

BTCS202M: Assignment 1 (Theory & Practical)

1. Difference Between Procedural and OOP (with Example)

Procedural programming uses functions to perform tasks in sequence, while OOP focuses on objects that contain both data and behavior. OOP better models real-world systems, making code reusable and manageable. For example, OOP would represent a "Car" as a class with properties and methods, unlike procedural code which separates them.

2. Define OOP and Core Characteristics

Object-Oriented Programming (OOP) is a paradigm where software is structured around objects. Each object represents an entity with data (attributes) and operations (methods). The four key features of OOP are encapsulation, abstraction, inheritance, and polymorphism. These features allow for organized, reusable, and easy-to-maintain code structures.

3. Abstraction in OOP

Abstraction means showing only essential details and hiding the internal complexities. It lets programmers focus on what an object does rather than how it works. For example, a Car class may offer a start() method without showing the engine's internal mechanism, keeping the interface clean and easy to use.

4. Benefits of OOP Over Procedural Programming

OOP offers better organization, easier maintenance, and real-world modeling. It promotes reusability through inheritance and flexibility through polymorphism. Procedural code can become complex and repetitive as a project grows, while OOP keeps code modular, reducing duplication and improving clarity, especially in large applications.

5. Real-World Problem Using OOP

A school management system is a good example. Students, Teachers, and Courses can be modeled as classes. Each class holds specific data and behavior, making the code logical and manageable. OOP fits well here because it allows reuse, easy expansion, and real-world structure representation through inheritance and abstraction.

6. Four Key Principles of OOP

- **Encapsulation:** Bundling data and related methods.
- **Inheritance:** Deriving new classes from existing ones.
- **Polymorphism:** Performing tasks in multiple ways using a single interface.

- **Abstraction: Hiding unnecessary details from the user.**
- These principles help in building flexible, scalable, and robust applications.
-

7. Encapsulation and Code Example

Encapsulation hides object data using private access and allows controlled access using public methods. It protects sensitive data from unintended changes. For example, a Student class can store age privately and use `setAge()` and `getAge()` to control how it's accessed or modified, maintaining data integrity.

8. Inheritance and Code Example

Inheritance allows a new class to reuse features from an existing class. A Dog class can inherit from an Animal class, gaining access to common behaviors like `eat()`. This reduces code duplication and helps build hierarchical relationships, which are common in real-world systems.

9. Polymorphism (Overloading and Overriding)

Polymorphism lets you use the same function name in different ways. Function overloading allows multiple methods with the same name but different parameters. Function overriding allows a derived class to change the behavior of a method from the base class. This improves code flexibility and consistency.

10. Overloading vs Overriding

Overloading means having multiple functions with the same name but different arguments in the same class. Overriding means redefining a function in the derived class that already exists in the base class. Overloading occurs at compile time; overriding happens at runtime, supporting dynamic behavior.

11. Three Advantages of OOP

OOP improves code reuse through inheritance, offers better structure through classes and objects, and simplifies maintenance by modularizing code. These advantages make programs easier to write, understand, and modify. OOP is especially useful in large applications due to its organized and scalable approach to coding.

12. Application Domains for OOP

OOP is commonly used in GUI development, gaming, real-time systems, simulations, and web applications. Domains like mobile apps, CAD tools, and enterprise software benefit from OOP because it allows developers to model complex systems efficiently using object hierarchies and relationships.

13. OOP's Impact on Maintainability and Reusability

OOP boosts maintainability by organizing code into modular, self-contained classes. Reusability comes from inheritance, allowing developers to build new functionality on top of existing code. Changes are easier to manage, and shared code can be used across multiple projects without rewriting.

14. OOP in Large and Complex Systems

OOP supports large projects by breaking them into smaller, manageable pieces using objects and classes. Teams can work independently on different parts of a program. Code becomes easier to understand, test, and debug. It helps handle complexity through structure, abstraction, and reuse of components.

15. Benefits of OOP in Software Development

OOP allows better data management, easier debugging, and improved collaboration among developers. It enhances code clarity and allows building scalable systems with reusable parts. OOP mimics real-world objects, making it easier to design and maintain software, especially when requirements grow or change.

16. Basic Structure of a C++ Program

A basic C++ program includes headers, the `main()` function, and statements. `#include <iostream>` is used to access input/output. `using namespace std;` avoids writing `std::`. `int main()` is the starting point. The program ends with `return 0;`, signaling successful execution.

17. Purpose of Namespaces in C++

Namespaces prevent naming conflicts in large projects where multiple variables, functions, or classes might share the same name. By using namespace, you group identifiers logically and avoid clashes, especially when integrating multiple libraries or modules in a single program.

18. Identifiers and Rules

Identifiers are names for variables, functions, classes, etc. They must start with a letter or underscore, can't use special characters (except `_`), and can't be a reserved keyword. Names should be meaningful and consistent to improve readability and avoid errors.

19. Variables vs Constants

Variables can change values during execution, while constants remain fixed once defined. In C++, use `int x = 10;` for variables and `const int x = 10;` for constants. Constants help protect data from accidental modification, improving code safety and reliability.

20. Control Structures in C++ (with Code)

Control structures manage decision-making and loops. if-else is used for conditions, for and while for repetition. Example:

```
for (int i = 1; i <= 5; i++) {  
    cout << i << " ";  
}
```

This loop prints numbers 1 to 5. These structures guide the program's logical flow.

21. static_cast – Convert int to double and vice versa

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int intVal = 10;  
    double doubleVal = static_cast<double>(intVal); // int to double  
    cout << "Double value: " << doubleVal << endl;  
  
    double d = 9.87;  
    int i = static_cast<int>(d); // double to int  
    cout << "Int value: " << i << endl;  
  
    return 0;  
}
```

22. dynamic_cast – Safe downcasting in polymorphism

```
#include <iostream>
```

```
using namespace std;
```

```

class Base {
public:
    virtual void show() { cout << "Base class\n"; }
};

class Derived : public Base {
public:
    void show() override { cout << "Derived class\n"; }
};

int main() {
    Base* basePtr = new Derived(); // upcasting
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr); // safe downcasting

    if (derivedPtr)
        derivedPtr->show();
    else
        cout << "Casting failed\n";

    delete basePtr;
    return 0;
}

```

23. const_cast – Modify a constant variable

```

#include <iostream>

using namespace std;

int main() {
    const int a = 10;

    int* modifiable = const_cast<int*>(&a);

    *modifiable = 20; // dangerous, modifies read-only memory

```

```
    cout << "Modified value (undefined behavior): " << *modifiable << endl;

    return 0;
}
```

24. reinterpret_cast – Convert int to char

```
#include <iostream>

using namespace std;

int main() {
    int num = 65;

    char* ch = reinterpret_cast<char*>(&num);

    cout << "Character value: " << *ch << endl;

    return 0;
}
```

25. C-style Cast – Convert int to double

```
#include <iostream>

using namespace std;

int main() {
    int num = 42;

    double d = (double)num; // C-style cast

    cout << "Double value: " << d << endl;

    return 0;
}
```

Assignment 2 (Theory and Practical)

1. What is the purpose of the main function in a C++ program?

The main() function is the entry point of every C++ program. It is where the execution of the program begins.

2. Explain the significance of the return type of the main function.

The return type of `main()` is `int`, which indicates the program's exit status to the operating system. A return value of 0 typically means successful execution.

3. What are the two valid signatures of the main function in C++?

- `int main()`
- `int main(int argc, char* argv[])`

4. What is function prototyping and why is it necessary in C++?

Function prototyping is declaring a function before its use to inform the compiler about the function's name, return type, and parameters. It ensures type checking and helps avoid errors.

5. How do you declare a function prototype for a function that returns an integer and takes two integer parameters?

cpp

CopyEdit

```
int add(int a, int b);
```

6. What happens if a function is used before it is prototyped?

The compiler will throw an error or assume incorrect default return types, leading to undefined behavior.

7. What is the difference between a declaration and a definition of a function?

- Declaration tells the compiler about the function's existence.
- Definition provides the actual body or implementation of the function.

8. How do you call a simple function that takes no parameters and returns void?

cpp

CopyEdit

```
void myFunction();
```

```
...
```

```
myFunction();
```

9. Explain the concept of "scope" in the context of functions.

Scope defines the visibility and lifetime of variables or functions. Variables inside a function have local scope, while functions declared outside any class or function have global scope.

10. What is call by reference in C++?

Call by reference passes the actual address of the arguments, allowing the function to modify the original variables.

11. How does call by reference differ from call by value?

- Call by value passes a copy; changes do not affect the original variables.
 - Call by reference passes the reference; changes affect the original variables.
-

12. Provide an example of a function that uses call by reference to swap two integers.

```
void swap(int &a, int &b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

13. What is an inline function in C++?

An inline function is a function for which the compiler replaces the function call with the actual code of the function to reduce overhead.

14. How do inline functions improve performance?

They eliminate the overhead of function calls, especially for small, frequently called functions.

15. Explain the syntax for declaring an inline function.

cpp

CopyEdit

```
inline int square(int x) {  
    return x * x;  
}
```

16. What are macros in C++ and how are they different from inline functions?

Macros are preprocessor directives using #define to define code snippets. Unlike inline functions, they don't perform type checking and can lead to unexpected behaviors.

17. Explain the advantages and disadvantages of using macros over inline functions.

Advantages:

- Simple text substitution

Disadvantages:

- No type checking
- Difficult to debug
- Prone to errors

18. Provide an example to illustrate the differences between macros and inline functions.

cpp

CopyEdit

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

```
inline int square(int x) { return x * x; } // Inline function
```

19. What is function overloading in C++?

Function overloading allows multiple functions with the same name but different parameter lists in the same scope.

20. How does the compiler differentiate between overloaded functions?

The compiler uses the number and types of parameters (function signature) to differentiate.

21. Provide an example of overloaded functions in C++.

cpp

CopyEdit

```
int add(int a, int b) {  
    return a + b;  
}
```

```
float add(float a, float b) {  
    return a + b;  
}
```

22. What are default arguments in C++?

Default arguments allow you to assign default values to function parameters during declaration.

23. How do you specify default arguments in a function declaration?

cpp

CopyEdit

```
void display(int a, int b = 10);
```

24. What are the rules for using default arguments in functions?

- Default arguments must be specified from right to left.
- Once a default argument is provided, all arguments to its right must also have defaults.

25. Provide an example of a function with default arguments.

cpp

CopyEdit

```
void greet(string name = "User") {  
    cout << "Hello, " << name << "!";  
}
```

Assignment 3 (Theory and Practical)

1. What is an object in C++?

An object in C++ is an instance of a class. It represents a real-world entity with attributes (data members) and behaviors (member functions). Objects are created from classes and occupy memory during program execution.

2. What is a class in C++ and how does it differ from an object?

A class is a blueprint or template for creating objects. It defines the structure and behaviors that the objects created from it will have. While a class defines properties and methods, an object is an actual entity created using that class.

3. Explain the concept of encapsulation with an example.

Encapsulation is the bundling of data and methods that operate on that data within a single unit, i.e., a class. It restricts direct access to some of an object's components, which can prevent the accidental modification of data.

cpp

CopyEdit

```
class Account {  
private:  
    double balance;  
  
public:  
    void deposit(double amount) {  
        if (amount > 0) balance += amount;  
    }  
  
    double getBalance() const {  
        return balance;  
    }  
};
```

4. How do you define a class in C++?

cpp

CopyEdit

```
class ClassName {  
private:  
    // private members  
  
public:  
    // public members  
};
```

5. Describe the syntax for creating an object of a class.

```
ClassName objectName;
```

6. What are private members in a class and how are they accessed?

Private members are variables and functions that cannot be accessed directly outside the class. They are accessed and modified through public member functions.

7. What are public members in a class and how are they accessed?

Public members are variables and functions that can be accessed directly from outside the class.

8. Explain the significance of access specifiers in a class.

Access specifiers (private, public, protected) control the visibility and accessibility of class members. They help in implementing data hiding and encapsulation.

9. Provide an example of a class with both private and public members.

cpp

CopyEdit

```
class Person {  
private:  
    string name;
```

public:

```
void setName(const string& n) {  
    name = n;  
}
```

```
string getName() const {  
    return name;  
}
```

```
};
```

10. How does data hiding work in C++?

Data hiding is achieved by making data members private and providing public getter and setter functions to access and modify them. This prevents unauthorized access and modification.

11. What is a static data member in C++?

A static data member is shared by all objects of a class. It is initialized outside the class definition.

cpp

CopyEdit

```
class MyClass {  
    static int count;  
};
```

12. How do you declare and initialize a static data member?

cpp

CopyEdit

```
int MyClass::count = 0;
```

13. What is a static function member in C++?

A static function member belongs to the class rather than any object. It can only access static data members and other static functions.

cpp

CopyEdit

```
class MyClass {  
public:  
    static void display() {  
        // code  
    }  
};
```

14. How do static function members differ from regular function members?

Static function members can be called without creating an object and can only access static members of the class.

15. Provide an example of a class with static data and function members.

cpp

CopyEdit

```
class Counter {  
private:  
    static int count;  
  
public:  
    static void increment() {  
        count++;  
    }  
  
    static int getCount() {  
        return count;  
    }  
};  
  
int Counter::count = 0;
```

16. What is a constructor in C++ and why is it important?

A constructor is a special member function that initializes objects. It is called automatically when an object is created.

17. Explain the different types of constructors in C++.

- **Default constructor:** No parameters.
 - **Parameterized constructor:** Takes parameters to initialize objects with specific values.
 - **Copy constructor:** Initializes a new object as a copy of an existing object.
-

18. What is a default constructor and when is it used?

A default constructor is a constructor that takes no arguments. It is used to initialize objects with default values.

19. How do parameterized constructors work?

Parameterized constructors allow initializing objects with specific values at the time of creation.

cpp

CopyEdit

```
class Point {  
public:  
    int x, y;  
  
    Point(int x_val, int y_val) : x(x_val), y(y_val) {}  
};
```

20. What is a copy constructor and what is its purpose?

A copy constructor initializes a new object as a copy of an existing object. It is used when an object is passed by value or returned by value.

cpp

CopyEdit

```
class MyClass {  
public:  
    MyClass(const MyClass& other) {
```

```
        // copy data from other
    }
};
```

21. Explain the concept of constructor overloading.

Constructor overloading is defining multiple constructors with different parameter lists to initialize objects in different ways.

cpp

CopyEdit

```
class MyClass {
public:
    MyClass() { /* default */ }
    MyClass(int x) { /* initialize with x */ }
};
```

22. How does a constructor initializer list work?

A constructor initializer list initializes data members before the constructor body executes.

cpp

CopyEdit

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

23. What is a destructor in C++ and what is its purpose?

A destructor is a special member function that is called when an object is destroyed. It is used to release resources allocated by the object.

24. How is a destructor declared and defined?

cpp

CopyEdit

```
class MyClass {  
public:  
    ~MyClass() {  
        // cleanup code  
    }  
};
```

25. What happens if a destructor is not explicitly defined in a class?

If not explicitly defined, the compiler provides a default destructor that performs no action. However, if the class allocates resources dynamically, a custom destructor is needed to release them.

26. Explain the concept of automatic and dynamic storage duration in relation to destructors.

- Automatic storage duration: Objects are destroyed when they go out of scope.
 - Dynamic storage duration: Objects are destroyed when delete is called.
-

27. How do destructors differ from constructors?

Constructors initialize objects, while destructors clean up before objects are destroyed.

28. What is operator overloading in C++ and why is it useful?

Operator overloading allows defining custom behavior for operators. It makes code more intuitive and readable.

29. Describe the syntax for overloading an operator.

```
ReturnType operator op(Parameters) {  
    // code  
}
```

30. Which operators can and cannot be overloaded in C++?

ChatGPT said:

Operators That Can Be Overloaded in C++

Most operators in C++ can be overloaded. Here are some common ones:

Arithmetic Operators:

- `+, -, *, /, %`

Comparison Operators:

- `==, !=, <, >, <=, >=`

Assignment Operators:

- `=, +=, -=, *=, /=, %=`

Unary Operators:

- `++, --, !, ~`

Bitwise Operators:

- `&, |, ^, <<, >>`

Logical Operators:

- `&&, ||`

Other Operators:

- `[]` (subscript)
- `()` (function call)
- `->` (member access via pointer)
- `*` (dereference)
- `&` (address-of)
- `new, delete, new[], delete[]`

Operators That Cannot Be Overloaded in C++

Some operators are fundamental to the language and cannot be overloaded:

- `.` (member access)
 - `.*` (pointer-to-member access)
 - `::` (scope resolution)
 - `?:` (ternary conditional)
 - `sizeof` (size operator)
 - `typeid` (for RTTI)
 - `alignof` (alignment requirement)
 - `noexcept, static_cast, dynamic_cast, reinterpret_cast, const_cast`
-

31. Provide an example of overloading the "+" operator for a custom class.

Example:

```
#include <iostream>

using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

    // Overloading the '+' operator
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;
    c3.display();
    return 0;
}
```

Explanation:

- The operator+ function is overloaded to add two Complex objects.
- It returns a new Complex object with the sum of the real and imaginary parts.

32. Explain the concept of friend functions in the context of operator overloading.

Friend Functions:

- **Definition:** A friend function is a function that is not a member of a class but has access to its private and protected members.
 - **Usage in Operator Overloading:** Friend functions are often used to overload operators that need to access private members of a class. This is particularly useful for binary operators where symmetry is required (e.g., $a + b$ and $b + a$ should behave identically).
-

33. What is a friend function in C++ and how is it declared?

Definition:

- A friend function is a function that is not a member of a class but can access its private and protected members.

Declaration:

cpp

CopyEdit

```
class ClassName {  
    friend ReturnTyp FunctionName(ParameterList);  
};
```

Example:

cpp

CopyEdit

```
class Box {  
private:  
    double width;  
public:  
    Box() : width(0) {}  
    friend void printWidth(Box b);  
};
```

```
void printWidth(Box b) {  
    cout << "Width: " << b.width << endl;
```

}

34. How do friend functions differ from member functions?

- **Member Functions:**
 - Defined inside the class.
 - Can access all members (private, protected, and public) of the class.
 - Implicitly have a this pointer.
 - **Friend Functions:**
 - Defined outside the class.
 - Not members of the class but can access its private and protected members.
 - Do not have a this pointer.
-

35. Explain the benefits and potential drawbacks of using friend functions.

Benefits:

- **Access to Private Members:** Allows external functions to access private and protected members of a class.
- **Operator Overloading:** Useful for overloading operators that require access to private data.
- **Separation of Concerns:** Can help in separating the implementation of operations from the class definition.

Drawbacks:

- **Breaks Encapsulation:** Exposes internal details to external functions, potentially violating encapsulation.
 - **Maintenance Challenges:** Can make the code harder to maintain and understand.
 - **Limited Scope:** Friendship is not inherited, so derived classes do not automatically grant access to their friends.
-

36. What is inheritance in C++ and why is it important?

Definition:

- **Inheritance** is a mechanism in C++ that allows a class (derived class) to inherit properties and behaviors (data members and member functions) from another class (base class).

Importance:

- **Code Reusability:** Promotes reusability of code by allowing new classes to use existing functionality.

- **Extensibility:** Enables the creation of new classes based on existing ones, facilitating extension of functionality.
 - **Polymorphism:** Supports dynamic behavior and method overriding, leading to more flexible and maintainable code.
-

37. Explain the different types of inheritance in C++.

1. **Single Inheritance:** A derived class inherits from a single base class.

cpp

CopyEdit

```
class Base {};
```

```
class Derived : public Base {};
```

2. **Multiple Inheritance:** A derived class inherits from more than one base class.

cpp

CopyEdit

```
class Base1 {};
```

```
class Base2 {};
```

```
class Derived : public Base1, public Base2 {};
```

3. **Multilevel Inheritance:** A class derives from another class, which is also derived from another class.

cpp

CopyEdit

```
class Grandparent {};
```

```
class Parent : public Grandparent {};
```

```
class Child : public Parent {};
```

4. **Hierarchical Inheritance:** Multiple classes derive from a single base class.

cpp

CopyEdit

```
class Base {};
```

```
class Derived1 : public Base {};
```

```
class Derived2 : public Base {};
```

5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

cpp

CopyEdit

```
class Base {};  
  
class Intermediate : public Base {};  
  
class Derived : public Intermediate, public AnotherBase {};
```

38. How do you implement single inheritance in C++?

cpp

CopyEdit

```
class Base {  
  
public:  
  
    void display() {  
        cout << "Base class display function." << endl;  
    }  
};  
  
class Derived : public Base {  
  
public:  
  
    void show() {  
        cout << "Derived class show function." << endl;  
    }  
};  
  
int main() {  
    Derived obj;  
    obj.display();  
    obj.show();  
    return 0;  
}
```

Explanation:

- The Derived class inherits from the Base class using the public access specifier.
- It can access public members of the Base class.

39. What is multiple inheritance and how does it differ from single inheritance?

Multiple Inheritance:

- A derived class inherits from more than one base class.
- Allows a class to combine functionalities from multiple classes.

Difference from Single Inheritance:

- Single Inheritance: A derived class inherits from only one base class.
- Multiple Inheritance: A derived class inherits from multiple base classes.

Example:

cpp

CopyEdit

```
class ClassA {  
public:  
    void methodA() {}  
};
```

```
class ClassB {  
public:  
    void methodB() {}  
};
```

```
class Derived : public ClassA, public ClassB {  
public:  
    void methodC() {}  
};
```

40. Describe hierarchical inheritance with an example.

Definition:

- Multiple derived classes inherit from a single base class.

Example:

cpp

CopyEdit

```
class Animal {  
public:  
    void eat() {  
        cout << "Eating..." << endl;  
    }  
};
```

```
class Dog : public Animal {  
public:  
    void bark() {  
        cout << "Barking..." << endl;  
    }  
};
```

```
class Cat : public Animal {  
public:  
    void meow() {  
        cout << "Meowing..." << endl;  
    }  
};
```

Explanation:

- Both Dog and Cat classes inherit from the Animal class.

41. What is multilevel inheritance and how is it implemented in C++?

Definition:

Multilevel inheritance is a type of inheritance where a class is derived from a class, which is itself derived from another class, forming a chain of inheritance.

Example:

cpp

CopyEdit

```
class Grandparent {  
public:
```



```
void speak() {  
    cout << "Speaking from Grandparent" << endl;  
}  
};
```

```
class Parent : public Grandparent {  
public:  
    void talk() {  
        cout << "Talking from Parent" << endl;  
    }  
};
```

```
class Child : public Parent {  
public:  
    void say() {  
        cout << "Saying from Child" << endl;  
    }  
};
```

42. Explain the concept of hybrid inheritance.

Definition:

Hybrid inheritance is a combination of two or more types of inheritance (single, multiple, multilevel, or hierarchical). It is used when the design of a program demands multiple forms of inheritance.

Example:

cpp

CopyEdit

```
class A {  
public:  
    void showA() {}  
};
```

```
class B : public A {  
public:  
    void showB() {}  
};
```

```
class C {  
public:  
    void showC() {}  
};
```

```
class D : public B, public C {  
public:  
    void showD() {}  
};
```

43. What are access modifiers in C++ and what are the different types?

Access Modifiers:

They control the accessibility of class members.

Types:

1. **Public:** Members are accessible from anywhere.
 2. **Private:** Members are accessible only within the class.
 3. **Protected:** Members are accessible within the class and by derived classes.
-

44. How do public, private, and protected access modifiers affect inheritance?

- **Public Inheritance:**
 - Public members of the base remain public in the derived class.
 - Protected members remain protected.
 - Private members are not accessible.
- **Protected Inheritance:**
 - Public and protected members become protected in the derived class.
- **Private Inheritance:**

- Public and protected members become private in the derived class.

45. Explain how access modifiers control member accessibility in derived classes.

- Private members: Not accessible directly in derived classes.
- Protected members: Accessible in derived classes.
- Public members: Fully accessible in derived classes if inherited publicly.

Access modifiers ensure encapsulation and data hiding, allowing controlled interaction with class members.

46. What is function overriding in the context of inheritance?

Definition:

Function overriding occurs when a derived class redefines a base class function with the same name, return type, and parameters.

It enables runtime polymorphism.

47. How do you override a base class function in a derived class?

Example:

cpp

CopyEdit

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

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

- Use the same function signature in the derived class.
 - The base function should be marked as virtual for proper overriding.
-

48. Explain the use of the "virtual" keyword in function overriding.

Purpose:

The virtual keyword enables dynamic dispatch, allowing the derived class's version of a function to be called through a base class pointer or reference.

49. What is the significance of the "override" specifier in C++11 and later?

override keyword:

- Ensures that the function is meant to override a base class function.
- Helps catch errors at compile time (e.g., incorrect function signatures).

Example:

cpp

CopyEdit

```
void show() override; // ensures it overrides a base class virtual function
```

50. What is a virtual base class in C++ and why is it used?

Definition:

A virtual base class prevents multiple copies of a base class when using multiple inheritance.

Purpose:

Used to resolve the diamond problem where an ambiguity arises due to multiple inheritance paths to a base class.

51. How do you declare and implement a virtual base class?

Declaration:

cpp

CopyEdit

```
class A {  
public:  
    void display() {}  
};
```

```
class B : virtual public A {};
```

```
class C : virtual public A {};
```

```
class D : public B, public C {};
```

- Use virtual keyword during inheritance of the base class.

52. Explain the role of virtual base classes in resolving ambiguity in multiple inheritance.

Problem:

In multiple inheritance, if two derived classes inherit from the same base class, and a further class inherits from both, there are two copies of the base class.

Solution:

Using virtual inheritance ensures only one shared copy of the base class exists in the inheritance chain, eliminating ambiguity.

53. Provide an example of using a virtual base class to avoid the diamond problem in inheritance.

Example:

cpp

CopyEdit

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
public:
```

```
    void show() {
```

```
        cout << "Class A" << endl;
```

```
    }
```

```
};
```

```
class B : virtual public A {};
```

```
class C : virtual public A {};
```

```
class D : public B, public C {};
```

```
int main() {
```

```
D obj;

obj.show(); // No ambiguity

return 0;
}
```

Explanation:

- Without virtual, D would inherit two copies of A from B and C.
- Using virtual ensures only one shared instance of A.

Assignment 4 (Theory and Practical)

1. What is polymorphism in C++ and why is it important?

Polymorphism means "many forms" and in C++, it allows functions or objects to behave differently based on the context.

It is important because it enables code reusability, extensibility, and supports dynamic behavior, which is essential for object-oriented programming.

2. Explain the concept of compile-time (static) polymorphism with examples.

Static polymorphism is resolved at compile time. It includes:

- Function overloading: Same function name, different parameter types.
- Operator overloading: Custom behavior for operators.

Example:

cpp

CopyEdit

```
class Print {
public:
    void show(int x) { std::cout << "Integer: " << x << std::endl; }
    void show(double x) { std::cout << "Double: " << x << std::endl; }
};
```

3. Describe the concept of runtime (dynamic) polymorphism with examples.

Dynamic polymorphism is resolved at runtime using virtual functions and inheritance.

Example:

cpp

CopyEdit

```
class Animal {
public:
    virtual void speak() { std::cout << "Animal speaks\n"; }
};
```

```
class Dog : public Animal {
public:
    void speak() override { std::cout << "Dog barks\n"; }
};
```

```
int main() {
    Animal* a = new Dog();
    a->speak(); // Outputs: Dog barks
}
```

4. What is the difference between static and dynamic polymorphism?

Feature	Static Polymorphism	Dynamic Polymorphism
Resolved at	Compile-time	Runtime
Uses	Function/Operator overloading	Virtual functions, inheritance
Performance	Faster	Slightly slower
Flexibility	Less	More

5. How is polymorphism implemented in C++?

- Static: Achieved through function and operator overloading.
 - Dynamic: Achieved through virtual functions and base class pointers or references.
 - At runtime, C++ uses a vtable (virtual table) to resolve virtual function calls.
-

6. What are pointers in C++ and how do they work?

A pointer is a variable that stores the memory address of another variable. They allow dynamic memory allocation, array manipulation, and are essential for object-oriented features like polymorphism.

7. Explain the syntax for declaring and initializing pointers.

cpp

CopyEdit

```
int a = 10;
```

```
int* p = &a; // p holds the address of a
```

- `int* p`: Declares a pointer to an int.
- `&a`: "Address of" operator.

8. How do you access the value pointed to by a pointer?

Use the dereference operator (*):

cpp

CopyEdit

```
int a = 10;
```

```
int* p = &a;
```

```
std::cout << *p; // Outputs: 10
```

9. Describe the concept of pointer arithmetic.

You can perform arithmetic on pointers to navigate through memory (especially arrays):

cpp

CopyEdit

```
int arr[3] = {10, 20, 30};
```

```
int* p = arr;
```

```
p++; // Now points to arr[1]
```

```
std::cout << *p; // Outputs: 20
```

10. What are the common pitfalls when using pointers?

- Dangling pointers: Using pointers after memory is freed.
- Memory leaks: Not deallocating dynamically allocated memory.
- Null pointer dereference: Accessing memory via a null pointer.
- Pointer arithmetic errors: Going out of bounds.

- **Uninitialized pointers:** Using pointers before assigning them a valid address.

11. How are pointers used with objects in C++?

Pointers in C++ can be used to refer to objects. This allows dynamic allocation, manipulation of object properties, and accessing members via the pointer.

Example:

cpp

CopyEdit

```
class MyClass {  
public:  
    int x;  
    MyClass(int val) : x(val) {}  
};  
  
int main() {  
    MyClass* p = new MyClass(10); // Dynamically allocate an object  
    std::cout << p->x; // Access member via pointer (Outputs: 10)  
    delete p; // Don't forget to deallocate  
}
```

12. Explain the process of dynamically allocating objects using pointers.

To dynamically allocate an object, use the new keyword. The object will be created on the heap.

Example:

cpp

CopyEdit

```
MyClass* obj = new MyClass(5); // Allocate MyClass object dynamically
```

- new allocates memory for the object and returns a pointer to it.
- Remember to free the memory using delete when you're done with the object to avoid memory leaks.

13. Provide an example of accessing object members using pointers.

You can access object members using the -> operator when dealing with pointers.

Example:

cpp

CopyEdit

```
class MyClass {  
public:  
    int data;  
    void display() { std::cout << data << std::endl; }  
};  
  
int main() {  
    MyClass* obj = new MyClass;  
    obj->data = 20; // Access member using the pointer  
    obj->display(); // Call member function using the pointer  
    delete obj; // Free memory  
}
```

14. What is the difference between a pointer to an object and a reference to an object?

- Pointer: Can be nullptr, can point to null, can be reassigned to point to another object, and requires dereferencing.
 - Syntax: `ClassName* ptr = &obj;`
 - Reference: Cannot be null, must always refer to an existing object, and cannot be reassigned after initialization.
 - Syntax: `ClassName& ref = obj;`
-

15. How do you release dynamically allocated objects in C++?

Use the delete keyword to free memory allocated using new.

Example:

cpp

CopyEdit

```
MyClass* obj = new MyClass;  
  
delete obj; // Frees the memory allocated for obj  
  
• delete[] is used for arrays allocated with new[].
```

16. What is the this pointer in C++ and what is its significance?

The this pointer is an implicit pointer in non-static member functions that points to the current object. It allows you to refer to the object's data and functions inside the class.

Significance: It helps in distinguishing between member variables and function arguments with the same name.

17. How is the this pointer used in member functions?

The this pointer is used to access members of the current object inside a non-static member function.

Example:

cpp

CopyEdit

```
class MyClass {  
public:  
    int x;  
    MyClass(int x) { this->x = x; } // Use 'this' to differentiate parameter and member variable  
};
```

18. Explain how the this pointer can be used to return the current object.

The this pointer can be returned from a member function to enable method chaining or to refer to the current object.

Example:

cpp

CopyEdit

```
class MyClass {  
public:  
    MyClass* setX(int x) {  
        this->x = x;  
        return this;  
    }  
    int x;  
};
```

```
int main() {  
    MyClass obj;  
    obj.setX(5)->setX(10); // Chaining methods using 'this'  
}
```

19. What is a virtual function in C++ and why is it used?

A virtual function allows derived classes to override it, enabling runtime polymorphism. It ensures that the correct function is called for an object, regardless of the type of reference or pointer used.

20. Describe the syntax for declaring a virtual function.

Declare a function as virtual in the base class.

Example:

cpp

CopyEdit

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

21. Explain the concept of a vtable (virtual table) and its role in virtual functions.

A vtable is a mechanism used by C++ to implement dynamic polymorphism. It is a table of function pointers where each entry corresponds to a virtual function of a class. When a virtual function is called, the appropriate function is selected from the vtable based on the type of the object.

22. What is a pure virtual function and how is it declared?

A pure virtual function is a function that has no implementation in the base class and must be overridden in derived classes. It is declared by assigning 0 to the function declaration.

Syntax:

cpp

CopyEdit

```
class Base {
```

```
public:

    virtual void show() = 0; // Pure virtual function

};
```

23. Provide an example of a class with pure virtual functions.

cpp

CopyEdit

```
class Shape {

public:

    virtual void draw() = 0; // Pure virtual function

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

};
```

```
class Circle : public Shape {

public:

    void draw() override { std::cout << "Drawing Circle\n"; }

    double area() override { return 3.14 * radius * radius; }

private:

    double radius = 5;

};
```

24. What are the implications of having pure virtual functions in a class?

- A class containing pure virtual functions becomes an abstract class.
 - Abstract classes cannot be instantiated directly.
 - Derived classes must implement all pure virtual functions to become concrete.
-

25. How is polymorphism implemented using inheritance and virtual functions?

Inheritance allows derived classes to override base class methods, and virtual functions enable the correct method to be called for an object based on its runtime type.

Example:

cpp

CopyEdit

```
class Base {  
  
public:  
    virtual void show() { std::cout << "Base class\n"; }  
};  
  
class Derived : public Base {  
  
public:  
    void show() override { std::cout << "Derived class\n"; }  
};
```

26. Provide an example of implementing polymorphism with base and derived classes.

cpp

CopyEdit

```
class Base {  
  
public:  
    virtual void show() { std::cout << "Base class\n"; }  
};  
  
class Derived : public Base {  
  
public:  
    void show() override { std::cout << "Derived class\n"; }  
};  
  
int main() {  
    Base* b = new Derived();  
    b->show(); // Outputs: Derived class  
}
```

27. Explain the concept of late binding in the context of polymorphism.

Late binding occurs when the function to be called is determined at runtime (e.g., using virtual functions). This allows for dynamic dispatch and polymorphism.

28. How does the compiler manage polymorphism in C++?

The compiler uses a vtable to manage virtual functions and ensures that the correct function is called for an object during runtime. The virtual function calls are resolved through a mechanism of dynamic dispatch.

29. What is an abstract class in C++?

An abstract class is a class that contains at least one pure virtual function. It cannot be instantiated and serves as a base for derived classes.

30. How do abstract classes differ from regular classes?

- Abstract classes cannot be instantiated and are designed to be inherited.
- Regular classes can be instantiated and may or may not contain pure virtual functions.

31. Explain the role of abstract methods in abstract classes.

Abstract methods (also called pure virtual functions) are functions that are declared in an abstract class but have no implementation. They are meant to be overridden in derived classes. Any class containing at least one abstract method becomes an abstract class and cannot be instantiated. Abstract methods define a contract that derived classes must follow.

Example:

cpp

CopyEdit

```
class AbstractClass {  
public:  
    virtual void show() = 0; // Pure virtual function (abstract method)  
};
```

32. Provide an example of defining and using an abstract class.

cpp

CopyEdit

```
class Shape {  
public:
```

```

    virtual void draw() = 0; // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override { std::cout << "Drawing Circle\n"; }
};

int main() {
    // Shape s; // Error: Cannot instantiate an abstract class
    Circle c;
    c.draw(); // Outputs: Drawing Circle
}

```

- Shape is an abstract class because it contains a pure virtual function draw().

33. What are the benefits of using abstract classes in C++?

- **Polymorphism:** Abstract classes allow for the implementation of polymorphic behavior in C++.
- **Code Organization:** They provide a clear contract for derived classes, promoting structured and organized code.
- **Enforce Interface:** Abstract classes ensure that derived classes implement specific methods.
- **Separation of concerns:** Abstract classes allow you to define shared behavior while leaving the implementation details to derived classes.

34. What is exception handling in C++ and why is it important?

Exception handling in C++ is a mechanism to handle runtime errors, allowing the program to continue executing even if something goes wrong. It is important because it provides a way to manage errors gracefully, improving the robustness and reliability of the program.

35. Describe the syntax for throwing and catching exceptions in C++.

The syntax for throwing and catching exceptions is as follows:

- **Throw:** The throw keyword is used to raise an exception.

- **Catch:** The catch block is used to handle exceptions.

Example:

cpp

CopyEdit

```
try {  
    throw std::runtime_error("Something went wrong");  
} catch (const std::exception& e) {  
    std::cout << "Caught exception: " << e.what() << std::endl;  
}
```

36. Explain the concept of try, catch, and throw blocks.

- **try:** Defines a block of code where exceptions might be thrown.
- **throw:** Used to signal that an exception has occurred and is propagated to the nearest catch block.
- **catch:** Catches exceptions thrown by the try block and handles them accordingly.

Example:

cpp

CopyEdit

```
try {  
    throw 10; // Throw an integer exception  
} catch (int e) { // Catch block handles integer exceptions  
    std::cout << "Caught exception: " << e << std::endl;  
}
```

37. What is the role of the catch block in exception handling?

The catch block is used to catch exceptions thrown in the try block. It provides a way to handle errors and exceptions without terminating the program. Multiple catch blocks can be used to handle different types of exceptions.

38. Provide an example of handling multiple exceptions in C++.

You can have multiple catch blocks to handle different exception types.

Example:

cpp

CopyEdit

```
try {  
    throw 5; // Integer exception  
} catch (int e) {  
    std::cout << "Caught integer exception: " << e << std::endl;  
} catch (const std::exception& e) {  
    std::cout << "Caught exception: " << e.what() << std::endl;  
}
```

39. How does the throw keyword work in exception handling?

The throw keyword is used to generate an exception. When an exception is thrown, control is transferred to the nearest catch block that matches the type of the exception.

Example:

cpp

CopyEdit

```
throw "Error: Something went wrong"; // Throwing an exception
```

40. What is the purpose of the finally block in exception handling?

C++ does not have a finally block. However, in languages like Java, a finally block is used to guarantee that certain code will run regardless of whether an exception is thrown or not, such as cleaning up resources. In C++, you can achieve similar functionality using RAII (Resource Acquisition Is Initialization) or by putting cleanup code in a catch block or after the try-catch block.

41. How do you create custom exception classes in C++?

To create custom exceptions, you typically inherit from `std::exception` or one of its derived classes, and override the `what()` function to provide an error message.

Example:

cpp

CopyEdit

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

```
        return "Custom exception occurred";
    }
};
```

42. What are templates in C++ and why are they useful?

Templates in C++ allow you to write generic functions and classes that can work with any data type. Templates make code more flexible and reusable, reducing the need for writing separate versions of the same function or class for different data types.

43. Describe the syntax for defining a function template.

A function template is defined using the template keyword, followed by a type parameter.

Example:

cpp

CopyEdit

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

44. Provide an example of a function template that performs a generic operation.

cpp

CopyEdit

```
template <typename T>
T multiply(T a, T b) {
    return a * b;
}

int main() {
    std::cout << multiply(4, 5) << std::endl; // Works for int
    std::cout << multiply(2.5, 3.5) << std::endl; // Works for double
}
```

45. What is a class template and how is it different from a function template?

A class template allows you to define a class blueprint that can work with any data type. Unlike a function template, which is used to define functions, a class template defines entire classes.

46. Explain the syntax for defining a class template.

The syntax for a class template is similar to a function template, but it's for classes.

Example:

cpp

CopyEdit

```
template <typename T>
class Box {
public:
    T value;
    Box(T val) : value(val) {}
    T getValue() { return value; }
};
```

47. Provide an example of a class template that implements a generic data structure.

cpp

CopyEdit

```
template <typename T>
class Stack {
private:
    std::vector<T> elements;
public:
    void push(T value) { elements.push_back(value); }
    T pop() {
        T value = elements.back();
        elements.pop_back();
        return value;
    }
};
```

};

48. How do you instantiate a template class in C++?

To instantiate a template class, specify the data type in angle brackets.

Example:

cpp

CopyEdit

```
Stack<int> intStack; // Instantiating a Stack for int
```

```
Stack<double> doubleStack; // Instantiating a Stack for double
```

49. What are the advantages of using templates over traditional class inheritance?

- **Code Reusability:** Templates allow you to write code once and use it with any type.
 - **Type Safety:** Templates ensure type safety, while inheritance can sometimes lead to issues like slicing or type casting.
 - **Flexibility:** Templates are more flexible than inheritance when creating generic data structures or algorithms.
-

50. How do templates promote code reusability in C++?

Templates allow you to write generic, reusable code. Instead of writing multiple functions or classes for different data types, you write a single function or class template, and it automatically works with any type provided at compile-time.

Assignment 5 (Theory and Practical)

1. What are streams in C++ and why are they important?

Streams in C++ are abstract representations of input/output (I/O) devices. They provide a consistent way to read from input sources (like keyboard or files) and write to output targets (like console or files), making I/O operations easier and more uniform.

2. Explain the different types of streams in C++.

C++ provides three primary types of streams:

- **Input streams (istream)** – e.g., cin, ifstream
 - **Output streams (ostream)** – e.g., cout, ofstream
 - **Bidirectional streams (iostream)** – e.g., fstream (for both input and output)
-

3. How do input and output streams differ in C++?

- Input streams (istream) are used to read data into the program.
- Output streams (ostream) are used to write data from the program.

Example:

cpp

CopyEdit

```
int x;
```

```
std::cin >> x; // input
```

```
std::cout << x; // output
```

4. Describe the role of the iostream library in C++.

The iostream library provides essential stream classes like istream, ostream, and iostream, and objects like cin, cout, cerr, and clog, enabling console input/output operations.

5. What is the difference between a stream and a file stream?

- Stream is a general I/O abstraction for data flow.
 - File stream (ifstream, ofstream, fstream) is a stream specifically designed for reading from or writing to files.
-

6. What is the purpose of the cin object in C++?

cin is an instance of istream and is used to take input from the standard input device (usually the keyboard).

7. How does the cin object handle input operations?

cin uses the extraction operator (>>) to read data from the user and store it into variables. It skips whitespace by default and converts data to the appropriate type.

8. What is the purpose of the cout object in C++?

cout is an instance of ostream and is used to display output on the standard output device (usually the console).

9. How does the cout object handle output operations?

cout uses the insertion operator (<<) to send data to the output stream, converting data into a textual format suitable for display.

10. Explain the use of the insertion (<<) and extraction (>>) operators in conjunction with cin and cout.

- << (insertion): Used with cout to print output.
 - std::cout << "Hello";
 - >> (extraction): Used with cin to read input.
 - std::cin >> x;
-

11. What are the main C++ stream classes and their purposes?

Class	Purpose
istream	Input from keyboard or files
ostream	Output to console or files
ifstream	Input from files
ofstream	Output to files
fstream	Input/output to files
stringstream	I/O with strings

12. Explain the hierarchy of C++ stream classes.

cpp

CopyEdit

ios

└─ istream

| └─ ifstream

| └─ istringstream

└─ ostream

| └─ ofstream

| └─ ostringstream

└─ iostream

└─ fstream
└─ stringstream

13. What is the role of the istream and ostream classes?

- **istream:** Base class for input streams (cin, ifstream)
 - **ostream:** Base class for output streams (cout, ofstream)
They provide overloaded operators and methods for handling input and output.
-

14. Describe the functionality of the ifstream and ofstream classes.

- **ifstream:** Reads data from files (input).
- **ofstream:** Writes data to files (output).

Example:

cpp

CopyEdit

```
std::ifstream fin("data.txt");  
std::ofstream fout("output.txt");
```

15. How do the fstream and stringstream classes differ from other stream classes?

- **fstream:** Supports both input and output with files.
 - **stringstream:** Reads/writes to/from a std::string instead of a file or console.
-

16. What is unformatted I/O in C++?

Unformatted I/O deals with raw character data without formatting or type conversion.

17. Provide examples of unformatted I/O functions.

- **get():** Reads a single character.
- **put():** Writes a single character.
- **read():** Reads binary data.
- **write():** Writes binary data.

Example:

cpp

CopyEdit

```
char ch;  
  
std::cin.get(ch);  
  
std::cout.put(ch);
```

18. What is formatted I/O in C++?

Formatted I/O involves presenting data in a structured way (with precision, width, etc.). It includes type conversions and data formatting.

19. How do you use manipulators to perform formatted I/O in C++?

Use `<iomanip>` header manipulators like:

- `std::setw(n)`: Set width
- `std::setprecision(n)`: Set precision
- `std::fixed`: Fixed-point notation

Example:

cpp

CopyEdit

```
#include <iomanip>  
  
std::cout << std::fixed << std::setprecision(2) << 3.14159;
```

20. Explain the difference between unformatted and formatted I/O operations.

Feature	Unformatted I/O	Formatted I/O
Data processing	Raw characters	Type conversion and formatting
Speed	Faster	Slightly slower

Example functions `get()`, `put()`, `read()`, `write()` `>>`, `<<`, `setw`, `setprecision`

21. What are manipulators in C++?

Manipulators are functions that modify the formatting of input and output streams. They allow you to control how data is read from or written to streams.

22. How do manipulators modify the behavior of I/O operations?

Manipulators affect the formatting of data—such as setting field width, precision, alignment, and number formats—without altering the data itself. They are used with << or >> operators in I/O streams.

23. Provide examples of commonly used manipulators in C++.

Common manipulators (in <iomanip>):

- `std::setw(n)` – Set width
 - `std::setprecision(n)` – Set decimal precision
 - `std::fixed` – Fixed-point notation
 - `std::left` / `std::right` – Align output
 - `std::hex`, `std::dec`, `std::oct` – Set numeric base
-

24. Explain the use of the `setw`, `setprecision`, and `fixed` manipulators.

cpp

CopyEdit

```
#include <iostream>
```

```
#include <iomanip>
```

```
int main() {
```

```
    double pi = 3.14159265;
```

```
    std::cout << std::setw(10) << "Value" << std::endl;    // Sets column width
```

```
    std::cout << std::setprecision(4) << pi << std::endl;    // Sets significant digits
```

```
    std::cout << std::fixed << std::setprecision(2) << pi;    // Fixed-point with 2 decimal places
```

```
}
```

25. How do you create custom manipulators in C++?

Custom manipulators are functions that return a reference to the stream and accept a stream as a parameter.

Example:

cpp

CopyEdit

```
std::ostream& myEndl(std::ostream& os) {  
    os << "\n----\n";  
    return os;  
}
```

```
std::cout << "Hello" << myEndl;
```

26. What is a file stream in C++ and how is it used?

A file stream is a stream used to perform input/output operations on files using classes like `ifstream`, `ofstream`, and `fstream`.

27. Explain the process of opening and closing files using file streams.

cpp

CopyEdit

```
std::ofstream fout("file.txt"); // Open for writing
```

```
std::ifstream fin("file.txt"); // Open for reading
```

```
// Use the file...
```

```
fout << "Hello";
```

```
fout.close(); // Always close files after use
```

```
fin.close();
```

You can also use `.open()` and `.close()`:

cpp

CopyEdit

```
fout.open("file.txt");
```

```
fout.close();
```

28. Describe the different modes in which a file can be opened.

File open modes (in `<fstream>`):

- `std::ios::in` – Read mode

- `std::ios::out` – Write mode
- `std::ios::app` – Append mode
- `std::ios::binary` – Binary mode
- `std::ios::ate` – Start at end of file
- `std::ios::trunc` – Truncate existing content

Example:

cpp

CopyEdit

```
std::ofstream fout("data.txt", std::ios::app);
```

29. How do you read from and write to files using file streams?

cpp

CopyEdit

```
std::ofstream fout("data.txt");
```

```
fout << "Writing to file\n";
```

```
fout.close();
```

```
std::ifstream fin("data.txt");
```

```
std::string line;
```

```
std::getline(fin, line);
```

```
std::cout << line;
```

```
fin.close();
```

30. Provide an example of using file streams to copy the contents of one file to another.

cpp

CopyEdit

```
std::ifstream source("source.txt");
```

```
std::ofstream dest("dest.txt");
```

```
std::string line;
```

```
while (std::getline(source, line)) {
```

```
    dest << line << std::endl;
}
```

```
source.close();
```

```
dest.close();
```

31. What are the main C++ file stream classes and their purposes?

Class	Purpose
-------	---------

ifstream	Input from a file
----------	-------------------

ofstream	Output to a file
----------	------------------

fstream	Input and output to a file
---------	----------------------------

32. Explain the role of the ifstream, ofstream, and fstream classes.

- ifstream: Used to read from files (read()/>>)
 - ofstream: Used to write to files (write()/<<)
 - fstream: Combines ifstream and ofstream to support both reading and writing
-

33. How do you use the ifstream class to read data from a file?

cpp

CopyEdit

```
std::ifstream fin("file.txt");
```

```
std::string word;
```

```
while (fin >> word) {
```

```
    std::cout << word << std::endl;
```

```
}
```

```
fin.close();
```

34. How do you use the ofstream class to write data to a file?

cpp

CopyEdit

```
std::ofstream fout("output.txt");  
  
fout << "This is a test." << std::endl;  
  
fout.close();
```

35. Describe the functionality of the fstream class for both input and output operations.

cpp

CopyEdit

```
std::fstream file("data.txt", std::ios::in | std::ios::out);  
  
std::string word;  
  
file >> word;      // Read  
  
file << "New data"; // Write  
  
file.close();
```

36. What are file management functions in C++?

These are standard library functions that manage file operations:

- `remove("filename")` – Deletes a file
 - `rename("old", "new")` – Renames a file
 - `seekg()`, `seekp()` – Move input/output pointers
 - `tellg()`, `tellp()` – Get input/output pointer position
-

37. How do you use the remove and rename functions to manage files?

cpp

CopyEdit

```
#include <cstdio>  
  
remove("old.txt");      // Deletes the file  
  
rename("temp.txt", "final.txt"); // Renames the file
```

38. Explain the purpose of the seekg and seekp functions in file management.

- `seekg(pos)` – Sets get (input) pointer to a specific location.
- `seekp(pos)` – Sets put (output) pointer to a specific location.

These functions are used to read from or write to specific positions in a file.

39. Provide examples of using file management functions to manipulate file pointers.

cpp

CopyEdit

```
std::ifstream fin("data.txt");  
fin.seekg(5);           // Move to the 6th byte  
char ch;  
fin.get(ch);  
std::cout << ch;        // Prints the character at position 5  
fin.close();
```

40. What are file modes in C++?

File modes define how files are opened. Common modes include:

- `std::ios::in`: Open for reading
- `std::ios::out`: Open for writing
- `std::ios::app`: Append to end
- `std::ios::binary`: Open in binary mode
- `std::ios::trunc`: Truncate file to zero length
- `std::ios::ate`: Start at end of file

Modes can be combined using bitwise OR (`|`).

41. Describe the different file modes available in C++.

C++ file modes (defined in `<fstream>`) specify how a file is opened:

Mode	Description
<code>ios::in</code>	Open for input (read).
<code>ios::out</code>	Open for output (write).
<code>ios::app</code>	Append to end of file.
<code>ios::ate</code>	Open and move to end, but allow writing anywhere.
<code>ios::trunc</code>	Truncate file to zero length if it exists.
<code>ios::binary</code>	Open file in binary mode instead of text mode.

You can combine them using the bitwise OR operator (`|`).

42. How do you specify a file mode when opening a file?

You specify file modes using the second argument in the `.open()` function or constructor of `ifstream`, `ofstream`, or `fstream`.

Example:

cpp

CopyEdit

```
std::ofstream fout("file.txt", std::ios::out | std::ios::app);
```

43. Explain the difference between binary and text file modes.

- **Text Mode:** Interprets data as human-readable characters. Line endings may be converted (e.g., `\n` to `\r\n` on Windows).
 - **Binary Mode:** Treats the file as a raw sequence of bytes with no conversion, making it suitable for storing non-text data like images or serialized objects.
-

44. Provide examples of opening files in different modes using file streams.

cpp

CopyEdit

```
std::ifstream fin("input.txt", std::ios::in); // Read text file
```

```
std::ofstream fout("output.txt", std::ios::out | std::ios::app); // Append to file
```

```
std::fstream bin("data.bin", std::ios::in | std::ios::out | std::ios::binary); // Binary read/write
```

45. What are binary files in C++ and how do they differ from text files?

Binary files store data in raw byte format. They do not contain human-readable characters and can efficiently store structured data like structs, classes, or arrays.

Differences:

- **Text file:** Human-readable, slow to parse, subject to encoding issues.
 - **Binary file:** Compact, fast, but not human-readable.
-

46. Explain the process of reading from and writing to binary files.

Use `read()` and `write()` functions with `fstream`, `ifstream`, or `ofstream` opened in `ios::binary` mode.

Example (writing and reading a struct):

cpp

CopyEdit

```
struct Data {
```

```
    int id;
```

```
    char name[20];
```

```
};
```

```
Data d = {1, "Alice"};
```

```
std::ofstream fout("data.bin", std::ios::binary);
```

```
fout.write(reinterpret_cast<char*>(&d), sizeof(d));
```

```
fout.close();
```

```
Data d2;
```

```
std::ifstream fin("data.bin", std::ios::binary);
```

```
fin.read(reinterpret_cast<char*>(&d2), sizeof(d2));
```

```
fin.close();
```

47. What are random access files in C++?

Random access files allow you to move to a specific byte (position) in a file and read or write from there. This is useful for updating or reading only a part of a large file without processing it entirely.

48. How do you perform random access operations on files?

Use the following methods:

- seekg(pos) – Move input (get) pointer
- seekp(pos) – Move output (put) pointer
- tellg() – Get current input pointer position
- tellp() – Get current output pointer position

49. Provide examples of using file streams to implement random access in binary files.

Example: Updating a specific record in a binary file:

```
struct Record {
```

```
int id;  
char name[20];  
};  
  
std::fstream file("records.bin", std::ios::in | std::ios::out | std::ios::binary);
```

```
int recordNumber = 2;  
Record newRecord = {3, "Updated"};
```

```
file.seekp(recordNumber * sizeof(Record), std::ios::beg);  
file.write(reinterpret_cast<char*>(&newRecord), sizeof(newRecord));  
file.close();
```

This example updates the 3rd record in a binary file (index 2). Using `seekp()` allows you to jump directly to the needed record.