Object oriented programming-2072

SectionA
Attempt any two questions.

1. Explain the object oriented programming with its advantages. What are the features of object oriented languages? Explain.

Ans. Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of "objects". An object is an instance of a class, which is a blueprint or template that defines the characteristics and behaviors of a particular type of object. In OOP, data and functions are grouped together in objects, which can interact with each other to perform tasks.

Advantages of OOP:

1. Modularity: OOP enables developers to break down complex systems into smaller, more manageable modules, or objects. This makes it easier to develop, maintain, and modify software applications.
2. Reusability: OOP promotes code reuse through the use of inheritance and polymorphism. This means that developers can build on existing code rather than having to create new code from scratch.
3. Encapsulation: OOP allows developers to hide the details of an object's implementation from the outside world. This protects the integrity of the object and makes it easier to modify or replace its internal workings without affecting the rest of the system.
4. Flexibility: OOP enables developers to design systems that are more flexible and adaptable to change. Objects can be modified or replaced without affecting the rest of the system, and new objects can be easily added to the system to support new features.
5. Maintenance: OOP makes it easier to maintain software applications over time. By grouping related data and functions into objects, developers can quickly identify and fix bugs and make updates to the system.

Features of Object-Oriented Languages:

1. Abstraction: Abstraction is the process of identifying the essential features of an object and ignoring the non-essential ones. Object-oriented languages provide mechanisms for creating abstract classes and interfaces, which can be used to define common features of related objects.
2. Inheritance: Inheritance is the process of creating new classes based on existing classes. Object-oriented languages support inheritance, which allows developers to create new classes that inherit properties and behaviors from existing classes.
3. Encapsulation: Encapsulation is the process of hiding the internal details of an object from the outside world. Object-oriented languages provide mechanisms for defining public and private members of a class, which can be used to control access to an object's data and methods.

4. Polymorphism: Polymorphism is the ability of objects to take on multiple forms. Object-oriented languages support polymorphism, which allows developers to define methods that can operate on objects of different types.
5. Message passing: In object-oriented programming, objects communicate with each other by sending messages. Object-oriented languages provide mechanisms for sending messages between objects, which can be used to implement complex interactions between objects in a system.

Overall, object-oriented programming provides a powerful and flexible approach to software development that can help developers build complex systems that are easy to maintain and modify over time.

2. Write a program to overload the unary minus operator using friend function.

Ans. Sure! Here's an example program that overloads the unary minus operator (-) using a friend function in C++:

```
#include <iostream>

using namespace std;


class MyClass {

  private:

        int num;


  public:

        MyClass(int n) {

           num = n;

        }


        // Declare the unary minus operator as a friend function

        friend MyClass operator-(MyClass obj);
```

```
        void display() {
        cout << "The number is: " << num << endl;
        }
};

// Define the unary minus operator as a friend function
MyClass operator-(MyClass obj) {
        obj.num = -(obj.num);
        return obj;
}

int main() {
        MyClass obj(10);
        obj.display();

        // Call the unary minus operator using the '-' symbol
        MyClass obj2 = -obj;
        obj2.display();

        return 0;
}
```

In this program, we create a class called **MyClass** that has a private data member **num** and a constructor to set its value. We then declare the unary minus operator as a friend function inside the class.

The friend function **operator-** takes an object of **MyClass** as its parameter and returns a new **MyClass** object with its **num** value negated. Inside the function, we access the private data member of the object using the friend access specifier.

In the **main()** function, we create an object **obj** of **MyClass** with the value 10 and display its value. We then call the unary minus operator using the '-' symbol on **obj**, which returns a new **MyClass** object with the value -10. We assign this new object to **obj2** and display its value using the **display()** function.

The output of the program will be:

The number is: 10

The number is: -10

3. Explain the role of inheritance in object oriented programming. What is public, private and protected dentation? Explain.

Ans. Inheritance is a fundamental concept in object-oriented programming that enables a class to inherit properties and behaviors from another class. Inheritance allows developers to create new classes that are built upon existing classes, reducing code duplication and increasing code reuse.

Inheritance works by creating a relationship between two classes, where one class (the derived or child class) inherits properties and behaviors from another class (the base or parent class). The derived class can then extend or modify the properties and behaviors of the parent class.

The parent class is also known as the base class, and the derived class is also known as the subclass.

There are three types of inheritance in object-oriented programming:

1. Single Inheritance: In single inheritance, a derived class is derived from a single base class.
2. Multiple Inheritance: In multiple inheritance, a derived class is derived from multiple base classes.
3. Hierarchical Inheritance: In hierarchical inheritance, a base class is derived by multiple derived classes.

In C++, we can declare the inheritance relationship between classes using the **class** keyword and the colon (**:**) symbol. For example, if we want to create a derived class **ChildClass** that inherits from a base class **ParentClass**, we can declare it like this:

```
class ChildClass : public ParentClass {

    // class members
```

};

Here, the **public** keyword specifies the access level of the inherited members of the parent class in the derived class.

The access level of members in C++ classes can be declared using the following keywords:

1. **public**: Members declared as **public** can be accessed by any part of the program.
2. **private**: Members declared as **private** are only accessible within the class.
3. **protected**: Members declared as **protected** are accessible within the class and any derived classes, but not accessible outside the class hierarchy.

By default, the access level of members in a C++ class is **private**.

Here's an example of a class hierarchy that demonstrates inheritance and access levels:

```
class Animal {
  protected:
        int age;

  public:
        void setAge(int a) {
        age = a;
        }
};

class Dog : public Animal {
  public:
        void displayAge() {
        cout << "The age of the dog is: " << age << endl;
        }
```

```
};


int main() {
        Dog myDog;
        myDog.setAge(3);
   myDog.displayAge();


        return 0;
}
```

In this example, we have a base class **Animal** that has a protected member **age** and a public member function **setAge()**. We then create a derived class **Dog** that inherits from **Animal** and has a public member function **displayAge()** that displays the value of **age**.

In the **main()** function, we create an object **myDog** of class **Dog**, set its age to 3 using the **setAge()** function inherited from **Animal**, and then display its age using the **displayAge()** function.

The output of the program will be:

The age of the dog is: 3

Overall, inheritance is a powerful concept in object-oriented programming that enables developers to create complex class hierarchies and increase code reuse. The access levels of members in a class hierarchy can be controlled using the **public**, **private**, and **protected** keywords.

Section-B

Attempt any eight questions.

4. Explain the syntax and rules of multiple inheritance in C++ with example.

Ans. Multiple inheritance is a feature of object-oriented programming languages, including C++, that allows a derived class to inherit from multiple base classes. This means that a class can have multiple immediate parent classes, each contributing its own set of members and functions to the derived class.

The syntax for multiple inheritance in C++ is as follows:

```
class DerivedClass : accessSpecifier BaseClass1, accessSpecifier BaseClass2, ...,
accessSpecifier BaseClassN {

        // class members

};
```

Here, **DerivedClass** is the derived class that inherits from multiple base classes **BaseClass1** through **BaseClassN**. Each base class is separated by a comma and must be preceded by an access specifier **accessSpecifier** which specifies the access level of the inherited members of the base class in the derived class.

The access specifiers for multiple inheritance in C++ are:

- **public**: The public members of the base class are accessible to everyone through the derived class.
- **protected**: The protected members of the base class are accessible only to the derived class and its subclasses.
- **private**: The private members of the base class are not accessible directly by the derived class.

Here's an example of multiple inheritance in C++:

```
#include <iostream>

using namespace std;



class A {

  protected:

        int a;



  public:

        A() {

        a = 10;

        }

};
```

```
class B {
  protected:
        int b;

  public:
        B() {
        b = 20;
        }
};

class C : public A, public B {
  public:
        void display() {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
        }
};

int main() {
        C obj;
        obj.display();

        return 0;
}
```

In this example, we have three classes: **A**, **B**, and **C**. Classes **A** and **B** are base classes and each has a protected member **a** and **b**, respectively. Class **C** is the derived class that inherits from **A** and **B**.

The **C** class has a public member function **display()** that displays the values of **a** and **b** inherited from **A** and **B**, respectively.

In the **main()** function, we create an object **obj** of class **C** and call the **display()** function, which displays the values of **a** and **b** inherited from **A** and **B**, respectively.

The output of the program will be:

a = 10

b = 20

In this example, we used the **public** access specifier to specify that the **a** and **b** members of **A** and **B** are publicly inherited by **C**. This means that the members can be accessed directly by the **C** class and its objects.

Overall, multiple inheritance in C++ can be a useful tool for creating complex class hierarchies that take advantage of code reuse by inheriting from multiple base classes. It is important to use access specifiers carefully to control the visibility of inherited members and avoid potential conflicts or ambiguities.

5. Explain do/while structure with example.

Ans. The **do/while** loop structure is a control structure in programming that allows a certain block of code to be executed repeatedly until a specified condition is met. The difference between a **do/while** loop and a **while** loop is that in a **do/while** loop, the block of code is executed at least once before the condition is checked.

The syntax of the **do/while** loop in C++ is as follows:

```
do {
        // code to be executed
} while(condition);
```

Here, the **do** keyword indicates the beginning of the loop block, which contains the code to be executed. After the code block is executed, the condition is checked using the **while** keyword. If the condition is true, the loop block is executed again; otherwise, the loop terminates.

Here's an example of using a **do/while** loop in C++:

```
#include <iostream>

using namespace std;


int main() {

	int i = 1;

	do {

	cout << i << endl;

	i++;

	} while(i <= 5);


	return 0;

}
```

In this example, we declare an integer variable **i** and initialize it to **1**. We then use a **do/while** loop to print the values of **i** from **1** to **5**. The loop block contains the **cout** statement to print the current value of **i**, followed by the **i++** statement to increment **i**.

Since the condition **i <= 5** is true for the first iteration, the loop block is executed and the value of **i** is printed to the console. The **i++** statement then increments the value of **i** to **2**. The loop continues to execute until the condition becomes false when **i** reaches **6**.

The output of the program will be:

```
1

2

3

4

5
```

In this example, the **do/while** loop structure ensures that the loop block is executed at least once, regardless of the initial value of **i**. This can be useful in situations where you need to ensure that a certain block of code is executed at least once, regardless of the input or other conditions.

6. Explain the Inline function with example.

Ans. An inline function is a function that is expanded in-line when it is called, rather than executing a function call. This means that the code for the inline function is inserted directly into the calling code, avoiding the overhead of a function call.

In C++, we can declare a function as inline by using the **inline** keyword before the function definition. Here's an example:

```
#include <iostream>

using namespace std;



inline int add(int a, int b) {

	return a + b;

}



int main() {

	int x = 5, y = 7;

	int z = add(x, y);

	cout << "z = " << z << endl;

	return 0;

}
```

In this example, we declare an inline function **add** that takes two integer parameters and returns their sum. We then call this function inside the **main** function and assign the result to an integer variable **z**. Finally, we print the value of **z** to the console.

Since the **add** function is declared as inline, the code for the function is inserted directly into the calling code when the function is called. This means that there is no function call overhead, and the code runs faster.

Note that the compiler may choose not to inline a function, even if it is declared as inline. In such cases, the **inline** keyword is treated as a suggestion to the compiler to inline the function, but the final decision is made by the compiler.

Inline functions are often used for small, frequently-used functions such as arithmetic operations, where the overhead of a function call can add up. However, it is important to note that overuse of inline functions can increase code size and reduce code maintainability, so they should be used judiciously.

7. What is multiple inheritance? Explain with example.

Ans. Multiple inheritance is a feature of object-oriented programming languages, including C++, that allows a class to inherit from more than one base class. This means that a derived class can have multiple parent classes, each of which contributes its own set of member variables and member functions to the derived class.

Here's an example of multiple inheritance in C++:

```
#include <iostream>

using namespace std;


class A {

public:

	void foo() {

	cout << "A::foo()" << endl;

	}

};


class B {

public:

	void bar() {
```

```
        cout << "B::bar()" << endl;
        }
};

class C : public A, public B {
public:
        void baz() {
        cout << "C::baz()" << endl;
        }
};

int main() {
        C c;
        c.foo();
        c.bar();
        c.baz();
        return 0;
}
```

In this example, we define three classes **A**, **B**, and **C**. Classes **A** and **B** each have a single member function **foo** and **bar**, respectively. Class **C** inherits from both **A** and **B** using the syntax **class C : public A, public B**, meaning that it is derived from both **A** and **B**.

The **C** class also has its own member function **baz**.

In the **main** function, we create an instance of the **C** class and call each of its member functions **foo**, **bar**, and **baz**. Since **C** inherits from both **A** and **B**, it can call the member functions of both classes, even though they are defined in different base classes.

The output of the program will be:

A::foo()

B::bar()

C::baz()

This example shows how multiple inheritance allows a derived class to inherit from multiple base classes, each of which contributes its own set of member functions and member variables to the derived class. Multiple inheritance can be useful in situations where a derived class needs to combine the functionality of multiple base classes. However, it can also lead to complex class hierarchies and code that is harder to understand and maintain, so it should be used judiciously.

8. Explain the static class members with examples.

Ans. In C++, a static class member is a member of a class that is shared among all instances of that class. This means that there is only one copy of the member, regardless of how many objects of the class are created.

Static class members are declared using the **static** keyword before the member declaration. Here's an example:

```
#include <iostream>

using namespace std;


class MyClass {

public:

        static int count;

        int id;


        MyClass() {

        id = ++count;

        }

};
```

```
int MyClass::count = 0;


int main() {
	MyClass obj1, obj2, obj3;
	cout << "obj1.id = " << obj1.id << endl;
	cout << "obj2.id = " << obj2.id << endl;
	cout << "obj3.id = " << obj3.id << endl;
	cout << "MyClass::count = " << MyClass::count << endl;
	return 0;
}
```

In this example, we define a class **MyClass** that has a static member **count** and a non-static member **id**. The **id** member is used to assign a unique ID to each object of the class, and the **count** member is used to keep track of how many objects have been created.

The **count** member is declared as static using the **static** keyword before the member declaration. The **count** member is also initialized to 0 outside the class definition.

In the **main** function, we create three objects of the **MyClass** class and print their **id** member variables to the console, as well as the value of the static **count** member. Since the **count** member is shared among all instances of the class, its value is incremented each time a new object is created, and its value is printed as 3 at the end of the program.

Static class members are useful for maintaining data that is shared among all instances of a class. They can also be used to implement class-level functionality, such as static member functions that operate on static data members. However, it is important to note that static data members can also introduce potential issues such as race conditions, so they should be used with care.

Top of Form

9. Differentiate between macro and function.

Ans. In programming, macros and functions are both tools used to accomplish tasks and solve problems. However, there are some key differences between the two:

1. Definition: A macro is a preprocessor directive that is processed before compilation, while a function is a code block that is executed during runtime.
2. Expansion: A macro is expanded inline by the preprocessor, meaning that the code of the macro is directly substituted into the source code at the point where the macro is used. On the other hand, a function is called by its name and executes its code block, returning a value if necessary.
3. Arguments: A macro can take arguments just like a function, but the arguments are not type-checked and can be any valid C++ expression. A function's arguments must be declared with specific types.
4. Return value: A macro does not return a value, as it is not a code block that can execute and return something. A function, however, can be designed to return a value.
5. Syntax: A macro is defined using the **#define** preprocessor directive and is usually written in all capital letters with underscores separating words. A function is defined using the **function return_type function_name(arguments)** syntax.

Here is an example of a macro and a function that accomplish the same task of computing the square of a number:

```
#define SQUARE(x) (x*x)



int square(int x) {

	return x * x;

}



int main() {

	int x = 5;

	int y = SQUARE(x);	// y = 25

	int z = square(x);	// z = 25

	return 0;

}
```

In this example, the macro **SQUARE** and the function **square** both compute the square of a number. The macro is defined using the **#define** preprocessor directive and takes a single argument **x**, which is squared using the **x*x** expression. The function is defined using the **int**

**square(int x)** syntax and returns the value of **x** squared using the **return x*x** statement. In the **main** function, we create a variable **x** and use it to compute the square using both the macro and the function.

10. Explain the use of break and continue statements in switch case statements in C++.
Ans. In C++, the switch case statement is used to execute different code blocks depending on the value of a given expression. The break and continue statements can be used within a switch case statement to control the flow of execution.

The break statement is used to terminate the execution of the switch case statement. When the break statement is encountered, the control is transferred to the statement following the switch case block. For example:

```
switch (x) {
        case 1:
        // Code block to execute if x is 1
        break;
        case 2:
        // Code block to execute if x is 2
        break;
        default:
        // Code block to execute if x is not 1 or 2
        break;
}
```

In the above example, if x is 1, the code block following the case 1 label is executed and then the break statement is encountered, which transfers the control to the statement following the switch case block.

The continue statement, on the other hand, is used to skip the remaining code in the current iteration of a loop and continue with the next iteration. When used within a switch case statement, the continue statement can be used to skip a particular case and move to the next case. For example:

```
switch (x) {
```

```
        case 1:

        // Code block to execute if x is 1

        break;

        case 2:

        // Code block to execute if x is 2

        continue;

        default:

        // Code block to execute if x is not 1 or 2

          break;
}
```

In the above example, if x is 2, the code block following the case 2 label is skipped due to the continue statement and the control is transferred to the next case label (in this case, the default label).

Overall, break and continue statements can be useful in controlling the flow of execution within a switch case statement and can help to write more efficient and concise code.

11. Explain the different storage classes in C++.

Ans.In C++, storage classes determine the storage duration, scope, and linkage of a variable or function. The following are the four storage classes available in C++:

1. Automatic storage class (default): Variables declared inside a function or block with no storage class specifier are automatically allocated memory from the stack at runtime. These variables have local scope and are destroyed when the function or block completes execution.
2. Static storage class: Variables declared with the static storage class specifier are allocated memory from the data segment of the program and retain their values between function calls. Static variables have local scope but are not destroyed when the function or block completes execution. Instead, they are destroyed when the program terminates.
3. Register storage class: Variables declared with the register storage class specifier are allocated memory from CPU registers instead of the stack. This can result in faster access times for frequently used variables. However, the compiler may ignore the register specifier and allocate memory from the stack if there are not enough registers available.

4. Extern storage class: Variables and functions declared with the extern storage class specifier have global scope and can be accessed from any function in the program. However, the memory for these variables is not allocated until they are defined in a separate file or scope. In other words, the extern specifier is used to declare a variable or function in one file or scope and define it in another file or scope.

Overall, the choice of storage class for a variable or function depends on its intended use and requirements for memory management, scope, and linkage.

12. Explain the multilevel inheritance. How is it different from multiple inheritance.

Ans. Multilevel inheritance and multiple inheritance are two different forms of inheritance in object-oriented programming.

Multilevel inheritance refers to a type of inheritance where a derived class is derived from another derived class. In other words, a class is derived from a class, which is derived from another class. This forms a hierarchical structure of classes, with each derived class inheriting the characteristics of its parent classes.

For example, consider the following class hierarchy:

```
class Animal {

public:

        void eat() {

        cout << "Eating...\n";

        }

};


class Mammal : public Animal {

public:

        void sleep() {

        cout << "Sleeping...\n";

        }
```

```
};


class Dog : public Mammal {

public:

        void bark() {

        cout << "Barking...\n";

        }

};
```

In this example, the Dog class is derived from the Mammal class, which is derived from the Animal class. As a result, the Dog class inherits the eat() method from the Animal class and the sleep() method from the Mammal class. It also has its own bark() method.

On the other hand, multiple inheritance refers to a type of inheritance where a derived class is derived from multiple base classes. In other words, a class can inherit the characteristics of more than one class. This can result in complex class hierarchies and can be challenging to manage.

For example, consider the following class hierarchy:

```
class Shape {

public:

        virtual void draw() = 0;

};


class Color {

public:

        virtual void fill() = 0;

};


class Rectangle : public Shape, public Color {
```

```
public:
        void draw() {
        cout << "Drawing Rectangle...\n";
        }
        void fill() {
        cout << "Filling Rectangle with Color...\n";
        }
};
```

In this example, the Rectangle class is derived from both the Shape and Color classes, which are unrelated to each other. As a result, the Rectangle class inherits the draw() method from the Shape class and the fill() method from the Color class.

Overall, the main difference between multilevel inheritance and multiple inheritance is that multilevel inheritance forms a hierarchical structure of classes, while multiple inheritance allows a class to inherit from multiple unrelated classes.

13. Explain the exceptional handling with example.

Ans. Exception handling is a mechanism used in programming languages like C++ to handle runtime errors or exceptional situations that occur during the execution of a program. Exception handling allows a program to recover from errors gracefully and continue execution, instead of abruptly terminating.

The basic concept of exception handling is to separate error-handling code from normal code by enclosing the code that might generate an error in a try block, and then catching and handling the error in a catch block.

Here is an example of how exception handling works in C++:

```
#include <iostream>

using namespace std;
```

```
int main() {
  int x = 10, y = 0, z;
  try {
        if (y == 0) {
        throw "Divide by zero error"; // throwing an exception
        }
        z = x / y;
        cout << "Result: " << z << endl;
  }
  catch (const char* msg) { // catching the exception
        cerr << "Error: " << msg << endl;
  }
  return 0;
}
```

In this example, we try to divide x by y, but since y is zero, a divide by zero error occurs. To handle this error, we enclose the code that might generate an error in a try block, and we catch the error in a catch block. In the catch block, we print an error message to the standard error output stream using the cerr object.

When the program is executed, the following output is produced:

```
Error: Divide by zero error
```

As we can see, the error message is printed to the console, and the program continues execution instead of crashing.

In summary, exception handling is an important feature of C++ that allows programmers to write robust and fault-tolerant code. By using exception handling, programmers can detect and handle errors at runtime, making their programs more reliable and easier to maintain.