1. **What is object-oriented programming? How does it differ from procedure-oriented programming? Explain features of object-oriented programming.**

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects and classes. It is a way of designing and organizing code that models real-world objects and systems. In OOP, a program is divided into objects that interact with each other to achieve a specific goal.

How does it differ from Procedure-Oriented Programming (POP)? Procedure-Oriented Programming (POP) is a programming paradigm that focuses on procedures or functions that perform specific tasks. In POP, a program is divided into functions that are executed in a specific order to achieve a goal.

The key differences between OOP and POP are:

* Focus: OOP focuses on objects and their interactions, while POP focuses on procedures and functions.
* Organization: OOP organizes code into objects and classes, while POP organizes code into functions and procedures.
* Modularity: OOP promotes modularity by encapsulating data and behavior within objects, while POP promotes modularity by breaking down code into smaller functions.

Features of Object-Oriented Programming in C++ Here are the key features of Object-Oriented Programming in C++:

* Classes and Objects:
  + A class is a blueprint or a template that defines the properties and behavior of an object.
  + An object is an instance of a class, which has its own set of attributes (data) and methods (functions).
* Encapsulation:
  + Encapsulation is the concept of bundling data and methods that operate on that data within a single unit (class).
  + It helps to hide the implementation details of an object from the outside world.
* Inheritance:
  + Inheritance is the mechanism by which one class can inherit the properties and behavior of another class.
  + It allows for code reuse and facilitates the creation of a hierarchy of classes.
* Polymorphism:
  + Polymorphism is the ability of an object to take on multiple forms.
  + It can be achieved through method overriding (where a subclass provides a different implementation of a method) or method overloading (where multiple methods with the same name can be defined with different parameters).
* Abstraction:
  + Abstraction is the concept of showing only the necessary information to the outside world while hiding the implementation details.
  + It helps to simplify complex systems by exposing only the essential features.
* Composition:
  + Composition is the concept of creating objects from other objects or collections of objects.
  + It allows for the creation of complex objects from simpler ones.

1. **Explain Memory Management Operators of C++ with examples.**

In C++, memory management is crucial for efficient program execution. C++ offers two primary operators for dynamic memory allocation and deallocation: new and delete.

**The new Operator:**

* Used to allocate memory dynamically on the heap (a region of memory outside the program's stack).
* Takes a data type as an argument and returns a pointer to the allocated memory.

**Example:**

|  |
| --- |
| C++  int\* ptr = new int; *// Allocates memory for an integer*  \*ptr = 45; *// Assigns value 45 to the allocated memory* |

**new for Arrays:**

* Can be used to allocate memory for arrays of any data type.
* Syntax: data\_type\* ptr = new data\_type[size];

|  |
| --- |
| **Example:**  C++  float\* marks = new float[5]; *// Allocates memory for an array of 5 floats*  for (int i = 0; i < 5; i++) {    marks[i] = i \* 10.0f; *// Assigning values to the array elements*  } |

**The delete Operator:**

* Used to deallocate memory previously allocated using new.
* Takes a pointer to the memory block as an argument and frees the memory.

**Important Note:**

* It's crucial to deallocate memory using delete when you're done with it to avoid memory leaks (unused memory allocation).
* For arrays, use delete[] ptr; to deallocate the entire array.

**Example:**

|  |
| --- |
| C++  delete ptr; *// Deallocates memory pointed to by ptr (assuming ptr was allocated using new int)*  delete[] marks; *// Deallocates memory for the entire float array* |

**Additional Points:**

* new can optionally initialize the allocated memory during allocation.
* C++ also supports smart pointers for automatic memory management, reducing the risk of memory leaks.

By effectively using new and delete, you can manage memory dynamically in your C++ programs.

1. **Explain Call by Value and Call by Reference with appropriate examples.**

In C++, functions can manipulate data passed to them from the calling program. There are two main ways this data can be passed: call by value and call by reference. They differ in how the function receives and interacts with the data.

**Call by Value**

* **Concept:** A copy of the actual argument's value is passed to the function's formal parameter.
* **Impact on Original Value:** Changes made to the parameter within the function do not affect the original variable in the calling program. They operate on separate copies.
* **Efficiency:** Less efficient for large data types (e.g., arrays) due to copying overhead.

**Example:**

#include <iostream>

using namespace std;

string swapName(string x, string y)

{

  string temp = x;

  x = y;

  y = temp;

}

int main()

{

  string fName = "MrX";

  string lName = "MrY";

  cout << "Before swap: " << "\n";

  cout << fName + " "<< lName << "\n";

  swapName(fName, lName);

  cout << "After swap: " << "\n";

  cout << fName + " "<< lName << "\n";

}

**Call by Reference**

* **Concept:** The address (memory location) of the actual argument is passed to the function's formal parameter. This creates a reference to the original variable.
* **Impact on Original Value:** Changes made to the parameter within the function directly modify the original variable because they refer to the same memory location.
* **Efficiency:** More efficient for large data types as only the reference (address) is copied.

**Example:**

#include <iostream>

using namespace std;

string swapName(string &x, string &y)

{

  string temp = x;

  x = y;

  y = temp;

}

int main()

{

  string fName = "MrX";

  string lName = "MrY";

  cout << "Before swap: " << "\n";

  cout << fName + " "<< lName << "\n";

  swapName(fName, lName);

  cout << "After swap: " << "\n";

  cout << fName + " "<< lName << "\n";

}

1. **Describe types of constructors with an example.**

There are several types of constructors in C++:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Move Constructor**

**1. Default Constructor:**

* A constructor with an empty parameter list (no arguments).
* If you don't explicitly define a constructor, the compiler provides a default one that initializes member variables to their default values (0 for integers, empty strings, etc.).
* You can also define your own default constructor to provide specific initialization

C++

class Point {

public:

    int x, y;

*// Default constructor (implicitly provided by the compiler if not defined explicitly)*

    Point() {

        x = 0;

        y = 0;

    }

};

int main() {

    Point p1; *// Calls the default constructor, p1.x and p1.y become 0*

    return 0;

}

**2. Parameterized Constructor:**

* Takes arguments to initialize the object's member variables during creation.
* Useful when you want to control the initial values of the object.
* You can have multiple parameterized constructors with different parameter lists (constructor overloading).

C++

class Rectangle {

public:

    int width, height;

*// Parameterized constructor*

    Rectangle(int w, int h) {

        width = w;

        height = h;

    }

};

int main() {

    Rectangle rect1(5, 3); *// Calls the parameterized constructor, rect1.width is 5 and rect1.height is 3*

    return 0;

}

**3. Copy Constructor:**

* Creates a new object as a copy of an existing object.
* Used when you want to pass objects by value (a copy is passed) or create a deep copy to avoid unintended modifications to the original object.
* The compiler provides a default copy constructor that performs a member-wise copy, but you can define your own for more control.

C++

class Student {

public:

    int id;

    std::string name;

*// Default copy constructor (implicitly provided by the compiler)*

    Student(const Student& other) {

        id = other.id;

        name = other.name; *// Performs a shallow copy (copies the pointer to the string)*

    }

};

int main() {

    Student s1(123, "Alice");

    Student s2(s1); *// Calls the copy constructor, s2 is a copy of s1 (shallow copy by default)*

    return 0;

}

**4. Move Constructor (C++11 and later):**

* Introduced in C++11 to efficiently transfer ownership of resources (like dynamically allocated memory) from one object to another.
* Takes an rvalue reference (represents a value that will be moved from) as an argument.
* Moves the resources from the source object to the target object, leaving the source object in a valid but empty state (often referred to as "moved-from" state).
* Useful for preventing unnecessary copying of large objects and improving performance.

C++

#include <memory>

class MyClass {

public:

    std::unique\_ptr<int> data;

*// Move constructor*

    MyClass(MyClass&& other) noexcept {

        data = std::move(other.data); *// Transfers ownership of the unique\_ptr from other*

    }

*// Destructor (to release the memory if necessary)*

    ~MyClass() {

        delete data;

    }

};

int main() {

    MyClass obj1(std::make\_unique<int>(42));

    MyClass obj2(std::move(obj1)); *// Calls the move constructor, obj2 takes ownership of the data*

    return 0;

}

1. **Explain the use of objects as function arguments with examples.**

In C++, you can pass objects of a class to functions as arguments, just like you pass primitive data types (like integers or strings). This allows functions to operate on the data stored within the object and potentially modify it.

**How It Works**

When you pass an object as an argument, by default, a copy of the object is created and passed to the function. This means any changes made to the object within the function will not affect the original object.

**Types of Passing Objects to Functions**

There are three primary ways to pass objects to functions in C++:

1. **Pass-by-Value**
2. **Pass-by-Reference**
3. **Pass-by-Const-Reference**

#include <iostream>

#include <string>

class Rectangle {

public:

    double width;

    double height;

    Rectangle(double w, double h) : width(w), height(h) {}

};

double calculateArea(const Rectangle& rect) {

    return rect.width \* rect.height;

}

int main() {

    Rectangle myRect(5.0, 3.0);

    double area = calculateArea(myRect);

    std::cout << "The area of the rectangle is: " << area << std::endl;

    return 0;

}

1. **Can we use object-oriented concepts using structure instead of classes? Justify your opinion.**

Yes, we can use object-oriented concepts with structures in C++ instead of classes. Here's a justification for this opinion:

1. Structures in C++ are very similar to classes:
   * Both can have member functions and data members
   * Both support access specifiers (public, private, protected)
   * Both can have constructors and destructors
   * Both can be inherited from and can inherit other structures or classes
2. The main difference between structures and classes in C++ is the default access specifier:
   * For structures, members are public by default
   * For classes, members are private by default

This means we can implement object-oriented principles using structures:

1. Encapsulation: We can use access specifiers to control access to data members and methods.
2. Inheritance: Structures can inherit from other structures or classes.
3. Polymorphism: We can use virtual functions in structures.
4. Abstraction: We can create abstract structures with pure virtual functions.

Here's a brief example demonstrating these concepts using a structure:

struct Shape {

    virtual double area() const = 0; *// Pure virtual function*

    virtual ~Shape() {} *// Virtual destructor*

};

struct Circle : Shape {

private:

    double radius;

public:

    Circle(double r) : radius(r) {}

    double area() const override {

        return 3.14159 \* radius \* radius;

    }

};

struct Rectangle : Shape {

private:

    double width, height;

public:

    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {

        return width \* height;

    }

};

1. **Define namespace. List any two manipulators with their uses.**

**Namespaces in C++**

In C++, a namespace is a declarative region that provides a scope to prevent naming conflicts. It acts like a container that groups together identifiers (names) like functions, variables, classes, etc. This helps organize code, especially when working with multiple libraries or large projects.

a) endl: Use: Inserts a newline character and flushes the output buffer. Example: cout << "Hello" << endl;

b) setw: Use: Sets the width of the next input/output field. Example: cout << setw(10) << 42;

1. **In which case default argument is used? Describe with an example.**

Default arguments in C++ are used when you want to provide a predefined value for a function parameter, allowing the function to be called with fewer arguments than it is defined to accept. This is particularly useful when a function has common or typical values for some parameters.

**Here's a concise explanation with an example:**

Case for using default arguments: When a function parameter often has a common value, but you want to retain the flexibility to specify a different value when needed.

#include <iostream>

using namespace std;

void printMessage(string message, int repeatCount = 1) {

    for (int i = 0; i < repeatCount; i++) {

        cout << message << endl;

    }

}

int main() {

*// Using default argument*

    printMessage("Hello"); *// Prints once*

*// Overriding default argument*

    printMessage("Hi", 3); *// Prints 3 times*

    return 0;

}

1. **Write a program to demonstrate returning objects from functions in C++.**

#include <iostream>

class Point {

public:

    int x, y;

    Point(int x = 0, int y = 0) : x(x), y(y) {}

    void display() {

        std::cout << "(" << x << ", " << y << ")" << std::endl;

    }

};

Point addPoints(Point p1, Point p2) {

    Point result(p1.x + p2.x, p1.y + p2.y);

    return result;

}

int main() {

    Point p1(2, 3);

    Point p2(4, 5);

    std::cout << "Point 1: ";

    p1.display();

    std::cout << "Point 2: ";

    p2.display();

    Point p3 = addPoints(p1, p2);

    std::cout << "Result: ";

    p3.display();

    return 0;

}

1. **Illustrate the use of the static variable with a simple program.**

#include <iostream>

class Counter {

public:

    static int count; *// Declare count as a static variable*

    Counter() {

        count++; *// Increment count for each object creation*

    }

    void displayCount() {

        std::cout << "Object count: " << count << std::endl;

    }

};

*// Initialize static variable outside the class (required)*

int Counter::count = 0; *// Set initial value to 0*

int main() {

    Counter c1;

    c1.displayCount(); *// Output: Object count: 1*

    Counter c2;

    c2.displayCount(); *// Output: Object count: 2*

    return 0;

}

**11. Write a program to create a class named Quadratic representing a function of form f(x) = ax² + bx + c, where x is a real variable and a, b, c are real constants. The class must satisfy the following requirements. a. A constructor that takes the values of a, b, and c as arguments should be provided. All three of these arguments should default to zero. b. A function that takes a single argument x returns the value of f(x). c. All data members should be private.**

#include <iostream>

class Quadratic {

private:

    double a;

    double b;

    double c;

public:

*// Constructor with default arguments*

    Quadratic(double a = 0.0, double b = 0.0, double c = 0.0) : a(a), b(b), c(c) {}

*// Function to calculate f(x)*

    double evaluate(double x) const {

        return a \* x \* x + b \* x + c;

    }

};

int main() {

*// Create a Quadratic object with default coefficients*

    Quadratic q1;

*// Create another Quadratic object with specific coefficients*

    Quadratic q2(2.0, -3.0, 1.0);

*// Test evaluate function for different values of x*

    double x1 = 2.0;

    double result1 = q1.evaluate(x1);

    double x2 = 1.0;

    double result2 = q2.evaluate(x2);

    std::cout << "f(x) for q1 at x = " << x1 << " is: " << result1 << std::endl;

    std::cout << "f(x) for q2 at x = " << x2 << " is: " << result2 << std::endl;

    return 0;

}