Assignment 4

1. What is Polymorphism in C++ and Why Is It Important?

Polymorphism in C++ refers to the ability of a function or object to behave in multiple forms. It allows the same function or method to work with different types of data. Polymorphism is important because it promotes flexibility and maintainability in code, allowing programmers to write more generic and reusable code. Inheritance and virtual functions in C++ enable polymorphism.

2. Compile-time (Static) Polymorphism with Examples

Compile-time polymorphism, also known as **static polymorphism**, is determined at compile time and occurs when the function or operator is resolved during compilation. This is achieved through **function overloading** and **operator overloading**.

Example of Function Overloading:

cpp
CopyEdit

```
class Printer {
public:
    void print(int i) {
        cout << "Printing integer: " << i << endl;
    }
    void print(double d) {
        cout << "Printing double: " << d << endl;
    }
};

int main() {
    Printer p;
    p.print(10);   // Calls print(int)
    p.print(10.5); // Calls print(double)
    return 0;
}
```

Example of Operator Overloading:

cpp
CopyEdit

```
class Complex {
public:
    int real, imag;
    Complex operator + (const Complex &c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
```

```
        return temp;
    }
};

int main() {
    Complex c1, c2, c3;
    c1.real = 10; c1.imag = 5;
    c2.real = 5; c2.imag = 3;
    c3 = c1 + c2; // Uses overloaded + operator
    cout << c3.real << " + " << c3.imag << "i" << endl;
    return 0;
}
```

3. Runtime (Dynamic) Polymorphism with Examples

Runtime polymorphism (also called **dynamic polymorphism**) occurs when the function to be executed is determined at runtime. It is achieved using **virtual functions** and **function overriding**.

Example:

cpp
CopyEdit

```
class Base {
public:
    virtual void display() {
        cout << "Base class display" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived class display" << endl;
    }
};

int main() {
    Base *bptr;
    Derived d;
    bptr = &d;
    bptr->display();  // Calls Derived's display function
(runtime polymorphism)
    return 0;
}
```

In this example, the `display()` function is overridden in the `Derived` class, and which function is called is determined at runtime.

4. Difference Between Static and Dynamic Polymorphism

- **Static Polymorphism**: Resolved at compile time. Achieved through function overloading and operator overloading. Example: method selection based on the argument types.
- **Dynamic Polymorphism**: Resolved at runtime. Achieved through inheritance and virtual functions. Example: function selection based on the object type (using base class pointers or references).

5. How Is Polymorphism Implemented in C++?

Polymorphism in C++ is implemented using:

- **Function overloading** (for compile-time polymorphism).
- **Operator overloading** (for compile-time polymorphism).
- **Virtual functions** (for runtime polymorphism).
- **Function overriding** (for runtime polymorphism).

6. What Are Pointers in C++ and How Do They Work?

A **pointer** in C++ is a variable that stores the memory address of another variable. Pointers allow for dynamic memory allocation, array handling, and object manipulation.

Example:

```
cpp
CopyEdit
int x = 10;
int* ptr = &x;  // ptr stores the address of x
```

Pointers work by using the **address-of operator (`&`)** to get the memory address of a variable, and the **dereferencing operator (`*`)** to access the value at that memory address.

7. Syntax for Declaring and Initializing Pointers

To declare and initialize pointers in C++, use the following syntax:

```
cpp
CopyEdit
int* ptr;      // Declaration
ptr = &var;    // Initialization (where var is a variable)
```

8. How Do You Access the Value Pointed to by a Pointer?

You can access the value of a variable using the dereferencing operator (`*`):

cpp
CopyEdit
```
int x = 10;
int* ptr = &x;
cout << *ptr;  // Output: 10
```

9. Concept of Pointer Arithmetic

Pointer arithmetic allows you to perform operations on pointers like incrementing or decrementing them, which moves the pointer across memory locations.

cpp
CopyEdit
```
int arr[] = {10, 20, 30};
int* ptr = arr;
cout << *ptr << endl;  // Output: 10
ptr++;                 // Move to the next element
cout << *ptr << endl;  // Output: 20
```

10. Common Pitfalls When Using Pointers

- **Dangling pointers**: Pointers that reference freed memory.
- **Memory leaks**: Not releasing memory allocated using `new`.
- **Dereferencing null pointers**: Accessing data through a pointer that hasn't been initialized.

11. Pointers with Objects in C++

Pointers to objects allow dynamic allocation and manipulation of objects.

cpp
CopyEdit
```
class MyClass {
public:
    int x;
    MyClass(int val) : x(val) {}
};

int main() {
    MyClass* ptr = new MyClass(10);  // Dynamically allocated
    cout << ptr->x << endl;          // Access using pointer
    delete ptr;                      // Free memory
    return 0;
}
```

12. Dynamically Allocating Objects Using Pointers

To dynamically allocate an object, use the `new` operator:

```
cpp
CopyEdit
MyClass* ptr = new MyClass(5);  // Dynamically allocating
memory for MyClass
delete ptr;                     // Freeing the allocated
memory
```

13. Example of Accessing Object Members Using Pointers

You can use the `->` operator to access object members via pointers:

```
cpp
CopyEdit
class MyClass {
public:
    void show() { cout << "Hello!" << endl; }
};

int main() {
    MyClass* ptr = new MyClass();
    ptr->show();  // Accessing the method using pointer
    delete ptr;
    return 0;
}
```

14. Difference Between a Pointer to an Object and a Reference to an Object

- **Pointer to an object**: Stores the address of the object and can be reassigned to point to different objects. Can be null.
- **Reference to an object**: An alias to the object, which cannot be reassigned to refer to a different object. Cannot be null.

15. How Do You Release Dynamically Allocated Objects in C++?

Use the `delete` operator to free the memory allocated for objects created with `new`:

```
cpp
CopyEdit
MyClass* ptr = new MyClass();
delete ptr;  // Freeing memory
```

16. What Is the `this` Pointer in C++ and What Is Its Significance?

The `this` pointer is a hidden pointer in C++ that points to the current object of the class. It is used within non-static member functions to refer to the current instance.

17. How Is the `this` Pointer Used in Member Functions?

The `this` pointer allows access to the current object within a member function:

cpp
CopyEdit
```
class MyClass {
public:
    void print() {
        cout << "Object address: " << this << endl;
    }
};
```

18. Using the `this` Pointer to Return the Current Object

You can return the current object from a member function using the `this` pointer:

cpp
CopyEdit
```
class MyClass {
public:
    MyClass* getObject() {
        return this;  // Returns the current object
    }
};
```

19. What Is a Virtual Function in C++ and Why Is It Used?

A **virtual function** is a function that is declared in the base class and overridden in a derived class. It is used to achieve **runtime polymorphism**, allowing the correct function to be called based on the type of the object pointed to by a base class pointer.

20. Syntax for Declaring a Virtual Function

cpp
CopyEdit
```
class Base {
public:
    virtual void display() {
        cout << "Base class display" << endl;
    }
};
```

21. Vtable (Virtual Table) and Its Role in Virtual Functions

A **vtable** is a table of function pointers used to implement virtual functions. Each class with virtual functions has a vtable, and when a virtual function is called, the appropriate function from the vtable is invoked.

22. Pure Virtual Function and Its Declaration

A **pure virtual function** is a virtual function that has no implementation in the base class and must be implemented in the derived class. It is declared by assigning `0` to the function:

```
cpp
CopyEdit
class Base {
public:
    virtual void display() = 0;  // Pure virtual function
};
```

23. Example of a Class with Pure Virtual Functions

```
cpp
CopyEdit
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};
```

24. Implications of Having Pure Virtual Functions

- The class containing pure virtual functions becomes **abstract**, meaning it cannot be instantiated directly.
- Derived classes must provide implementations for all pure virtual functions to be instantiated.

25. Polymorphism Using Inheritance and Virtual Functions

Polymorphism is achieved by declaring virtual functions in the base class and overriding them in derived classes. The function called depends on the type of the object, not the pointer type.

26. Example of Polymorphism with Base and Derived Classes

cpp
CopyEdit

```
class Base {
public:
    virtual void display() {
        cout << "Base class display" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived class display" << endl;
    }
};

int main() {
    Base* bptr = new Derived();
    bptr->display();  // Output: Derived class display
    delete bptr;
    return 0;
}
```

27. Late Binding in Polymorphism

Late binding (or **dynamic dispatch**) is the process where the method to be called is determined at runtime, not compile time. It happens when a method is virtual, allowing dynamic polymorphism.

28. Compiler's Management of Polymorphism

The **compiler** manages polymorphism using a **vtable** and the **this** pointer for virtual functions. The correct method is chosen during runtime based on the actual object type.

29. What Is an Abstract Class in C++?

An **abstract class** is a class that cannot be instantiated. It contains at least one pure virtual function, forcing derived classes to implement that function.

30. Difference Between Abstract and Regular Classes

An **abstract class** contains one or more pure virtual functions and cannot be instantiated, while a regular class may or may not have pure virtual functions and can be instantiated.

31. Role of Abstract Methods in Abstract Classes

Abstract methods (pure virtual functions) define a contract that must be fulfilled by derived classes. They enforce that derived classes implement specific behaviors.

32. Defining and Using an Abstract Class

cpp
CopyEdit

```
class Abstract {
public:
    virtual void func() = 0;  // Pure virtual function
};

class Concrete : public Abstract {
public:
    void func() override {
        cout << "Implemented function" << endl;
    }
};
```

33. Benefits of Using Abstract Classes

- **Separation of interface and implementation**: Allows defining common functionality without implementation details.
- **Code reusability**: Ensures derived classes implement required methods while providing shared functionality.

34. Exception Handling in C++ and Why It's Important

Exception handling in C++ helps handle runtime errors. It provides a way to transfer control from one part of the program to another when an error occurs, ensuring that errors do not crash the program.

35. Syntax for Throwing and Catching Exceptions

cpp
CopyEdit

```
try {
    throw 10;  // Throw an exception
} catch (int e) {
    cout << "Caught exception: " << e << endl;
}
```

36. Concept of Try, Catch, and Throw Blocks

- **`try` block**: Contains code that may throw an exception.
- **`throw` statement**: Used to throw an exception.
- **`catch` block**: Catches and handles the thrown exception.

37. Role of the Catch Block

The **catch block** handles exceptions thrown by the try block, preventing the program from crashing.

38. Example of Handling Multiple Exceptions in C++

cpp
CopyEdit

```
try {
    throw 10;  // Throwing integer exception
} catch (int e) {
    cout << "Caught integer: " << e << endl;
} catch (const char* msg) {
    cout << "Caught string: " << msg << endl;
}
```

39. How Does the Throw Keyword Work in Exception Handling?

The **throw** keyword is used to raise an exception and transfer control to the catch block.

40. Purpose of the Finally Block in Exception Handling

C++ does not have a **finally** block, but it can be mimicked using destructors or by using `try-catch` blocks where cleanup occurs in the catch block.

41. Creating Custom Exception Classes

Custom exceptions can be created by deriving from `std::exception` or any other exception class.

cpp
CopyEdit

```
class MyException : public std::exception {
public:
    const char* what() const throw() {
```

```
        return "My custom exception";
    }
};
```

42. Templates in C++ and Their Usefulness

Templates allow writing generic code that works with any data type. They enable **code reusability** and avoid repetition.

43. Syntax for Defining a Function Template

cpp
CopyEdit

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

44. Example of a Function Template

cpp
CopyEdit

```
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(5, 10) << endl;   // Output: 15
    cout << add(5.5, 10.5) << endl; // Output: 16
    return 0;
}
```

45. What Is a Class Template and How Is It Different from a Function Template?

A **class template** defines a blueprint for classes, allowing you to create a class for any data type.

Difference: A **function template** is for creating functions that can work with any data type, while a **class template** is for creating classes.

46. Syntax for Defining a Class Template

cpp
CopyEdit

```
template <typename T>
class Box {
private:
    T value;
public:
    void set(T v) {
        value = v;
    }
    T get() {
        return value;
    }
};
```

47. Example of Class Template for Generic Data Structure

```
cpp
CopyEdit
template <typename T>
class MyStack {
private:
    T arr[100];
    int top;
public:
    MyStack() : top(-1) {}
    void push(T val) { arr[++top] = val; }
    T pop() { return arr[top--]; }
};
```

48. Instantiating a Template Class in C++

```
cpp
CopyEdit
MyStack<int> stack1;  // Creating a stack for integers
MyStack<double> stack2;  // Creating a stack for doubles
```

49. Advantages of Using Templates over Traditional Class Inheritance

Templates allow you to write more **generic**, **reusable** code without needing to rely on inheritance, which may introduce unnecessary complexity.

50. How Do Templates Promote Code Reusability in C++?

Templates allow you to write functions and classes that work with any data type, making your code reusable without duplication for each data type.