2079

Created	@April 11, 2023 9:38 AM
Class	OOPL
Type	Exam Material
Materials	
Reviewed	☐

Group A

1. What is aggregation? Write a program for implementing following:

Create a class author with attributes name and qualification.

Also create a class publication with pname.

From these classes derive a classes derive a class book having attributes title and price.

Each of the three classes should have getdata() method to get their data from user.

The classes should have putdata() method to display the data. Create instance of the class book in main.

Association:

Association is a simple structural **connection or channel between classes** and is a relationship where all objects have their **own lifecycle and there is no owner**.

Example:

Multiple patients can associate with a single doctor and single patient can associate with multiple doctors, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Aggregation:

Aggregation is a **specialized form of Association** where all objects have their **own lifecycle but there is a ownership like parent and child.** Child object can not belong to another parent object at the same time. We can think of it as `“has-a”` relationship. In the simplest possible terms, it is when a class has an object of the other class.

Implementation details:

- Typically, we use pointer variable that point to an object that lives outside the scope of the aggregate class.
- Can use reference values that point to an object that lives outside the scope of the aggregate class.

Example:

A Employee can not belong to multiple Companies, but if we delete the company, employee object will not get destroyed.

Composition:

Composition is again a **specalized form of Aggregation.** It is a **strong** type of Aggregation. Here the `Parent` and `Child` objects **have coincident lifetimes**. `Child` object **doesn't have it's own lifecycle** and **if parent object gets delete, then all of it's child objects will also be deleted**.

Implementation:

Use normal member variables.

Example:

Relationship between person and it's birthday. There is no independdent life for birthday. If we delete the person, birthday will also be automatically deleted.

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

class author {
    public:
        char name[20], qualification[20];

        void getdata() {
            cout << "\nEnter Author Name: ";
            cin >> name;

            cout << "Enter Author Qualification: ";
            cin >> qualification;
        }

        void putdata() {
            cout << "Author Name: " << name;
            cout << "\nAuthor Qualification: " << qualification;
        }
};

class publication {
    public:
        char pname[20];

        void getdata() {
            cout << "Enter Publication Name: ";
            cin >> pname;
        }

        void putdata() {
            cout << "\nPublication Name: " << pname;
        }
};


class book: public author, public publication {
    public:
        char title[20];
        float price;

        void getdata() {
            cout << "Enter Book Title: ";
            cin >> title;

            cout << "Enter Book price: ";
            cin >> price;
        }

        void putdata(){
            cout << "\nBook Name: " << title;
            cout << "\nBook Price: Rs." << price;
        }
};


int main() {
    // Created an object of class book
    book obj;

    // read the data for each class by scope resolution
    obj.author::getdata();
    obj.publication::getdata();

    // read the data for class book itself
    obj.getdata();

    // display result of each class by scope resolution
    obj.author::putdata();
    obj.publication::putdata();
    // display result of the class book
    obj.putdata();

    return 0;
}
```

---

2. What is operator overloading? Why it is necessary to overload an operator? Write a program for overloading comparision operator.

Operator overloading in C++ allows you to **define how an operator works for a specific class or data type**. It allows you to **use the same operator for different operations** depending on the context of its use.

In other words, operator overloading allows you to redefine what an operator does when it is used with objects of a particular class.

For example, you can overload the `'+'` operator to perform addition of two objects of your class instead of just adding two numbers.

It makes the program more readable and easier to understand by allowing manipulation of objects of class's type to behave in the same manner as that of variables of built-in types. For example: if we want to add two complex numbers, then we need to define member function add() and call it using the following notation.

```
c3.add(c1, c2);
// or
c3 = c1.add(c2);
```

The above statements are obscure as the programmer needs to remember the function's name and how to call it. The problem worsens even more when the program becomes large and complex. A better alternative for performing the addition of two complex numbers is to overload the '+' operator. So the above statements can be written in an easily readable form as

```
c3 = c1+c2;
```

- Operator overloading is useful in large and complex programs involving multiple objects of different classes. It provides a common interface corresponding to an operator that performs similar operations on various objects. For example: In a program, the `+` operator can add two-time class objects, two date class objects etc.

```
time1 = time2 + time3;
date1 = date2 + date3;
```

- Debugging becomes easier as the code becomes more understandable and clearer.
- Operator overloading helps to gain greater control over object's behaviour in your program (i.e. controlling the lifetime of objects). For example, – You can overload memory allocation and deallocation functions for your classes to specify exactly how memory should be distributed and reclaimed for each new object.

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

class Comp {
    int price;
    public:
        Comp(int p = 0) {
            price = p;
        }
        bool operator ==(Comp a) {
            if (price == a.price) {
                return true;
            } else {
                return false;
            }
        }
};


int main() {
    Comp c1(200), c2(200);

    if (c1 == c2) {
        cout << "Both are equal!";
    } else {
        cout << "Not equal!";
    }

    return 0;
}
```

---

3. What is the use of constructor and destructor? Write a program for illustrating default constructor, parameterized constructor and copy constructor.

Constructors are special member functions that are called automatically when an object of a class is **created**. They are used to initialize the object's member variables and prepare it for use. Constructors have the same name as the class and do not have a return type, **not even void**.

Destructors are also special member functions that are called automatically when an object of a class is **destroyed**. They are used to perform any necessary cleanup operations, such as releasing resources or freeing memory, before the object is removed from memory. Destructors have the same name as the class, preceded by a tilde (`~`), and also do not have a return type.

The last object created gets destroyed first.

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

class Cons {
    int a;
    public:
        Cons() { // default constructor
            a=0;
        }

        Cons(int x) { // parameterized constructor
            a=x;
        }

        Cons(Cons &x) { // copy constructor (remember `&` operator)
            a = x.a;
        }

        void display() {
            cout << "a: " << a << endl;
        }
};


int main() {
    // default constructor
    Cons a1;
    a1.display();

    // parameterized constructor
    Cons a2(5);
    a2.display();

    // copy constructor
    Cons a3(10), a4;
    a4 = a3;
    a3.display();
    a4.display();

    // or
    Cons a5(20);
    Cons a6(a5); // remember Cons a6(a5); nai huna parcha, Cons a6 eauta line ma a6(a5) arko line ma huna hudaina
    a5.display();
    a6.display();

    return 0;
}
```

Group B

4. Describe the characteristics of object oriented programming language.

https://hamrocsit.com/note/oops/introduction-to-oops/

5. Define class and object with suitable example. How members of class can be accessed?

📝 Class and objects are in mathi ko question ko answer ma.

In C++, a class is a blueprint or template for creating objects that encapsulate data and behavior. Objects are instances of a class, which means they are created using the blueprint specified by the class.

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

class MyClass {
    public:
        int value;
        int roll;
};

int main() {
    MyClass myobj;

    myobj.value = 20;
    myobj.roll = 1;

    cout << "Value: " << myobj.value << "\nRoll: " << myobj.roll;

    return 0;
}
```

The members of a class can be accessed using the member access operator (`.`) or the pointer-to-member operator (`->`) depending on how the object is accessed.

1. Using the dot operator (`.`): When you have an object of a class, you can access its members using the dot operator followed by the name of the member. For example, if you have a class called `Person` and an object of that class called `p`, you can access the `age` member variable of `p` using `p.age`.

   📝 Exactly Like above example for class and objects

2. Using the arrow operator (`->`): When you have a pointer to an object of a class, you can access its members using the arrow operator followed by the name of the member. For example, if you have a class called `Person` and a pointer to an object of that class called `p_ptr`, you can access the `age` member variable of the object using `p_ptr->age`.

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

class Person {
    public:
        int age;
};

int main() {
    Person p;
    Person* p_ptr = &p;
    p_ptr->age = 25;
    cout << "Age: " << p_ptr->age << endl;
    delete p_ptr;
    return 0;
}
```

If the data member is defined as **private or protected**, then we cannot access the data variables directly. Then we will have to create special public member function to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

6. What is inline function? Why it is used? Write a program to illustrate inline function.

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call instead of being called as a separate function. This substitution is performed by the C++ compiler at compile time. **This can result in faster execution time and smaller code size.**

Uses of Inline function:

- When the perfomance is required.
- Ideal for small functions that are used frequently in the code.

```
// fahrenheit to celcius
#include<iostream>
using namespace std;

inline double convert(double f) {
    return ((5.0/9.0) * (f - 32.0));
}

int main() {
    double f=50.0;

    cout << convert(f);

    return 0;
}
```

```
// celcius to fahrenheit
#include<iostream>
using namespace std;

inline double convert(double c) {
    return (c * 1.8) + 32;
}

int main() {
    double c=30.0;

    cout << convert(c);

    return 0;
}
```

7. What are the various class access specifiers? How public inheritance differs from private inheritance?

- `public`: The `public` specifier makes the class members **accessible from anywhere in the program**. This means that the members can be accessed by any function, class or object.
- `private`: The `private` specifier makes the class members **accessible only within the class**. This means that the members can only be accessed by the member functions of the class and not by any other function, class or object.
- `protected`: The `protected` specifier makes the class members **accessible within the class and its derived classes**. This means that the members can be accessed by the member functions of the class and its derived classes, but not by any other function, class or object.

```
class MyClass {
  public: // accessible from anywhere
    int publicVar;

  private: // accessible only within the class only
    int privateVar;

  protected: // accessible within the class and its derived classes
    int protectedVar;

  // member functions can access all class members
};
```

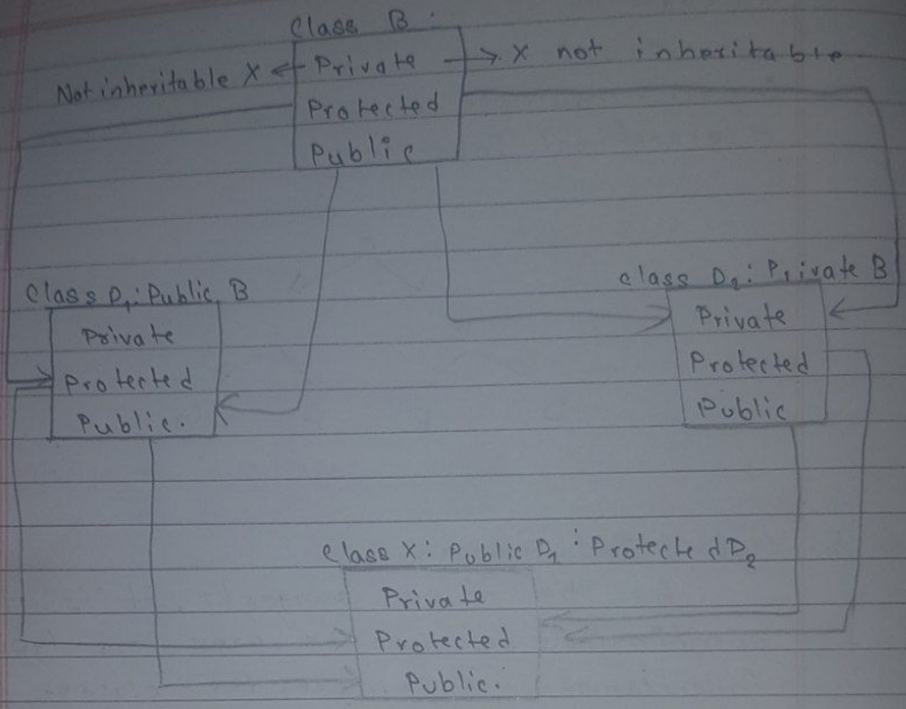
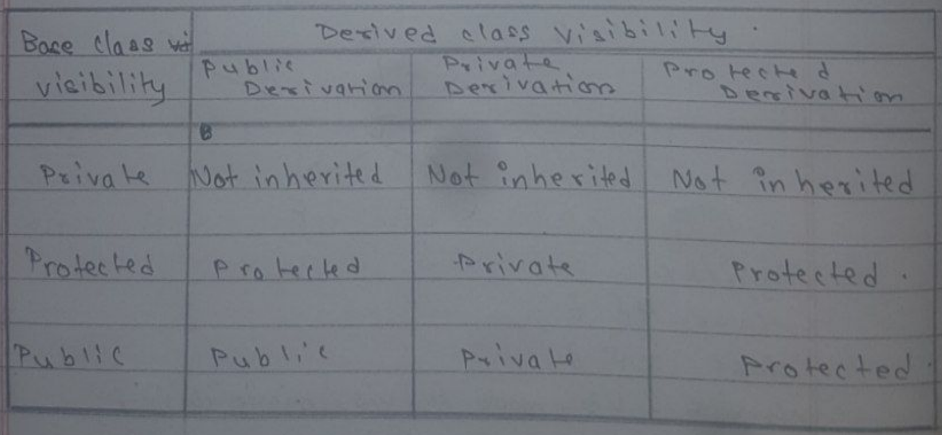


fig : Visibility of members.



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.

Visibility of inherited members.

8. Write a program to implement function template with multiple arguments.

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

template<class T>
void display(T a, T b) {
    cout << "a: " << a << "\nb: " << b << endl;
}
```

```
int main() {
    int a=1, b=2;
    double c=5.99, d=6.99;
    char x='x', y='y';

    display(a, b);
    display(c, d);
    display(x, y);

    return 0;
}
```

---

9. Write a program to illustrate the use of seekg() and tellg().

```
// file(9.txt) contains: hello bachhe loggg
#include<iostream>
#include<fstream>
using namespace std;

int main() {
    ifstream file("9.txt");

    int position = file.tellg();
    cout << "Current position: " << position << endl; // 0

    char ch;
    file.get(ch);
    cout << ch << endl; // h

    file.seekg(2);
    position = file.tellg();
    cout << "Current position: " << position << endl; // 2

    file.get(ch);
    cout << ch << endl; // l

    return 0;
}
```

---

10. How dynamic memory allocation is done using new and delete? Write a program for illustrating use of new and delete.

In C++, dynamic memory allocation allows us to **allocate memory for objects at run-time**. This is useful **when we don't know the size of the object at compile-time or when we need to create objects on the heap instead of on the stack**. Dynamic memory allocation in C++ is done using the `new` and `delet` operators.

The `new` operator is used to allocate memory for an object at run-time.

```
int* myInt = new int; // allocate memory for a single integer
```

In this example, we use the `new` operator to allocate memory for a single integer. The `new` operator returns a pointer to the newly allocated memory, which we store in the `myInt` variable.

To allocate memory for an array of objects, we can use the following syntax:

```
int* myIntArray = new int[5]; // allocate memory for an array of 5 integers
```

In this example, we use the `new` operator to allocate memory for an array of 5 integers. The `new` operator returns a pointer to the first element of the newly allocated array, which we store in the `myIntArray` variable.

After finishing the use of these dynamic memory allocated variables, we must release it using the `delete` operator.

```
delete myInt; // release memory for a single integer
```

In this example, we use the `delete` operator to release the memory that was allocated for a single integer. We pass the pointer to the memory that was allocated using `new` to the `delete` operator.

```
delete[] myIntArray; // release memory for an array of integers
```

In this example, we use the `delete[]` operator to release the memory that was allocated for an array of integers. We pass the pointer to the first element of the array to the `delete[]` operator.

It's important to note that whenever we use `new` to allocate memory, we must use `delete` to release that memory. Failure to do so can result in memory leaks, which can cause the program to run out of memory over time.

> In computer science, the heap is a region of memory that is used for dynamic memory allocation. The heap is an area of memory that is not managed by the program's stack and is used for objects whose lifetime extends beyond the scope of the program's functions or blocks.

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

int main() {
  // Dynamically allocate an integer
  int* p = new int;

  // Assign a value to the integer
  *p = 42;

  // Print the value of the integer
  cout << "The value of the integer is " << *p;

  // Deallocate the integer
  delete p; // don't forget to deallocate

  return 0;
}
```

```
// for array
#include<iostream>
using namespace std;
```

```
int main() {
    // Dynamically allocate an array of integers
    int* intArr = new int[5];

    // Assign values to the array
    for (int i=0; i<5; i++) {
        intArr[i] = 1+i; // stores by adding 1 and i's current value
    }

    // Print the values of the array
    for (int i=0; i<5; i++) {
        cout << intArr[i] << " ";
    }

    // Deallocate the array
    delete[] intArr; // don't forget to deallocate

    return 0;
}
```

11. Write short notes on:

- Friend Function:

  In C++, a friend function is a function that is **not a member of a class but has access to the `private` and `protected` members of the class**. A friend function is declared inside a class with the keyword `friend`. This allows the function to access the `private` and `protected` members of the class as if it were a member of the class itself. The friend function can be defined either inside or outside the class.

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

class MyClass {
    int x;

    public:
        MyClass(int num) : x(num) {}

        // Declare friend function
        friend void printX(MyClass obj);
};

// Define friend function
void printX(MyClass obj) {
    cout << "Value of x is: " << obj.x << endl;
}

int main() {
  MyClass obj(5);

  // Call friend function
  printX(obj);

  return 0;
}
```

- Early binding:

  Early binding **(also known as static binding or compile-time binding)** is the process of linking a function call to the appropriate function definition at compile time. **Early binding is used when the type of the object is known at compile time**. This means that the compiler can determine which function to call based on the type of the object.

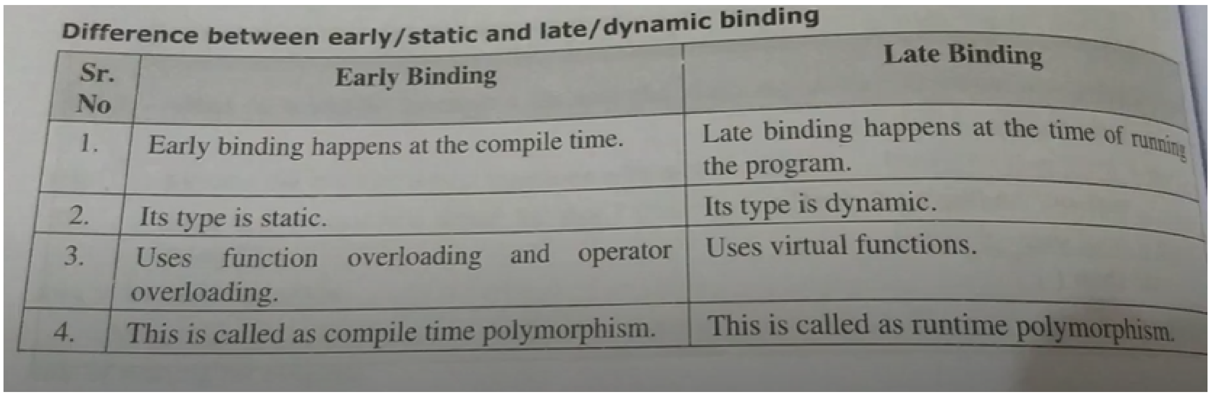
```
#include<iostream>
using namespace std;

class A {
    public:
        void show() {
            cout << "In Base Class" << endl;
        }
};

class B : public A {
    public:
        void show() {
            cout << "In Derived Class" << endl;
        }
};


int main() {
    B b;

    b.show(); // In Derived Class

    b.A::show(); // In Base Class

    return 0;
}
```

We already know which function will be called. So this is called Early Binding.

- Late Binding:

  Late binding (also known as dynamic binding or runtime binding) is the process of linking a function call to the appropriate function definition at runtime. **Late binding is used when the type of the object is not known until runtime**. This means that the compiler cannot determine which function to call based on the type of the object. Instead, the function call is resolved at runtime based on the actual type of the object.

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

class A {
    public:
        virtual void show() {
            cout << "In Base Class" << endl;
        }
};

class B : public A {
    public:
        void show() {
            cout << "In Derived Class" << endl;
        }
};


int main() {
    A* baseptr; // pointer of base class
    B b; // derived class object

    baseptr = &b; // associating ptr with obj

    // if no virtual function is used in base class
    baseptr->show(); // In Base Class, becz the pointer is of base class

    // if virtual function us used in base class, like in this program
    baseptr->show(); // In Derived Class

    return 0;
}
```



Difference between early/static and late/dynamic binding

Sr. No	Early Binding	Late Binding
1.	Early binding happens at the compile time.	Late binding happens at the time of running the program.
2.	Its type is static.	Its type is dynamic.
3.	Uses function overloading and operator overloading.	Uses virtual functions.
4.	This is called as compile time polymorphism.	This is called as runtime polymorphism.