SCHEME OF VALUATION
(Scoring Indicators)

Revision: 15 **Course Code: 3134**

Course Title: OBJECT ORIENTED PROGRAMMING THROUGH C++

Qst. No.	Scoring Indicator	Split up score	Sub Total	Total
I	**PART – A**	Conditional operator : 1 mark + Syntax: 1 mark	2 Marks	10 marks
1.	• A ternary operator has three operands. The conditional operator is a ternary operator. • Syntax operand1 ? operand2 : operand3			
2.	• `strcpy()` :It implements assignment for character arrays. • `strcat()` :It implements the concatenation of two strings.	1 mark for each	2 Marks	
3.	• Destructor is a class member function, with the same name as the class name preceded by a tilde (~) symbol. • Used to perform explicit memory de-allocation to any member of the object.	2 marks	2 Marks	
4.	• A friend function is a nonmember function that has the same access rights to class members as does any member function. • A friend function can access private (or protected) member functions and private (or protected) data members.	2 marks	2 Marks	

5.	A virtual function whose declaration ends with = 0 is called a pure virtual function.
	Example:
	Class Shape
	{
	public:
	virtual float calculatearea() = 0;
	};
		2 marks	2 marks	

II	**PART – B**			
1.	**Arrays as Homogeneous Aggregates** ✓ Array is a defined data type. ✓ An *array* is a finite collection of elements of the same data type in contiguous memory locations. ✓ All elements of the array have to be of the same type. This is why arrays are known as homogeneous aggregates.	Array: 2 marks + Features /	6 Marks	30 marks

	✓ Arrays are ordered collections of data. This means that each element of the array has the previous element and the next element.
	✓ ***Index***: The array has a name, and the individual array elements are accessing using the name of the array appended with the subscript (or index), the position of this element in the ordered collection.
	✓ Arrays are finite. The number of elements in the array has to be known at compile time and cannot be changed during program execution.
	Defining arrays
	Array definitions cause memory for the array to be allocated during execution time. The syntax of array definition is.
	data_type array_name[size];
	Example:
	int hours[7]; char name[20]; double amount[20];
	Operations over Arrays
	All operations on arrays can be performed over individual array elements only. In array operations, individual array elements are referred by using the subscript operator and the index value.
	Example:
	`side[2] = 40; // use as lvalue`
	`num = side[2] * 2; // use as rvalue`
		description:
		2 marks
		+
		Definition &
		access:
		2 marks		

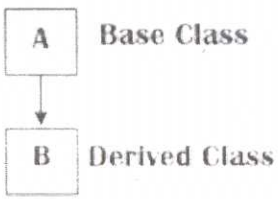
2.	**Program**
	#include <iostream>
	using namespace std;
	int main()
	{
	int num, N, i;
	cout<<"Enter the number: ";
	cin>>num;
	cout<<"Enter the limit: ";
	cin>>N;
	for(i = 1; i <= N; i++)
	{
	cout<<i<<" * "<<num<<" = "<<(num * i)<<endl;
	}
	return 0;
	}
		Preprocessor:
		1 mark
		+
		Main fn.
		1 mark
		+
		Var.
		declaration:
		1 mark
		+
		Read input:
		1 mark
		+
		Any loop:
		1 mark
		+
		Display
		result:
		1 mark
			6
			marks	

3.	• An inline function is a function, which replaces the function call with the statements in the function definition. • If a function is inline, the compiler places a copy of the code of			

	that function at each point where the function is called, at compile time.
	• The keyword **inline** is used before the function name and define the function befsore any calls are made to the function.
	• Example: (any similar example)
	`#include <iostream>`
	`using namespace std;`
	`inline int Max(int x, int y)`
	`{`
	`  return (x > y)? x : y;`
	`}`
	`int main()`
	`{`
	`  cout << "Max (20,10): " << Max(20,10) << endl;`
	`  cout << "Max (0,200): " << Max(0,200) << endl;`
	`  return 0;`
	`}`
	The output is
	`Max (20,10): 20`
	`Max (0,200): 200`
		Inline:
		2 marks
		+
		Explanation:
		2 marks
		+
		Example:
		2 marks
			6
			marks	

4.	Basic class syntax
	✓ A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces.
	✓ Syntax
	`class class_name`
	`{`
	`  Access_specifier1:`
	`  Member_variable;`
	`  Member_functions;`
	`  Access_specifier2:`
	`  Member_variable;`
	`  Member_functions;`
	`};`
	Class Access Modifiers (Access Specifiers)
	✓ It determines the access attributes of the members of the class that follows it.
	✓ There are three access modifiers: public, protected & private.
		Class syntax:
		2 marks
		+
		Description:
		3 marks
		+
		Example:
		1 mark
			6
			marks	

	Class Member Functions
	✓ A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
	✓ Member functions can be defined in two ways:
	▪ within the class definition or
	▪ separately using **scope resolution operator**
	✓ Example
	`class Box {`
	`  public:`
	`  double length;  // Length of a box`
	`  double breadth;  // Breadth of a box`
	`  double height;  // Height of a box`
	`};`			

5.	✓ Inheritance is the capability of one class to acquire properties and characteristics of another class.
	✓ The class whose properties are inherited by other class is called the ***Parent*** or ***Base*** or ***Super class***.
	✓ The class which inherits properties of other class is called ***Child*** or ***Derived*** or ***Sub class***.
	 
	Single Inheritance
	• A derived class with only one base class is called **single inheritance**.
	• It is the simplest form of inheritance.
	• It is implemented as:
	`class A`
	`{`
	`  //members of class A`
	`};`
	`class B:public/protected/private A`
	`{`
	`  //members of class B`
	`};`
		Base class & derived class: 3 marks
		+
		Single inheritance: 3 marks
			6 marks	

6.	**Inheritance** ✓ Inheritance is a way to form new classes using classes that have already been defined.	Inheritance: 3 marks	6	

	✓ Inheritance is the process of passing the attributes and behaviors of one class down to its descendant classes. ✓ One class can use features from another class to extend its functionality (an "Is a" relationship) i.e., a Car is a Automobile. **Composition** ✓ Object composition is a way to combine simple objects or data types into more complex ones. ✓ Composition is the process of making one class as a data member of another class. ✓ Allowing a class to contain object instances in other classes so they can be used to perform actions related to the class (an "has a" relationship) i.e., a person has the ability to walk.	+ Composition: 3 marks	marks	

7.	**Type cast operators**
	✓ A special operator that forces one data type to be converted into another.
	✓ As an operator, a cast is unary and has the same precedence as any other unary operator.
	✓ General form
	`(type) expression`
	where `type` is the desired data type.
	✓ Example program
	`#include <iostream>`
	`using namespace std;`
	
	`main()`
	`{`
	`double a = 21.09399;`
	`float b = 10.20;`
	`int c ;`
	
	`c = (int) a;`
	`cout << "Line 1 - Value of (int)a is :" << c << endl ;`
	
	`c = (int) b;`
	`cout << "Line 2 - Value of (int)b is :" << c << endl ;`
	
	`return 0;`
	`}`
	Output
	`Line 1 - Value of (int)a is :21`
	`Line 2 - Value of (int)b is :10`
		Type cast operator & description: 2 marks + General form: 1 mark + Example: 3 marks	6 marks	

PART C

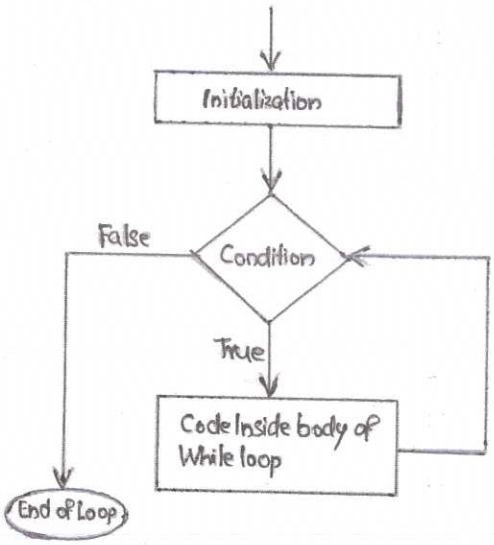
III a)

while Loop

- ✓ A **while** loop repeatedly executes a target statement as long as a given condition is true.
- ✓ The syntax of a **while** loop is:

```
while(condition)
{
    statement(s);
}
```

- ✓ The condition may be any expression. The loop iterates while the condition is true.
- ✓ When the condition becomes false, the program control passes to the line immediately following the loop.
- ✓ It is an entry controlled loop - a while loop might not execute at all, when the condition is tested and the result is false. The loop body will be skipped and the first statement after the while loop will be executed.



Flow diagram

- ✓ Example code segment to find factorial of a number

```
i = 1, factorial = 1;
while ( i <= number)
{
   factorial = factorial * i;
   i = i + 1;
}
```

do…while Loop

- ✓ The **do...while** loop checks its condition at the bottom of the loop. A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.
- ✓ The syntax of a **do...while** loop is:

```
do
```

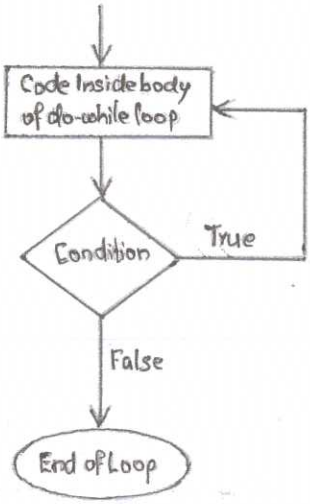
While loop: 4 marks + Do while : 4 marks

(syntax: 1 mark + Explanation or flow diagram: 2 marks + Example (any similar program code): 1 mark)

8 marks

60 marks

```
{
    statement(s);
}while( condition );
```



Flow diagram

- ✓ It is an exit – controlled loop. The conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.
- ✓ If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.
- ✓ Example code segment to find factorial of a number

```
i = 1, factorial = 1;
do
{
        factorial = factorial * i;
        i = i + 1;
} while ( i <= number);
```

III b)

Program

```
#include <iostream>
using namespace std;
int main()
{
  int monthnumber;
  printf("Enter the month number(1 - JAN, 2 - FEB, etc): ");
  scanf("%d",&monthnumber);
  switch( monthnumber )
  {
    case 1:
        printf("The month is January");
        break;
    case 2:
        printf("The month is February");
```

Preprocessor: 1 mark + Main fn. 1 mark + Var. declaration: 1 mark + Read input: 1 mark + Switch: 2 marks + Display

7 marks

```
            break;
        // Add all other case values 3 to 10 here
        case 11:
            printf("The month is November");
            break;
        case 12:
            printf("The month is December");
            break;
        default:
            printf("Invalid data!");
            break;
    }
    return 0;
}
```

result: 1 mark

IV a)

Program

```
#include<iostream>
using namespace std;
struct employee
{
   char name[20], designation[30];
   int age;
   float salary;
};
int main()
{
   int count, i;
   employee e[25];
   cout<<"Enter the number of employees: ";
   cin>>count;
   for(i = 0; i < count; i++)
   {
      cout<<"Enter the details of employee "<<i+1<<endl;
      cout<<"Name : ";
      cin>>e[i].name;
      cout<<"Age : ";
      cin>>e[i].age;
      cout<<"Designation : ";
      cin>>e[i].designation;
      cout<<"Salary : ";
      cin>>e[i].salary;
   }
   cout<<"\n The Employee Details. \n";
   for(i = 0; i < count; i++)
   {
      cout<<"Employee "<<i+1<<endl;
      cout<<"Name : "<<e[i].name<<endl;
      cout<<"Age : "<<e[i].age<<endl;
      cout<<"Designation : "<<e[i].designation<<endl;
      cout<<"Salary : "<<e[i].salary<<endl<<endl;
```

Structure: 2 marks + Array of objects: 1 mark + Read count: 1 mark + Read data: 2 marks + Display data: 2 marks

8 marks

```
    }
    return 0;
}
```

IV b)

Storage classes

- ✓ Defines the scope (visibility) and life-time of variables and/or functions within a C++ Program.
  - Auto
  - Register
  - Static
  - Extern

Storage class: 1 mark + List: 2 marks + Description: 4 marks (1 mark for each)

7 marks

Auto

- ✓ It is default for variables defined as local in a block scope or in a function scope.
- ✓ The memory is allocated on the stack.

```
{
   int count;
   auto int month;
}
```

Register

- ✓ It is used for variables kept in high-speed registers rather than in random-access memory.
- ✓ This means that the variable has a maximum size equal to the register size and can't have the unary '&' operator applied to it.

```
{
   register int  miles;
}
```

Static

- ✓ The **`static`** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.
- ✓ Making local variables **`static`** allows them to maintain their values between function calls.
- ✓ When making global variables static, it causes that variable's scope to be restricted to the file in which it is declared.

Extern

- ✓ The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files.
- ✓ The *extern* is used to declare a global variable or function in another file.
- ✓ The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions

V a)

Constructor

- ✓ Constructor is a class member function that is used to initialize class objects.
- ✓ The name of the constructor should be the same as the class name.
- ✓ The constructor cannot have a return type. It cannot return any values. A constructor cannot be called explicitly.
- ✓ Constructors are declared are

Class_name(parameter_list);

Here the parameter_list is optional.

Default constructor

- ✓ A default constructor is a constructor that has no parameters, or all the parameters have default values.
- ✓ If no constructors are available for a class, the compiler implicitly creates a default parameterless constructor.
- ✓ Example

```
class sample
{
    public:
        int x,y;
        sample()
        {
            x = 0; y = 0;
        }
};
int main()
{
    sample obj;
    cout<<"Default constructs x,y values are:
"<<obj.x<<", "<<obj.y;
    return 0;
}
```

The output is:

Default constructs x,y values are: 0, 0

Copy Constructor

- ✓ It is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- ✓ The copy constructor is used to:
  - o Initialize one object from another of the same type.
  - o Copy an object to pass it as an argument to a function.
  - o Copy an object to return it from a function.
- ✓ Syntax

```
Classname(const classname &obj)
{
    //body of constructor
}
```

- ✓ Example

Constructor: 2 marks + Default: 3 marks + Copy: 3 marks

8 marks

```
class Cylinder
{
     double radius, height;
  public:
    Cylinder (const Cylinder &c)
    {
      radius = c.radius;  height = c.height;
      cout << "Copy constructor: " <<radius<< ", "<<
height << endl;
     }
};
```

V b)

Function overloading

- ✓ The use of the same function name for different functions with a different number or type of parameter is called function name overloading.
- ✓ The compiler will distinguish among overloaded functions.
- ✓ The compiler knows which function to call in each situation because of the type of the argument.
- ✓ The value of overloaded functions is that they allow related sets of functions to be accessed with a common name.
- ✓ When overloading a function: the type and/or number of the parameters of each overloaded function must differ.
- ✓ We cannot overload function declarations that differ only by return type.
- ✓ Function overloading can be used for functions of the same class in the same way as it is used for functions in the same file.

- ✓ Example: (any similar example program)

Example of using the same function name, add(), for two different functions. The number of parameters is different: One function has two parameters, another function has three parameters.

```
int add(int x, int y)        // two parameters
{
        return x + y;
}
int add(int x, int y, int z)  // three parameters
{
        return x + y + z;
}
```

Function overloading: 1 mark + Explanation: 3 marks + Example: 3 marks

7 marks

VI a)

Program

```
#include<iostream>
using namespace std;
class circle
{
   float radius, area;  //data members
  public:
    void getdata();    //member function
    void calculate();  //member function
```

Preprocessor: 1 mark + Class: 2 marks +

8 marks

```
    void display();    //member function
};

void circle :: getdata()
{
   cout<<"\n Enter the value of Radius : ";
   cin>>radius;
}
void circle :: calculate()
{
   area = 3.14 * radius * radius;
}
void circle :: display()
{
   cout<<"\n Area of Circle : "<<area;
}
int main()
{
   circle cr;  //object created
   cr.getdata();      //calling function
   cr.calculate();
   cr.display();
   return 0;
}
```

Function definition outside class: 3 marks (1 mark for each fn.) + Main fn.: 2 marks

VI b)

Passing parameter by value

- ✓ This method copies the actual value of an argument into the formal parameter of the function.
- ✓ In this case, changes made to the parameter inside the function have no effect on the argument.
- ✓ By default, C++ uses call by value to pass arguments.

Passing parameter by pointer

- ✓ The calling by pointer method of passing arguments to a function copies the address of an argument into the formal parameter.
- ✓ Inside the function, the address is used to access the actual argument used in the call.
- ✓ This means that changes made to the parameter affect the passed argument.

Example: function swap() which exchanges the values of the two integer variables.

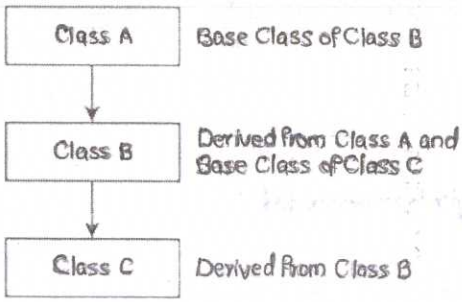
Operation	Call by value	Call by pointer
Function declaration	void swap(int x, int y);	void swap(int *x, int *y);
Function call	swap(a, b);	swap(&a, &b);

Call by value Explanation: 2 marks + Call by pointer Explanation: 2 marks + Example program or explanation: 3 marks (1 ½ marks for each)

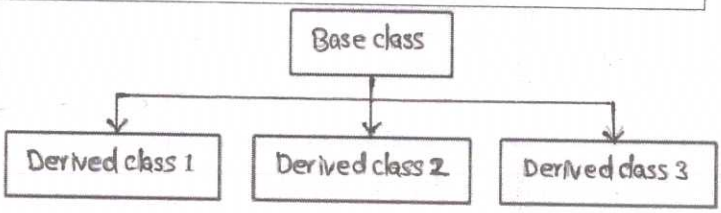
7 marks

	Function definition	void swap(int x, int y)
		{
		int temp;
		temp = x;
		x = y;
		y = temp;
		return;
		}
			void swap(int *x, int *y)
			{
			int temp;
			temp = *x;
			*x = *y;
			*y = temp;
			return;
			}		

	Example program or explanation including the data in this table.				

VII a)	**Multilevel inheritance**
	✓ A derived class with one base class and that base class is a derived class of another is called *multilevel inheritance.*
	 
	
	Fig. Multilevel Inheritance
	✓ It can be implemented as:
	`Class A`
	`{`
	`  //members of class A`
	`};`
	`Class B : public/protected/private A`
	`{`
	`  //members of class B`
	`};`
	`Class C : public/protected/private B`
	`{`
	`  //members of class C`
	`};`
	Hierarchical inheritance
	✓ Multiple derived classes with same base class is called *hierarchical inheritance.*
	`Class A`
	`{`
	`  //members of class A`
	`};`
	`Class B : public/protected/private A`
	`{`
	`  //members of class B`
	`};`
		Multilevel: 4 marks
		+
		Hierarchical: 4 marks
			8 marks	

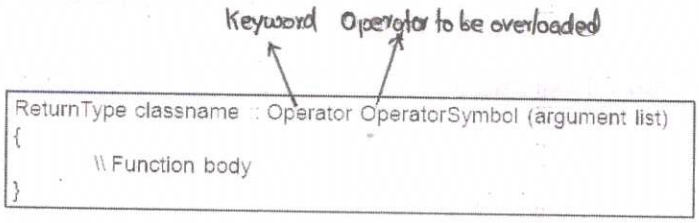
```
Class C : public/protected/private A
{
    //members of class C
};
```



VII b)

Operator overloading

- ✓ It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.
- ✓ Overloaded operator is used to perform operation on user-defined data type.
- ✓ For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.
- ✓ Operator overloading syntax:



Limitations on Operator Overloading

Operators that cannot be overloaded are:

- ✓ Scope resolution operator **::**
- ✓ **Sizeof**
- ✓ Member selection **.**
- ✓ Member pointer selector *
- ✓ Ternary operator **?:**

These are some restrictions on operator overloading.

- ✓ Precedence and Associativity of an operator cannot be changed.
- ✓ Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
- ✓ No new operators can be created, only existing operators can be overloaded.
- ✓ Cannot redefine the meaning of a procedure. User cannot change how integers are added.

Operator overloading: 3 marks + Limitations: 4 marks

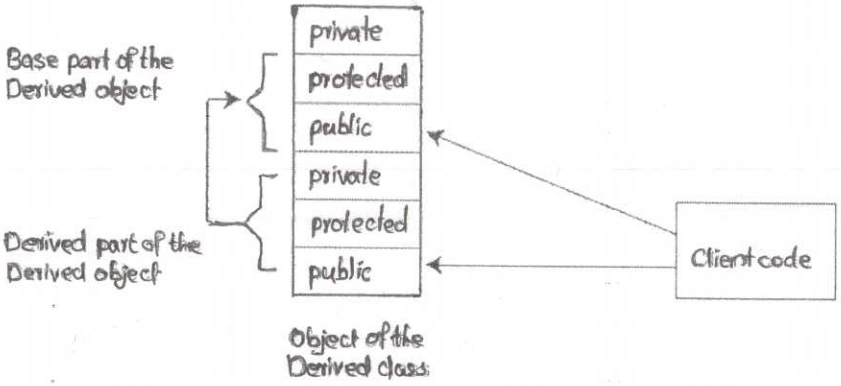
7 marks

VIII a)	**Program**
	`#include<iostream>`
	`#include<string.h>`
	`using namespace std;`
	
	`class String`
	`{`
	`private:`
	`char str[50];`
	`public:`
	`void getstring()`
	`{`
	`cout<<"\n Enter String : ";`
	`cin.getline(str,50);`
	`}`
	`void display_string()`
	`{`
	`cout<<str;`
	`}`
	`String operator+(String x) //Concatenating String`
	`{`
	`String s;`
	`strcat(str,x.str);`
	`strcpy(s.str,str);`
	`return s;`
	`}`
	`};`
	
	`int main()`
	`{`
	`String str1, str2, str3;`
	`str1.getstring();`
	`str2.getstring();`
	
	`cout<<"\nFirst String is: ";`
	`str1.display_string(); //Displaying First String`
	
	`cout<<"\n Second String is: ";`
	`str2.display_string(); //Displaying Second String`
	
	`str3 = str1+str2; //Concatenation.`
	`//Overloaded '+' operator`
	`cout<<"\n Concatenated String is : ";`
	`str3.display_string();`
	`return 0;`
	`}`
		Preprocessor: 1 mark + class: 3 marks + Main function: 4 marks	8 marks	

VIII b)	**Mode of derivation** ✓ Access to the base components of a derived class object depending on the mode of derivation of the derived class. ✓ Each base class can be inherited through private, protected, or public mode of inheritance.	Mode of derivation: 1 mark +	7 marks	

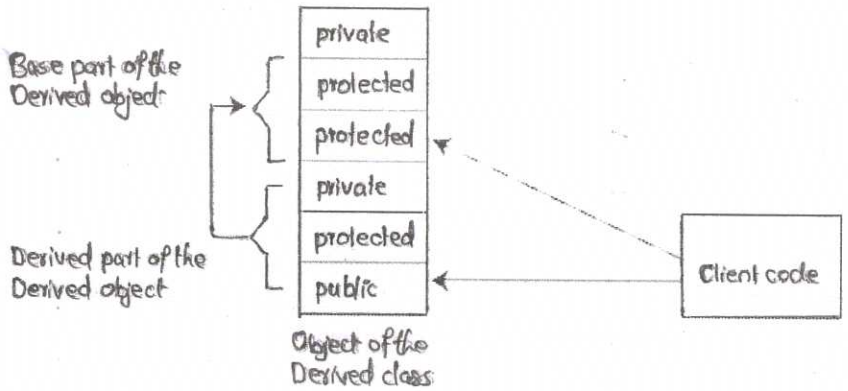
Public Inheritance

- ✓ Access status remains the same.
- ✓ Public, protected, and private base members remain public, protected, and private in an object of the derived class.
- ✓ This is the least restrictive case; nothing changes.
- ✓ Hence, derived class methods can access protected and public base members of a derived object.



Protected Inheritance

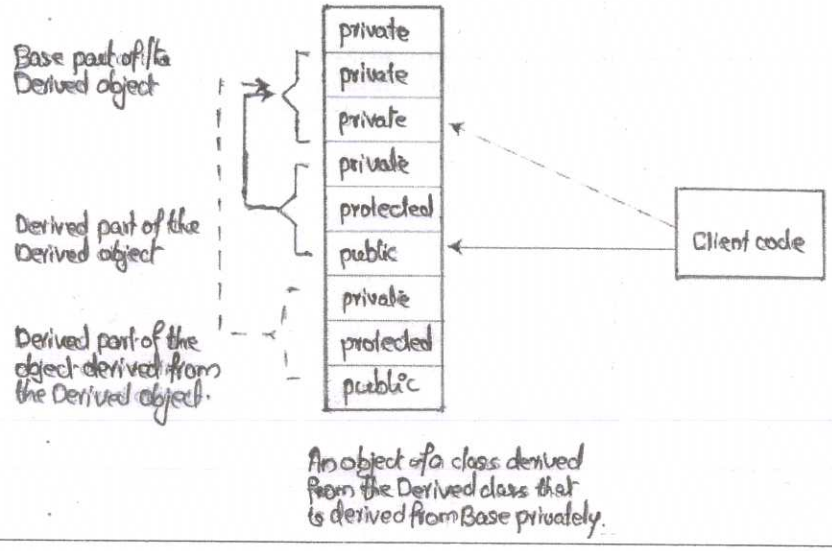
- ✓ Protected inheritance is the mechanism that limits client access to the services of the base class.
- ✓ Public and protected members inherited from the base class become protected in a derived class object.



Private Inheritance

- ✓ Private inheritance is a technique for limiting access to base services.
- ✓ It limits access by the clients of the derived classes and also by classes derived from the derived classes.
- ✓ When the base class is used as a private base, all public and protected base members become private members in a derived class object.

Public inheritance: 2 marks + Protected inheritance: 2 marks + Private inheritance: 2 marks



An object of a class derived from the Derived class that is derived from Base privately.

IX a)

Multiple inheritance

- ✓ A derived class with multiple base classes.
- ✓ The derived class inherits all data members of all bases and all member functions of all bases.

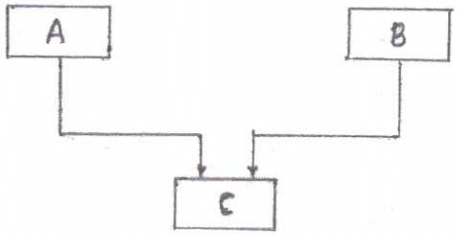


Figure: multiple inheritance

```
Class A
{
  //members of class A
};

Class B
{
    //members of class B
};

Class C : public/protected/private A ,
          public/protected/private B
{
    //members of class C
};
```

- ✓ The access rights can change depending on the mode of derivation.

Modes of derivation for multiple inheritance

- With ***public derivation***, Public, protected, and private base members remain public, protected, and private in an object of the derived class.
- With ***protected derivation***, Public and protected members inherited from the base class become protected in a derived class

Multiple inheritance: 3 marks + Mode of derivation: 2 marks + Example program: 3 marks

8 marks

	object. Protected inheritance limits client access to the services of the base class. • With ***private inheritance***, all base members become private in the derived class. The mode of derivation is private by default. ✓ Example program			
IX b)	✓ Using virtual function. ✓ A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class. ✓ Mainly used to achieve Run time polymorphism. ✓ Explanation ✓ Base class pointer that holding the address of derived class object.	Virtual function & run time polymorphism: 2 marks + Program: 3 marks + Explanation: 2 marks	7 marks	
X a)	**Exception** ✓ A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. ✓ Exceptions provide a way to transfer control from one part of a program to another.	Exception: 1 mark + Try, catch,	8 marks	

```
#include<iostream>
using namespace std;
class Base
{
 public:
     virtual void display()
     {
      cout << "Base class is invoked.";
     }
};
class Derived:public Base
{
 public:
     void display()
     {
      cout << "Derived Class is invoked.";
     }
};

int main()
{
 Base* b;        //Base class pointer
 Derived d;      //Derived class object
 b = &d;
 b->display();      //Late Binding Occurs
}
```

Output

Derived class is invoked

C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- ***throw*** – A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- ***catch*** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- ***try*** – A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

✓ A try/catch block is placed around the code that might generate an exception.

✓ Code within a try/catch block is referred to as protected code.

✓ The syntax for using try/catch as follows

```
try
{
   // protected code
} catch( ExceptionName e1 )
{
   // catch block
}
catch( ExceptionName e2 )
{
   // catch block
}
catch( ExceptionName eN )
{
   // catch block
}
```

Throwing Exceptions

✓ Exceptions can be thrown anywhere within a code block using **throw** statement.

✓ The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

✓ An example of throwing an exception when dividing by zero condition occurs

```
double division(int a, int b)
{
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

throw:
1 mark
+
Syntax of try/catch:
2 marks
+
Throwing exception:
2 marks
+
Catching exception:
2 marks

	Catching Exceptions ✓ The **catch** block following the **try** block catches any exception. ✓ Here we can specify what type of exception we want to catch. ✓ This code will catch an exception of *ExceptionName* type.			
X b)	**Template class** ✓ Templates are the foundation of generic programming. ✓ It involves writing code in a way that is independent of any particular type. ✓ *Template* class is a method to reuse class design. ✓ Instead of the class with a fixed type of component, create the class where the type of component is treated as a class parameter. ✓ The general form of a generic class declaration is : ✓ Here, **type** is the placeholder type name, which will be specified when a class is instantiated. ✓ We can define more than one generic data type by using a comma-separated list. ✓ This parameter has a programmer-defined name, for example, `Type, T, Tp,` and so on. ✓ Its actual value can be any type, built-in or programmer-defined. ✓ The actual value cannot be known at the time of compiling the template definition. ***Template Instantiation*** ✓ Creating an object of a template class is called *instantiation*. ✓ The actual type is specified at template instantiation as the type name in angle brackets that is appended to the name of the template class. ✓ Example:	Template: 1 mark + Template class: 2 marks + General form & explanation: 2 marks + Template instantiation & example: 2 marks	7 marks	

```
try
{
   // protected code
}
catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

```
template <class type> class class-name
{
   :
}
```

```
class-name<data_type> obj-name;
```