Call by Reference (or Pass by Reference)

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. In pass by reference, reference variables are used as formal parameters.
- Internally, the compiler translates the references to address and copy of address is passed to formal parameters in called function. This means references are treated as addresses internally by compiler.
- When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.
- The formal parameters are of reference type which are accessed in the same way as normal value parameters. However, any modification to the formal parameters in called function is reflected in actual arguments in calling function.
- To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly, we need to declare the function parameters as reference types using &.

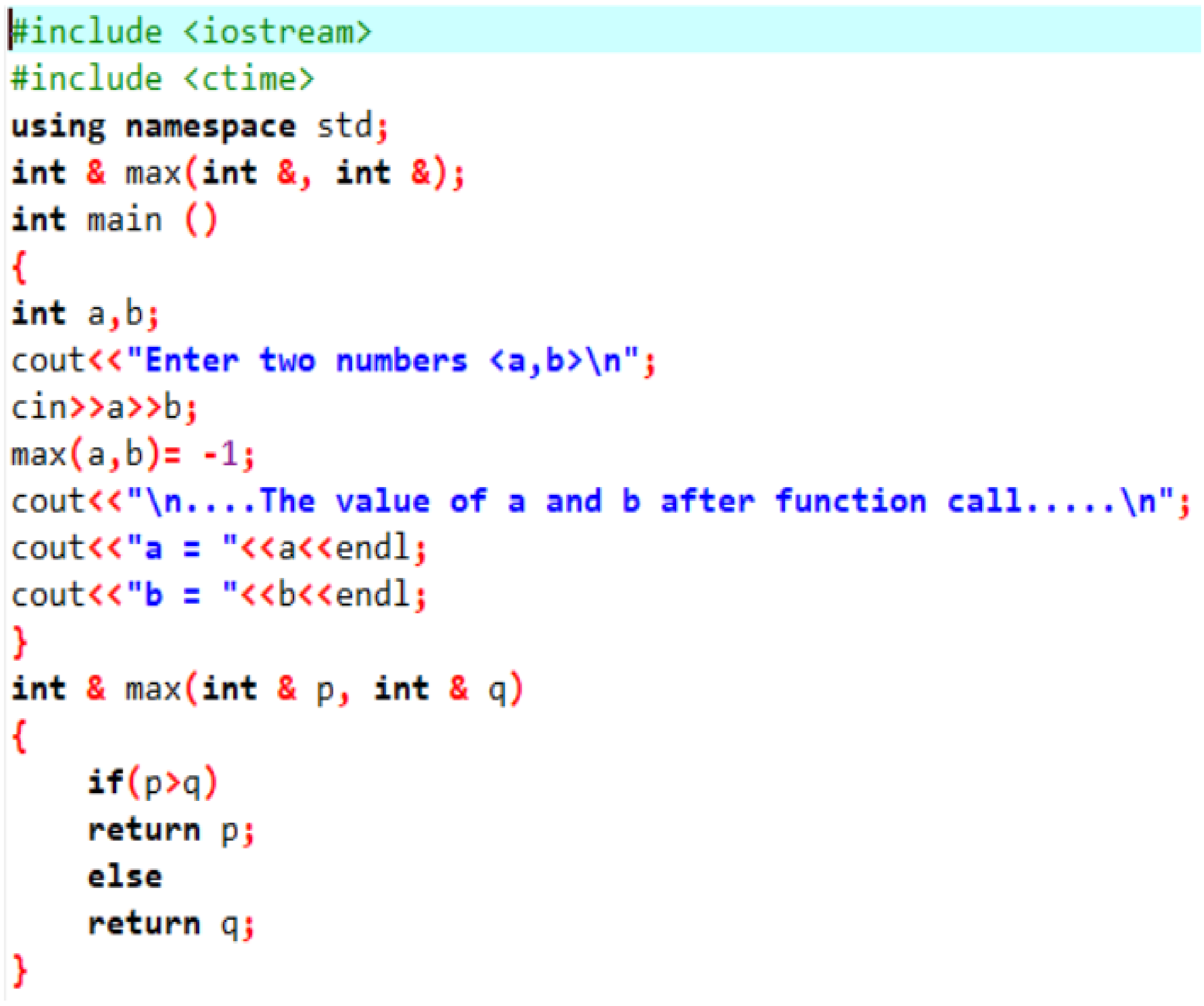
Example of call by reference

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

void fun(int & b)      //internally treated as void fun(int *b)  by compiler
{
     b=50;             //internally treated as *b = 50;
}

int main()
{
     int a =10;

     fun(a);           //internally compiler translates this to fun(&a);

     cout<<"Value of a: "<<a<<endl;

     return 0;
}

Output
Value of a: 50
```

Return by Reference

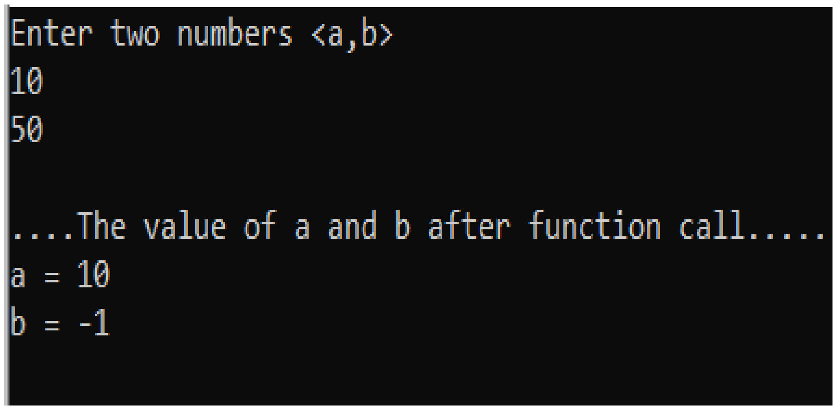
- When a variable is **returned by reference**, a **reference** to the variable is passed back to the caller.
- The caller can then use this **reference** to continue modifying the variable, which can be useful at times.
- When returning a reference, be careful that the object being referred to does not go out of scope. So, it is not legal to return a reference to local variable. But you can always return a reference on a static variable.
- The calling function can be used on the left side of an assignment statement if the definition of functions returns something by reference.

Example of Return by Reference



```
#include <iostream>
#include <ctime>
using namespace std;
int & max(int &, int &);
int main ()
{
int a,b;
cout<<"Enter two numbers <a,b>\n";
cin>>a>>b;
max(a,b)= -1;
cout<<"\n....The value of a and b after function call.....\n";
cout<<"a = "<<a<<endl;
cout<<"b = "<<b<<endl;
}
int & max(int & p, int & q)
{
    if(p>q)
    return p;
    else
    return q;
}
```

Output



```
Enter two numbers <a,b>
10
50

....The value of a and b after function call.....
a = 10
b = -1
```

Dynamic Memory Allocation

- Dynamically allocated memory is allocated on **Heap (free store),** during run-time and non-static and local variables get memory allocated on **Stack,** during compile time.
- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.
- For normal variables and objects like "int a, "Student s"; "char str[10]", etc, memory is automatically allocated by compiler in stack, (when control reaches its declaration) and deallocated ((destroyed), when control exits the block/function in which the variable/object is declared i.e. when it goes goes out of scope)
- For dynamically allocated memory , programmer explicitly allocates memory for variables and objects, and it is programmers' responsibility to explicitly deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes **memory leak** (memory is not deallocated until program terminates). In dynamic memory allocation , the allocation and deallocation of memory takes place at run time.
- Dynamic memory allocation is accomplished using the **new** operator and deallocation is accomplished using the **delete** operator
- When a program requires a variable, it uses new to allocate the variable. When the program no longer needs the variable, it uses delete to deallocate it
- The lifetime of a dynamically allocated variable, therefore, is the time between the execution of the new and delete statements.
- The dynamically allocated objects/variables do not have any predefined scope. They remain in memory until explicitly removed using delete.

Introduction to Memory Management:

DYNAMIC MEMORY ALLOCATION & DEALLOCATION (new & delete)

C uses malloc() and calloc() functions to allocate memory dynamically at run time. It uses the function free() to deallocated dynamically allocated memory.

- C++ supports these functions, it defines two unary operators new and delete that perform the task of allocating and deallocating the memory in a better and easier way.
- A object can be created by using new, and destroyed by using delete.
- A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete.

new operator:-

new operator can be used to create objects of any type .Hence new operator allocates sufficient memory to hold data of objects and it returns address of the allocated memory. Syntax:
Ex: int *p = new int;

Ex: int *p = new int[10];

delete operator:

If the variable or object is no longer required or needed is destroyed by "delete" operator, there by some amount of memory is released for future purpose. Synatx:

Eg: delete p;

If we want to free a dynamically allocated array:
delete [size] pointer-variable;

Program: write a program to find sum of list of integers

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

cout<<"Enter array size:"; cin>>n;
p=new int[n];
cout<<"Enter list of integers"<<endl; for(int i=0;i<n;i++)
cin>>p[i];
//logic for summation int s=0;
for( int i=0;i<n;i++)
s=s+p[i];
cout<<"Sum of array elements is\n"; cout<<s;
delete [ ]p;
return 0;
}
Enter array size:5 Enter list of integers 1 2 3 4 5
Sum of array elements is 15
```

Member Dereferencing operator:-

1. Pointer to a member declarator ::*
2. Pointer to member operator ->*
3. Pointer to member operator .*

1.Pointer to a member declarator ::*

This operator is used for declaring a pointer to the member of the class

```
 #include<iostream.h>
class sample
{
public:
int x;
};
int main()
{       sample s;        //object
int sample ::*p;//pointer decleration s.*p=10; //correct
cout<<s.*p;
}
```

Output:10

2. Pointer to member operator ->*

```
#include<iostream.h> class sample
{
public:
int x;
void display()
{
cout<<"x="<<x<<endl;
}
};

int main()
{
sample s;        //object
sample *ptr;
int sample::*f=&sample::x;
s.x=10;
ptr=&s; cout<<ptr->*f; ptr->display();
}
```

3. Pointer to member operator .*

```
#include<iostream.h> class sample
{

public:

};


int x;

int main()
{
sample s;        //object
int sample ::*p;//pointer decleration s.*p=10; //correct
cout<<s.*p;
}
```

Pointers to Objects:
Pointers to objects are useful for creating objects at run time. To access members arrow operator (   ) and de referencing operator or indirection (*) are used.
Declaration of pointer.
className*ptr
ex:
item *obj;
Here obj is a pointer to object of type item.

```
class item
{


public:

int code; float price;

void getdata(int a,float b)
{

code=a; price=b;
}
void show()
{
cout<<"code:"<<code<<"\n"<<"Price:"<<price<<endl;
}
};
```

Declaration of object and pointer to class item:

```
item obj;
item *ptr=&obj;
```

The member can be accessed as follows

a) Accessing members using dot operator obj.getdata(101,77.7); obj.show();

b) using pointer
ptr->getdata(101,77.7); ptr->show();

c) Using de referencing operator and dot operator
 (*ptr).getdata(101,77.7);
(*ptr).show();

Creating array of objects using pointer:
item *ptr=new item[10];
Above declaration creates memory space for an array of 10 objects of type item.

```
#include<iostream.h>
class item
{


public:

int code; float price;

void getdata(int a,float b)
{

code=a; price=b;
}
void show()
{
cout<<code<<"\t"<<price<<endl;
}
};
int main()
{
int n; int cd;
float pri;
cout<<"Enter number of objects to be created:"; cin>>n;
item *ptr=new item[n]; item *p;
p=ptr;
for(int i=0;i<n;i++)
{
cout<<"Enter data for object"<<i+1;
cout<<"\nEnter Code:";cin>>cd;
cout<<"Enter price:";
cin>>pri;
p->getdata(cd,pri); p++;
}

p=ptr;
cout<<"Data in various objects are "<<endl;
 cout<<"Sno\tCode\tPrice\n";
for(i=0;i<n;i++)
{
cout<<i+1<<"\t";
ptr->show();
ptr++;
}
 return 0;
}

cout<<i+1<<"\t";
ptr->show(); ptr++;
```

Pointers to Derived Classes:
Pointers can be declared to derived class. it can be used to access members of base class and derived class. A base class pointer can also be used to point to object of derived class but it can access only members that are inherited from base class.

```
#include<iostream.h>
 class base
{
public:
int a;
void get_a(int x)
{
a=x;
}
void display_a()
{
cout<<"In base"<<"\n"<<"a="<<a<<endl;
}
};
class derived:public base
{
int b; public:
void get_ab(int x,int y)
{
a=x; b=y;
}
void display_ab()
{
cout<<"In Derived "<<"\n"<<"a="<<a<<"\nb="<<b<<endl;
}
};
int main()
{
base b; base *bptr;
bptr=&b;//points to the object of base class bptr->get_a(100);
bptr->display_a();
derived d; derived *dptr;
dptr=&d;//points to the object of derived class
dptr->get_a(400);

dptr->display_a();
dptr->get_ab(300,200); dptr->display_ab();
bptr=&d;//points to the object of derived class bptr->get_a(400);
bptr->display_a();
return 0;
}
```

Output:

```
In base a=100
In base a=400
In Derived
a=300 b=200
In base
a=400
```

Introduction to Constructors:

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created.

Definition:- A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same name as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs
the values of data members of the class. A constructor is declared and defined as follows:

integer obj1; => not only creates object obj1 but also initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function.

CHARACTERISTICS OF CONSTRUCTOR

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

- They do not have return type, not even void.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- They make implicit calls to the operators new and delete when memory allocation is required.

Constructors are of 3 types:
1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1. Default Constructor:
A constructor that accepts no parameters is called the default constructor.

```
 #include<iostream.h>
#include<conio.h> class item
{
int m,n; public: item()
{
m=10; n=20;
}
void put();
};
void item::put()
{
cout<<m<<n;
}
void main()
{
item t; t.put();
getch();
}
```

2. Parameterized Constructors:-
The constructors that take parameters are called parameterized constructors.

```
 #include<iostream.h>
class item
{
int m, n;
public:
item(int x, int y)
{
m=x; n=y;
}
};
```

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

item t; may not work. We must pass the initial values as arguments to the constructor function when an object is declared.
This can be done in 2 ways:

```
 item t=item(10,20); //explicit call

item t(10,20); //implicit call
```

Eg:

```
#include<iostream.h>
 #include<conio.h>
class item
{
int m,n; public:
item(int x,int y)
{
m=x; n=y;
}
void put();
};
void item::put()
{
cout<<m<<n;
}
void main()
{
item t1(10,20);
item t2=item(20,30); t1.put();
t2.put();
getch();
}
```

3.Copy Constructor: A copy constructor is used to declare and initialize an object from another object.
Eg:

```
item t2(t1); or
item t2=t1;
```

1. The process of initializing through a copy constructor is known as copy initialization.
2. t2=t1 will not invoke copy constructor. t1 and t2 are objects, assigns the values of t1 to t2.
3. A copy constructor takes a reference to an object of the same class as itself as an argument.

```
#include<iostream.h>
class sample
{
int n; public:
sample()
{ n=0;
}
sample(int a)
{

n=a;
}
sample(sample &x)
{
n=x.n;
}
void display()
{
cout<<n;
}
};
void main()
{
sample A(100); sample B(A); sample C=A; sample D;
D=A;
A.      display();
B.      display();
C.      display();
D.      display();
}
```

Output: 100 100   100 100

Multiple Constructors in a Class:
Multiple constructors can be declared in a class. There can be any number of constructors in a class.

```
class complex
{
float real,img; public:
complex()//default constructor
{
real=img=0;
}
complex(float r)//single parameter parameterized constructor
{
real=img=r;
}

complex(float r,float i) //two parameter parameterized constructor
{
real=r;img=i;
}
complex(complex & c)//copy constructor
{
real=c.real; img=c.img;
}
complex sum(complex c )
{

complex t; t.real=real+c.real; t.img=img+c.img; return t;
}
voidshow()
{
If(img>0)
cout<<real<<"+i"<<img<<endl;
else
{
img=-img;
cout<<real<<"-i"<<img<<endl;
}
}
void main()
{
complex c1(1,2);
complex c2(2,2);
compex c3;
c3=c1.sum(c3);
 c3.show();

}
```

DESTRUCTORS:

A destructor, is used to destroy the objects that have been created by a constructor.
Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.
Eg: ~item() { }

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

```
#include<iostream>
using namespace std;
 class Marks
{
public:
int maths; int science;

//constructor Marks()
{
cout << "Inside Constructor"<<endl;
cout << "C++ Object created"<<endl;
}

//Destructor

~Marks()
{
cout << "Inside Destructor"<<endl;
cout << "C++ Object destructed"<<endl;
}
};

int main( )
{
Marks m1; Marks m2; return 0;
}
```

Output:
```
Inside Constructor
C++ Object created Inside Constructor C++ Object created Inside Destructor
C++ Object destructed Inside Destructor
C++ Object destructed
```

FRIEND FUNCTIONS:

The private members cannot be accessed from outside the class. i.e.a non-member function cannot have an access to the private data of a class. In C++ a non-member function can access private by making the function friendly to a class.

Definition:

A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining . It can access private members of a class. It is declared by using keyword “friend”

Ex:
```
class sample
{
int x,y; public:
sample(int a,int b); friend int sum(sample s);
};
sample::sample(int a,int b)
{
x=a;
y=b;
}
int sum(samples s)
{
int sum;
sum=s.x+s.y; return 0;
}
```

```
void main()
{
Sample obj(2,3); int res=sum(obj);
cout<< "sum="<<res<<endl;
}
```

A friend function possesses certain special characteristics:

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

```
#include<iostream.h>
class sample
{
int a; int b; public:
void setvalue()
{ a=25; b=40;
}
friend float mean(sample s);
};

float mean(sample s)
{
return float(s.a+s.b)/2.0;
}
int main()
{
sample X;
X.setvalue();
cout<<"Mean value="<<mean(X); return 0;
}
```

Program to find max of two numbers using friend function for two different classes

```
#include<iostream>
using namespace std;
class sample2; class sample1
{
int x; public:
sample1(int a);
friend void max(sample1 s1,sample2 s2)
};
sample1::sample1(int a)
```

```
{
x=a;
}
class sample2
{
int y; public:
sample2(int b);
friend void max(sample1 s1,sample2 s2);
};
Sample2::sample2(int b)
{
y=b;
}
void max(sample1 s1,sample2 s2)
{
If(s1.x>s2.y)
cout<<"Data member in Object of class sample1 is larger "<<endl;

else

}


cout<<"Data member in Object of class sample2 is larger "<<endl;

void main()
{
sample1 obj1(3); sample2 obj2(5); max(obj1,obj2);
}
```

Program to add complex numbers using friend function

```
 #include<iostream>
using namespace std;
class complex
{
float real,img;
public:
complex();
complex(float x,float y)
friend complex add_complex(complex c1,complex c2);
};
complex::complex()
{
real=img=0;
}
complex::complex(float x,float y)
```

```
{
real=x;img=y;
}
complex add_complex(complex c1,complex c2)
{
complex t;
t.real=c1.real+c2.real; t.img=c1.img+c2.img; return t;
}
void complex::display ()
{
cout<<real<<"+i"<<img<<endl;

}


cout<<real<<"+i"<<img<<endl;

int main()
{
complex obj1(2,3); complex obj2(-4,-6);
complex obj3=add_compex(obj1,obj2); obj3.display();
return 0;
}
```

Friend Class:

A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

```
#include <iostream.h>
class sample_1
{


public:

friend class sample_2;//declaring friend class int a,b;

void getdata_1()
{

cout<<"Enter A & B values in class sample_1"; cin>>a>>b;
}

void display_1()
{
cout<<"A="<<a<<endl; cout<<"B="<<b<<endl;
}
};
class sample_2
{


public:

int c,d,sum; sample_1 obj1;

void getdata_2()
{

obj1.getdata_1();
cout<<"Enter C & D values in class sample_2"; cin>>c>>d;
}
void sum_2()
{
sum=obj1.a+obj1.b+c+d;
}

void display_2()
{
cout<<"A="<<obj1.a<<endl;          cout<<"B="<<obj1.b<<endl;          cout<<"C="<<c<<endl;
cout<<"D="<<d<<endl; cout<<"SUM="<<sum<<endl;
}
};
int main()
{
sample_1 s1;
s1.getdata_1(); s1.display_1();
sample_2 s2;
s2.getdata_2(); s2.sum_2(); s2.display_2();
}
```

```
Enter A & B values in class sample_1:1 2 A=1
B=2
Enter A & B values in class sample_1:1 2 3 4
 Enter C & D values in class sample_2: A=1 B=2 C=3 D=4
SUM=10
```

‘this’ pointer in C++

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

The compiler supplies an implicit pointer along with the names of the functions as ‘this’. The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.

‘this’ pointer is not available in static member functions as static member functions can be called without any object(with class name).
For a class X, the type of this pointer is ‘X* ‘. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’
In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.
C++ lets object destroy themselves by calling the following code :

delete this;

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

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

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};
```

```
int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

Output:

```
 x = 20
```

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

UNIT - III

INHERITANCE:

The mechanism of deriving a new class from an old one is called inheritance or derivation. The old class is referred to as the base class and the new one is called the derived class or sub class. The derived class inherits some or all of the traits from the base class.
A class can also inherit properties from more than one class or from more than one level.Reusability is an important feature of OOP

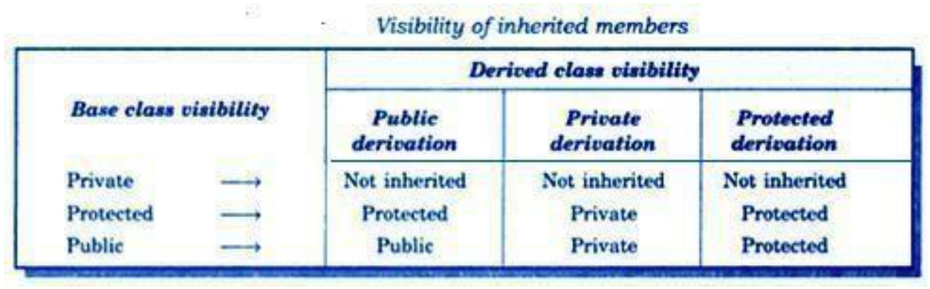
A derived class can be defined by specifying its relationship with the base class in addition to its own details.

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

```
class ABC : private XYZ
{
members of ABC;
};

//private derivation

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

// protected derivationmembers of ABC;
};
class ABC:XYZ          //private by default
{
members of ABC;
};
```

When a base class is **privately** inherited by a derived class, public members of the base class can only be accessed by the member functions of the derived class. private members of base class are inaccessible to the objects of the derived class
When a base class is **protected** inherited by a derived class, public members of the base class can only be accessed by the member functions of the derived class. private members of base class are inaccessible to the objects of the derived class. If private members of base class are to be inherited to derived class then declare them as protected
When the base class is **publicly** inherited, public members of the base class is publicly inherited, public members of the base class become public members of the derived class and therefore they are accessible to

the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class
In inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capability of the existing classes, becomes a very powerful tool for incremental program development


Visibility of inherited members

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private →	Not inherited	Not inherited	Not inherited
Protected →	Protected	Private	Protected
Public →	Public	Private	Protected

Types of Inheritance:
1.Single Inheritance
2.Multi level Inheritance
3.Mutiple Inheritance
4.Hybrid inheritance
5. Hierarchical Inheritance.

1.SINGLE INHERITANCE: one derived class inherits from only one base class. It is the most simplest form of Inheritance.

/ /Base class

//Derived class

```
#include<iostream>
using namespace std;
class A
{
public:
int a,b; void get()
{
cout<<"Enter any two Integer values"<<endl; cin>>a>>b;
}
};
class B:public A
{
int c; public:
void add()
{
```

```
c=a+b; cout<<a<<"+"<<b<<"="<<c;
}
};
int main()
{
B b; b.get();
b.add();
}
Output:
Enter any two Integer values 1 2
1+2=3
```

2. **MULTILEVEL INHERITANCE:** In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

```
#include<iostream.h>
 class A
{
public:
int a,b; void get()
{
cout<<"Enter any two Integer values"<<endl; cin>>a>>b;
}
};

class B:public A
{
public:
int c;
void add()
{
c=a+b;
}
};

class C:public B
{
public:
void show()
{ cout<<a<<"+"<<b<<"="<<c;
}
};
int main()
{
C c;
c.get();
c.add();
c.show();
}
```

Output:

```
Enter any two Integer values 12 14
12+14=26
```

3. Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.

Syntax:

```
class D : visibility A, visibility B,….
{
………………
}
```

```
#include<iostream.h>
 class A
{
public: int a;


};
class B
{
public:


};

void getA()
{
cout<<"Enter an Integer value"<<endl; cin>>a;
}



int b;
void getB()
{
cout<<"Enter an Integer value"<<endl; cin>>b;
}


class C:public A,public B
{
public: int c;
void add()
{
c=a+b; cout<<a<<"+"<<b<<"="<<c<<endl;
}
};
int main()
{
C obj; obj.getA();
obj.getB();
obj.add();
}
```

```
Enter an Integer value 12 Enter an

Integer value 13
12+13=25
```

4. Hybrid Inheritance: Hybrid inheritance is combination of two or more inheritances such as single,multiple,multilevel or Hierarchical inheritances.

```
#include<iostream.h>
class arithmetic
{
protected:
int num1, num2;

public:

};


void getdata()
{
cout<<"For Addition:"; cout<<"\nEnter the first number: "; cin>>num1;
cout<<"\nEnter the second number: "; cin>>num2;
}

class plus:public arithmetic
{
protected:
int sum;

public:
};


void add()
{
sum=num1+num2;
}

class minus
{
protected:
int n1,n2,diff;

public:


void sub()
{
cout<<"\nFor Subtraction:"; cout<<"\nEnter the first number: "; cin>>n1;
cout<<"\nEnter the second number: ";

cin>>n2; diff=n1-n2;
}
};
class result: public plus, public minus
{
public:
void display()
{
cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum; cout<<"\nDifference of "<<n1<<" and
"<<n2<<"= "<<diff;
}
};
int main()
{
result z;
z.getdata();
z.add();
z.sub();
z.display();
}
```

```
For Addition:
Enter the first number: 1 Enter the second number: 2
For Subtraction:
Enter the first number: 3 Enter the second number: 4
Sum of 1 and 2= 3
Difference of 3 and 4= -1
```

5. Hierarchical Inheritance:- Inheriting is a method of inheritance where one or more derived classes is derived from common base class.

```
#include<iostream.h>
class A //Base Class
{
public:
int a,b;
void getnumber()
{

cout<<"\n\nEnter Number :\t"; cin>>a;
}
};
class B : public A //Derived Class 1
{
public:
void square()
{
getnumber(); //Call Base class property cout<<"\n\n\tSquare of the number :\t"<<(a*a);
}
};
class C :public A //Derived Class 2
{
public:
void cube()
{
getnumber(); //Call Base class property cout<<"\n\n\tCube of the number :::\t"<<(a*a*a);
}
};
int main()
{
B b1;           //b1 is object of Derived class1
b1.square(); //call member function of class B C c1;  //c1 is object of Derived class 2 c1.cube(); //call
member function of class C
}
```

```
Enter Number : 2
Square of the number : 4 Enter Number : 3
Cube of the number ::: 27
```

UNIT - IV

STATIC CLASS MEMBERS

Static Data Members Static Member Functions
Static Data Members:

A data member of a class can be qualified as static. A static member variable has certain special characteristics:
 It is initialized to zero when the first object of its class is created. No other initialization is permitted.
 Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created. It is visible only within the class, but its lifetime is the entire program.
 Static data member is defined by keyword static‟
Syntax:
Data type class name::static_variable Name;
Ex: int item::count;

```
#include<iostream.h>
#include<conio.h>
class item
{

public:

static int count; int number;

void getdata(int a)
{

number=a; count++;
}
void getcount()
{
cout<<"count is"<<count;
}
};
int item::count;//decleration int main()
{
item a,b,c;
a.getcount();
b.getcount();
c.getcount(); a.getdata(100); b.getdata(200); c.getdata(300); cout<<"After reading data"; a.getcount();
b.getcount();
c.getcount(); return 0;
}
```

```
Output: count is 0
count is 0
count is 0
After reading data count is 3
count is 3
count is 3
```

Static Member Functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:
A static function can have access to only other static members (functions or variables) declared in the same class.
A static member function is to be called using the class name (instead of its objects) as follows: class-name :: function-name;

```
#include<iostream.h> class test
{


public:

int code;
static int count;

void setcode()
{

code=++count;
}
void showcode()
{
cout<<"object number"<<code;
}
static void showcount()
{
cout<<"count"<<count;
}
};
int test::count; int main()
{
test t1,t2;
t1.setcode();
t2.setcode(); test::showcount();

test t3;

}


t3.setcode(); test::showcount(); t1.showcode(); t2.showcode(); t3.showcode(); return 0;
```

Output:

```
count 2
count 3
object number 1
object number 2
object number 3
```

Const member functions in C++

Like member functions and member function arguments, the objects of a class can also be declared as **const**. an object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.
A const object can be created by prefixing the const keyword to the object declaration. Any attempt to change the data member of const objects results in a compile-time error.

Syntax:

const Class_Name Object_name;

- When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.
- Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

A function becomes const when the const keyword is used in the function's declaration. The idea of const functions is not to allow them to modify the object on which they are called. It is recommended the practice to make as many functions const as possible so that accidental changes to objects are avoided.

Following is a simple example of a const function.

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}

    // We get compiler error if we add a line like "value = 100;"
    // in this function.
    int getValue() const {return value;}
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Output:

```
20
```

When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

For example the following program has compiler errors.

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
    int getValue() {return value;}
};

int main() {
    const Test t;
    cout << t.getValue();
    return 0;
}
```

Output:

```
 passing 'const Test' as 'this' argument of 'int
Test::getValue()' discards qualifiers
```

Let's look at another example:

```
// Demonstration of constant object,
// show that constant object can only
// call const member function
#include<iostream>
using namespace std;
class Demo
{
    int value;
    public:
    Demo(int v = 0) {value = v;}
    void showMessage()
    {
        cout<<"Hello World We are Tushar, "
        "Ramswarup, Nilesh and Subhash Inside"
        " showMessage() Function"<<endl;
    }
    void display()const
    {
        cout<<"Hello world I'm Rancho "
        "Baba Inside display() Function"<<endl;
    }
};
int main()
{
    //Constant object are initialised at the time of declaration using constructor
     const Demo d1;
     //d1.showMessage();Error occurred if uncomment.
     d1.display();
     return(0);
}
OUTPUT : Hello world I'm Rancho Baba Inside display() Function
```

const Data Members

Data members of a class may be declared as const. Such a data member **must** be initialized by the constructor using an initialization list. Once initialized, a const data member may never be modified, not even in the constructor or destructor.

Data members that are both static and const have their own rules for initialization.

RUNTIME POLYMORPHISM USING VIRTUAL FUNCTIONS

Static & Dynamic Binding

Polymorphism means one name -multiple forms.
The overloaded member functions are „selected" for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and compiler is able to select the appropriate function for a particular call at the compile time itself. This is called Early Binding or Static Binding or Static Linking. Also known as compile time polymorphism. Early binding means that an object is bound to its function call at the compile time.
It would be nice if the appropriate member function could be selected while the program is running. This is known as runtime polymorphism. C++ supports a mechanism known as virtual function to achieve run time polymorphism.
At the runtime, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time.

VIRTUAL FUNCTIONS

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.

Function Overriding

When we use the same function name in both the base and derived classes, the function in the bas class is declared as virtual using the keyword virtual preceding its normal declaration.

When a function is made virtual, C++ determines which function to use at runtime based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

```
#include<iostream.h> class Base
{
public:
void display()
{
cout<<"Display Base";
}
virtual void show()
{
cout<<"Show Base";
}
};
class Derived : public Base
{
public:
void display()
{
cout<<"Display Derived";
}
void show()
{
cout<<"show derived";
}
};
void main()
{
Base b;
Derived d;
Base *ptr;
cout<<"ptr points to Base";
ptr=&b;
ptr->display(); //calls Base
ptr->show(); //calls Base
cout<<"ptr points to derived";
ptr=&d;
ptr->display(); //calls Base
ptr->show(); //class Derived
}
```

Output:

```
ptr points to Base

Display Base Show Base
ptr points to Derived Display Base
```

Show Derived

When ptr is made to point to the object d, the statement ptr->display(); calls only the function associated with the Base i.e.. Base::display()
where as the statement ptr->show(); calls the derived version of show(). This is because the function display() has not been made virtual in the Base class.

Rules For Virtual Functions:

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

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

Pure Virtual Functions and Abstract Classes in C++

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
        // Data members of class
public:
        // Pure Virtual Function
        virtual void show() = 0;
/* Other members */
};
```

A complete example:

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

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

class Base
{
int x;
public:
        virtual void fun() = 0;
        int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
        int y;
public:
        void fun() { cout << "fun() called"; }
};

int main(void)
{
        Derived d;
        d.fun();
        return 0;
}
```

Output:

```
fun() called
```

Important Points:

1) A class is abstract if it has at least one pure virtual function.
In the following example, Test is an abstract class because it has a pure virtual function show().

```
// pure virtual functions make a class abstract
#include<iostream>
```

```
using namespace std;

class Test
{
   int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}
```

Output:

```
Compiler Error: cannot declare variable 't' to be of abstract
 type 'Test' because the following virtual functions are pure
within 'Test': note:     virtual void Test::show()
```

2) We can have pointers and references of abstract class type.
For example the following program works fine.

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

class Base
{
public:
    virtual void show() = 0;
};

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

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Output:

```
In Derived
```

3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
The following example demonstrates the same.

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
  Derived d;
  return 0;
}
```

```
Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived'  because the following virtual functions are pure within
'Derived': virtual void Base::show()
```

OVERLOADING

OPERATOR OVERLOADING

C++ has the ability to provide the operators with as special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading. We can overload all the operators except the following:
Class member access operator (“.” And ".*”) Scope resolution operator “::”
Size operator (sizeof) Conditional operator
To define an additional task to an operator, specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function.
The process of overloading involves following steps:

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

Ex:
complex complex::operator+(complex c)
{
complex t;
t.real=real+c.real; t.img=img+c.img; return t;
}

Concept of Operator Overloading

One of the unique features of C++ is Operator Overloading. Applying overloading to operators means, same operator in responding different manner. For example operator + can be used as concatenate operator as well as additional operator.

That is 2+3 means 5 (addition), where as "2"+"3" means 23 (concatenation).
Performing many actions with a single operator is operator overloading. We can assign a user defined function to an operator. We can change function of an operator, but it is not recommedned to change the actual functions of operator. We can't create new operators using this operator overloading.

Operator overloading concept can be applied in following two major areas (Benefits)

1. Extension of usage of operators
2. Data conversions

Rules to be followed for operator overloading:-

1. Only existing operators can be overloaded.
2. Overloaded operators must have at least one operand that is of user defined operators 3.We cannot change basic meaning of an operator.
4. Overloaded operator must follow minimum characteristics that of original operator
5. When using binary operator overloading through member function, the left hand operand must be an object of relevant class

The number of arguments in the overloaded operator" s arguments list depends on

1. Operator function must be either member function or friend function.

2. If operator function is a friend function then it will have one argument for unary operator & two arguments for binary operator
3. If operator function is a member function then it will have Zero argument for unary operator & one arguments for binary operator

Unary Operator Overloading

An unary operator means, an operator which works on single operand. For example, ++ is an unary operator, it takess single operand (c++). So, when overloading an unary operator, it takes no argument (because object itself is considered as argument).

Syntax for Unary Operator (Inside a class)

```
return-type operator operatorsymbol( )
{
//body of the function
}
```
Ex:
```
void operator-()
{
real=-real; img=-img;
}
```
Syntax for Unary Operator (Outside a class)
```
return-type classname::operator operatorsymbol( )
{
//body of the function
}
```
Example 1:-
```
void operator++()
{
counter++;
```

}
Example 2:-

```
void complex::operator-()
{
real=-real; img=-img;
}
```

The following simple program explains the concept of unary overloading.

```
#include < iostream.h > #include < conio.h >
// Program Operator Overloading class fact
{
int a;
public:
fact ()

{
a=0;
}
fact (int i)
{
a=i;
}
fact operator!()
{
int f=1,i; fact t;
for (i=1;i<=a;i++)
{
f=f*i;
}
t.a=f; return t;
}
void display()
{
cout<<"The factorial "<< a;
}
};
void main()
{

int x;

}


cout<<"enter a number"; cin>>x;
fact s(x),p; p=!s; p.display();
```

Output for the above program:

```
Enter a number 5
The factorial of a given number 120
```

Explanation:

We have taken „!" as operator to overload. Here class name is fact. Constructor without parameters to take initially value of „x" as 0. Constructor with parameter to take the value of „x" . We have create two objects one for doing the factorial and the other for return the factorial. Here number of parameter for an overloaded function is 0. Factorial is unary operator because it operates on one dataitem. operator overloading find the factorial of the object. The display function for printing the result.

Overloading Unary Operator -

Example 1:-

Write a program to overload unary operator –

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

float real,img; public:
complex();
complex(float x, float y); void display();
void operator-();
};
complex::complex()
{
real=0;img=0;
}
complex::complex(float x, float y)
{
real=x; img=y;
}
void complex::display()
{
int imag=img;
if(img<0)
{
imag=-img;
cout<<real<<" -i"<<imag<<endl;
}
else
cout<<real<<" +i"<<img<<endl;
}
void complex::operator-()
{
real=-real; img=-img;
}
int main()
```

```
{
complex c(1,-2);
c.display();
cout<<"After Unary - operation\n";
-c; c.display();
}
```

Example 2:- #include<iostream.h> using namespace std; class space

```
{
int x,y,z; public:
void getdata(int a,int b,int c); void display();

void operator-();
};
void space :: getdata(int a,int b,int c)
{
x=a; y=b; z=c;
}
void space :: display()
{
cout<<"x="<<x<<endl; cout<<"y="<<y<<endl; cout<<"z="<<z<<endl;
}
void space :: operator-()
{
x=-x;
y=-y;
z=-z;
}
int main()
{
space s; s.getdata(10,-20,30); s.display();
-s;
cout<<"after negation\n"; s.display();
}
```

```
Output: x=10 y=-20 z=30
after negation x=-10
y=20 z=-30
```

Example 3:-
Unary minus operator using a friend function

```
#include<iostream.h> #include<iostream.h> using namespace std; class space
{
int x,y,z; public:

void getdata(int a,int b,int c); void display();
friend void operator-(space &);
};
```

```
void space :: getdata(int a,int b,int c)
{
x=a; y=b; z=c;
}
void space :: display()
{
cout<<x<<" "<<y<<" "<<z<<endl;
}
void operator-(space &s)
{
s.x=-s.x;
s.y      =-s.y;
s.z      =-s.z;
}
int main()
{
space S; S.getdata(10,-20,30); S.display();
-S;
cout<<"after negation\n"; S.display();
}
```

Output:

```
10      -20 30
after negation
-10 20-30
```

Binary Operator Overloading

An binary operator means, an operator which works on two operands. For example, + is an binary operator, it takes single operand (c+d). So, when overloading an binary operator, it takes one argument (one is object itself and other one is passed argument).

Example

```
complex operator+(complex s)
{
complex t; t.real=real+s.real; t.img=img+s.img; return t;
}
```

The following program explains binary operator overloading: #include < iostream.h >

```
#include < conio.h > class sum
{
int a; public:
sum()
{
a=0;
}

sum(int i)
```

```
{

}


a=i;

sum operator+(sum p1)
{
sum t;


}
void main ()
{

t.a     =a+p1.a; return t;

cout<<"Enter Two Numbers:" int a,b;
cin>>a>>b; sum x(a),y(b),z; z.display(); z=x+y;
cout<<"after applying operator \n"; z.display();
getch();
}
```

Output for the above program:

```
Enter two numbers 5 6 After applying operator

The sum of two numbers 11
```

Explanation: The class name is „sum" . We have create three objects two for to do the sum and the other for returning the sum. " +" is a binary operator operates on members of two objects and returns the result which is member of a object.here number of parameters are 1. The sum is displayed in display function.

Write a program to over load arithmetic operators on complex numbers using member function

```
#include<iostream.h>
class complex
{


public:

float real,img;

complex(){ }
complex(float x, float y)
{

real=x; img=y;
}
complex operator+(complex c);
 void display();
};
complex complex::operator+(complex c)
{
complex temp;
temp.real=real+c.real;
 temp.img=img+c.img;
return temp;
}
void complex::display()
{
cout<<real<<"+i"<<img;
}

int main()
{
complex c1,c2,c3;
c1=complex(2.5,3.5);
 c2=complex(1.6,2.7);
c3=c1+c2;
c3.display();
return 0;
}
```

Overloading Binary Operators Using Friends

1. Replace the member function declaration by the friend function declaration in the class friend complex operator+(complex, complex)
2. Redefine the operator function as follows:

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

Write a program to over load arithmetic operators on complex numbers using friend function

```
#include<iostream.h>
class complex
{


public:

float real,img;

complex(){ } complex(float x, float y)
{
```

```
real=x; img=y;
}
friend complex operator+(complex); void display();
};
complex operator+(complex c1, complex c2)
{
complex temp;
temp.real=c1.real+c2.real; temp.img=c1.img+c2.img; return temp;
}
void complex::display()
{
If(img<0)
{


}
else

}

img=-img; cout<<real<<"-i"<<img;

cout<<real<<"+i"<<img;

int main()
{
complex c1,c2,c3;
c1=complex(2.5,3.5); c2=complex(1.6,2.7); c3=c1+c2; c3.display();
return 0;
}
```

UNIT - V

Template(Generic Programming)

Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
A significant benefit of object oriented programming is reusability of code which eliminates redundant coding. An important feature of C++ called templates strengthens this benefit of OOP and provides great flexibility to the language. Templates support generic programming, which allows to develop reusable software components such as functions, classes etc.. supporting different data types in a single framework.
Templates Concept Introduction
Instead of writing different functions for the different data types, we can define common function. For example

```
int max(int a,int b); // Returns maximum of two integers
 float max(float a,float b); // Return maximum of two floats
char max(char a,char b); // Returns maximum of two characters (this is called as function overloading)
```

But, instead of writing three different functions as above, C++ provided the facility called "Templates". With the help of templates, we can define only one common function as follows:

```
T max(T a,T b); // T is called generic data type
```

Template functions are the way of making function/class abstracts by creating the behavior of function without knowing what data will be handled by a function. In a sense this is what is known as "generic functions or programming".
Template function is more focused on the algorithmic thought rather than a specific means of single data type. For example you could make a templated stack push function. This push function can handle the insertion operation to a stack on any data type rather then having to create a stack push function for each different type.
Syntax:

```
template < class type >
ret_type fun_name(parameter list)
{
--------------//body of function

} //www.suhritsolutions.com
```

Features of templates:-

1. It eliminates redundant code
2. It enhances the reusability of the code.
3. It provides great flexibility to language

Templates are classified into two types. They are

1. Function templates
2 Class Templates.

Function Templates

The templates declared for functions are called as function templates. A function template defines how an individual function can be constructed.

Syntax :

```
template < class type,………> ret _type fun_ name(arguments)
{
-----------------// body of the function

}
```

CLASS TEMPLATES

The templates declared for classes are called class templates. A class template specifies how individual classes can be constructed similar to the normal class specification. These classes model a generic class which support similar operations for different data types. General Form of a Class Template

```
template <class T> class class-name
{
…….
…….
};
```

A class created from a class template is called a template class. The syntax for defining an object of a template class is:

```
classname<type>
objectname(arglist);
```

```
#include<iostream.h>
 #include<conio.h>
 template <class T>
class swap
{
T a,b;
public: swap(T x,T y)
{
a=x; b=y;
}
void swapab()
{
T temp;

temp=a; a=b; b=temp;
}
void showdata()
{
cout<<a<<b;
}
};
void main()
```

```
{
int m,n; float m1,n1;
cout<<"Enter integer values"; cin>>m>>n;
cout<<"Enter floating values"; cin>>m1>>n1;
swap<int> c1(m,n); swap<float> c2(m1,n1); c1.swapab(); c1.showdata(); c2.swapab(); c2.showdata();
}
```

Class Template with Multiple Parameters

Syntax:

```
template <class T1, class T2,….> class class-name
{
…….
…….
}
```

```
#include<iostream.h>
template <class T1,class T2>
class Test
{
T1 a; T2 b;
public:
Test(T1 x,T2 y)
{
a=x; b=y;
}
void show()

{
cout<<a<<b;
}
};
void main()
{
Test<float,int> test1(1.23,123);
Test<int,char> test2(100,"w");
test1.show();
test2.show();
}
```

FUNCTION TEMPLATES

Like class template we can also define function templates that would be used to create a family of functions with different argument types.

General Form:

```
template <class T>
return-type function-name (arguments of type T)
{
………
………
}
#include<iostream.h>
 template<class T>
void swap(T &x, T &y)
{
T temp = x; x=y; y=temp;
}
void fun(int m,int n,float a,float b)
{
cout<<m<<n; swap(m,n); cout<<m<<n; cout<<a<<b; swap(a,b); cout<<a<<b;
}
int main()
{ fun(100,200,11.22,33.44);
return 0;
}
```

Function Template with Multiple Parameters

Like template class, we can use more than one generic data type in the template statement, using a comma-separated list as shown below:

template <class T1, class T2,.> return-type function- name(arguments of types T1,T2.) {

```
……..
……..
}
#include<iostream.h>
 #inlcude<string.h>
template<clas T1, class T2>
void display(T1 x,T2 y)
{
cout<<x<<y;
}
int main()
{

display(1999,"EBG"); display(12.34,1234); return 0;
}
```

Exception handling

Exceptions: Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing .Anomalies might include conditions such ass division by zero, accessing an array outside of its bounds or running out of memory or disk space. When a program encounters an exception condition, it must be identified and handled.
Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.
Types of exceptions:There are two kinds of exceptions 1.Synchronous exceptions
2. Asynchronous exceptions

1. Synchronous exceptions: Errors such as "Out-of-range index" and "over flow" are synchronous exceptions
2. Asynchronous exceptions: The errors that are generated by any event beyond the control of the program are called asynchronous exceptions

The purpose of exception handling is to provide a means to detect and report an exceptional circumstance

Exception Handling Mechanism:

An exception is said to be thrown at the place where some error or abnormal condition is detected. The throwing will cause the normal program flow to be aborted, in a raised exception. An exception is thrown programmatic, the programmer specifies the conditions of a throw.
In handled exceptions, execution of the program will resume at a designated block of code, called a catch block, which encloses the point of throwing in terms of program execution. The catch block can be, and usually is, located in a different function than the point of throwing.
C++ exception handling is built upon three keywords: try, catch, and throw.
Try is used to preface a block of statements which may generate exceptions. This block of statements is known as try block. When an exception is detected it is thrown by using throw statement in the try block. Catch block catches the exception thrown by throw statement in the try block and handles it appropriately.

```
#include<iostream>
using namespace std;
int main()
{
int a,b;
cout<<"Enter any two integer values"; cin>>a>>b;
int x=a-b;
try
{
if(x!=0)
{
cout<<"Result(a/x)="<<a/x<<endl;
}
else
{
throw x;
}
}
catch(int ex)
{
cout<<"Exception caught:Divide By Zero \n";
}
}
```

THROWING MECHANISM

When an exception is detected, it can be thown by using throw statement in any one of the following forms

- throw(exception);
- throw exception;
- throw;

CATCHING MECHANISM:

Catch block is as below Catch(data type arg)

```
{
//statements for handling
//exceptions
}
```

Multiple catch statements:

```
try
{
//try block
}
catch(data type1 arg)
{
//catch block1
}
catch(data type2 arg)
{
//catch block2
}
………………
…………….. catch(data typeN arg)
{
//catch blockN
}
```

- When an exception is thrown, the exception handler are searched in order fore an appropriate match.
- It is possible that arguments of several catch statements match the type of an exception. In such cases the first handler that matches the exception type is executed

Write a Program to catch multiple catch statements

```
 #include<iostream.h>
void test(int x)
{

try
{

if(x==1) throw x;
else if(x==0) throw 'x';
else if(x==-1) throw 1.0;
cout<<"end of try block"<<endl;
}

catch(char c)
{
cout<<"caught a character"<<endl;
}
catch(int m)
{

cout<<"caught an integer"<<endl;
}

catch(double d)
{
cout<<"caught a double"<<endl;
}
}
int main()
{
test(1);
test(0);
test(-1);
test(2); return 0;
}
```

Output:

```
caught an integer caught a character caught a double end of try block
```

<u>Catch All Exceptions:</u>

all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them. In such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone.

```
catch(…)
{
………
}
```

Write a Program to catch all exceptions #include<iostream.h>

```
void test(int x)
{
try
{
if(x==0) throw x; if(x==0) throw 'x'; if(x==-1) throw 1.0;
}
catch(...)
{
cout<<"caught exception"<<endl;
}
}
int main()
{
test(-1);

test(0);
test(1);
return 0;
}
```

Re-throwing an Exception:

It is possible to pass exception caught by a catch block again to another exception handler. This I known as Re-throwing.

```
#include <iostream> using namespace std; void MyHandler()
{
try
{
throw "hello";
}
catch (const char*)
{
cout <<"Caught exception inside MyHandler\n"; throw; //rethrow char* out of function
}
}
int main()
{
cout<< "Main start     "<<endl;
try
{
MyHandler();
}
catch(const char*)
{
cout <<"Caught exception inside Main\n";
}
cout << "Main end"; return 0;
}
```

Specifying Exceptions:

Specification of exception restrict functions to throw some specified exceptions only with the use of throw(exception list) in the the header of the function.

General form

```
Type function_name(argument list) throw(exceptions -list)
{
Statements try
{
statements
}

}
```

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

void test(int x) throw(int,float,char)
{
switch(x)
{
case 1:throw x;
break; case 2:throw 'x';
break; case 3:throw double(x);
break; case 4:throw float(x);
break;
}
}
int main()
{
try
{
test(4);//test(4) leads to abnormal termination
}
catch(int i)
{
cout <<"Caught int type exception\n";
}
catch(float f)
{
cout <<"Caught float type exception\n";
}
catch(char c)
{
cout <<"Caught char type exception\n";
}
catch(double i)
{
cout <<"Caught Double type exception\n";
}
return 0;
}
```

File Handling in C++

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

We can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;
   cout << "Enter your age: ";

   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.

   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

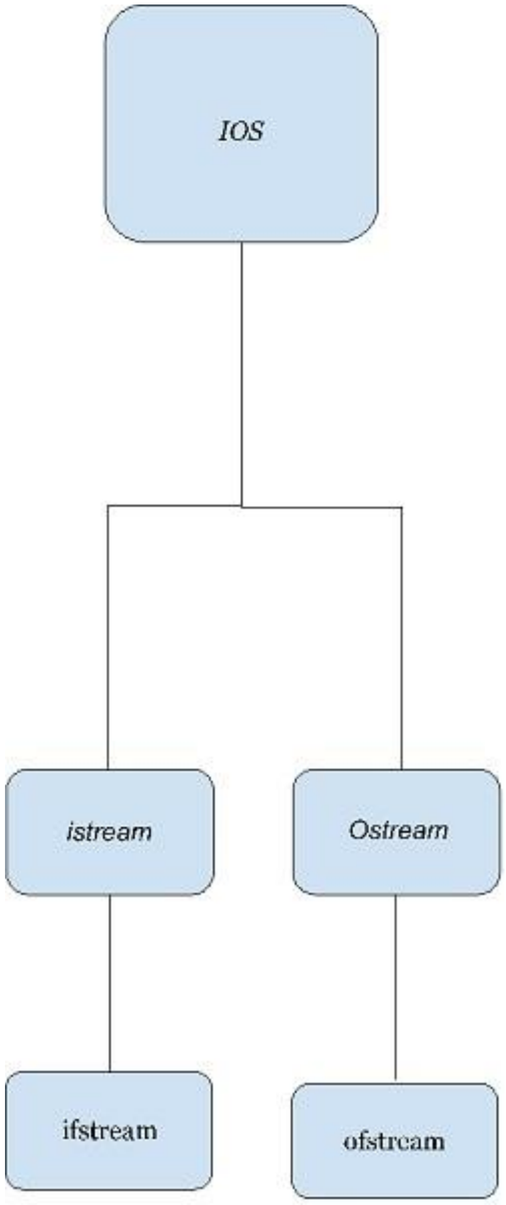
Stream Class

In C++ stream refers to the stream of characters that are transferred between the program thread and i/o.

Stream classes in C++ are used to input and output operations on files and io devices. These classes have specific features and to handle input and output of the program.

The **iostream.h** library holds all the stream classes in the C++ programming language.

Let's see the hierarchy and learn about them,

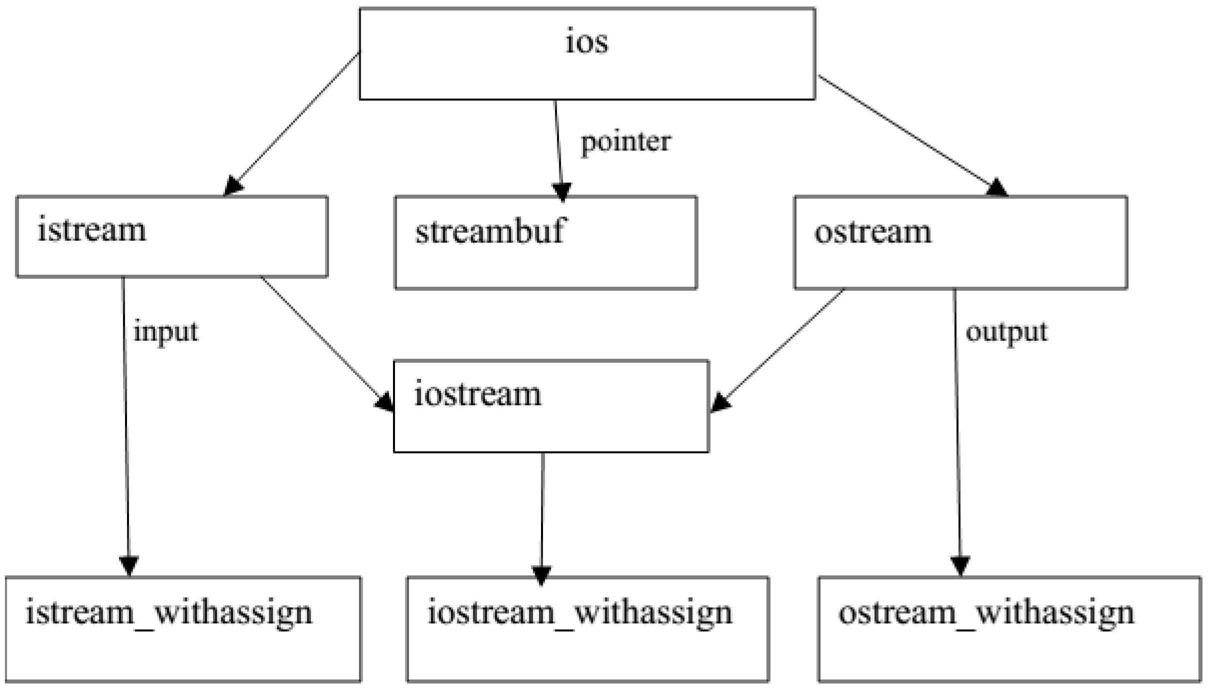


Classes of the **iostream** library.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.



Stream classes for console I/O operations

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

Example

OUT STREAM

COUT

```
#include <iostream>
using namespace std;
int main(){
   cout<<"This output is printed on screen";
}
```

Output

```
This output is printed on screen
```

PUTS

```
#include <iostream>

using namespace std;

int main(){

   puts("This output is printed using puts");

}
```

Output

```
This output is printed using puts
```

IN STREAM

CIN

```
#include <iostream>

using namespace std;

int main(){

   int no;

   cout<<"Enter a number ";

   cin>>no;

   cout<<"Number entered using cin is "<
```

Output

```
Enter a number 3453
Number entered using cin is 3453
```

gets

```
#include <iostream>

using namespace std;

int main(){

   char ch[10];
```

```
    puts("Enter a character array");

    gets(ch);

    puts("The character array entered using gets is : ");

    puts(ch);

}
```

Output

```
Enter a character array
thdgf
The character array entered using gets is :
thdgf
```

C++ Question Bank(Unit-Wise)

UNIT-1

1. What is the need of Object Oriented Programming paradigm? Write difference between Object Oriented and procedural Programming Language.

2. Describe Characteristics of Object oriented Programming

3. Describe data types in C++ in Details

4. Write a short note on visibility specifiers in C++

5. Give classification of operators available in C++ with the help of neat and clean diagram.

6. Draw a diagram to represent the basic structure of a programming C++.

7. What are the different uses of scope resolution operator? Explain with example

8. What do you mean by class and object? Write class structure to define class and object.

9. What do you mean by member function of a class? What are the basic methods to define member function of a class? Explain with example.

10. Write a program to display 5 student details using class and object.

UNIT-2

1. Explain Friend function with example. Also write the merits and demerits of using the friend function?

2. Explain function overloading with example.

3. What is Constructor? Explain types of Constructor with example.

4. Explain following with respect to C++ wit examples. 1)new operator 2) destructor 3) this pointer

5. List out the advantages of new operator over malloc().

6. What is the need of passing objects as arguments? Discuss different ways to pass objects as arguments to a function.

7. Write a program to add two distances using friend function.

8. Discuss default constructor and parameterized constructor with the help of an example in C++

9. What do you mean by constructor? Write a program to demonstrate constructor overloading.

10. what are the default arguments in c++? Explain with example.

UNIT-3

1. what do you mean by inheritance? Explain the types of inheritance with diagram.

2. what is derived class in C++. Write and explain syntax of derived class with example

3. what is multiple inheritance? Explain with Example.

4. what is the role of protected data members in inheritance. Explain with example

5. what is diamond problem in c++. Explain with clean and neat diagram. Also write an example to solve diamond problem in c++ .

6. what are the ambiguity in multiple inheritance. Explain solution of ambiguity using example.

7. Write a C++ program demonstrating use of the pure virtual function with the use of base and derived classes.

8. Explain Virtual functions with example.

9. What are the difference between function overloading and function overriding?

10. Explain abstract class with example

UNIT-4

1. What is the significance of static data and member functions in C++? Write a program to demonstrate static keyword.

2. Define Polymorphism .What is the difference between static & dynamic binding?

3. Define operator overloading? Explain how to overload unary operator.

4. Which operators cannot be overloaded? Write steps to overload + operator so that it can add two complex numbers.

5. Explain the concept of virtual destructors

6. Write a program to demonstrate the use of Constant keyword.

7. What is operator keyword in C++. Write a program to overload * operator using friend operator overloading.

8. List the operators that cannot be overloaded. Define a complete class by name distance with feet and inches as data member and overload += operator and two objects

9. Define operator overloading. How many arguments are required to overload unary and binary operators, respectively? Write syntax for each .

10. With the help of suitable code explain runtime polymorphism.

UNIT-5

1. Why exception handling is required? Explain with suitable examples

2. how can you define a catch statement that catches all type of exceptions? Illustrate the use of multiple catch statement with the help of example.

3. what do you mean by generic programming in c++.

4. write a program to copy content of one text file to another text file.

5. Define try, catch and throw statement in exception handling using suitable example.

6. Define function template. Differentiate function template with overloaded function.

7. What is Runtime error, Logical error and syntax error.

8. write short note on class template

9. write a program to display content of character and integer array using class template.

10. Write a program to swap the elements using function template.

List of Text Books

1. Balagurusamy – Object Oriented Programming with C++,seventh edition, McGraw Education, 2017.
2. K.R. Venugopal, Rajkumar, T Ravishankar, “Mastering C++”,McGraw Hill Publishing Co. Ltd, 2006.

List of Reference Books

1. Robert Lafore, Object Oriented Programming in Turbo C++,First Edition, Galgotia Publications.
2. Herbert Shildt, " C++ : The complete reference", Fourth Edition,McGraw Hill Education, 2017