ASSIGNMENT-4

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

Polymorphism in C++ allows objects of different types to be treated as objects of a common base class. It enables the same interface to represent different underlying forms (data types). There are two main types: compile-time (static) and runtime (dynamic). Compile-time polymorphism uses function and operator overloading. Runtime polymorphism uses virtual functions and inheritance. Polymorphism enhances code flexibility and maintainability. It supports dynamic method binding and reduces code duplication. It is essential in large-scale software for modularity. Using polymorphism, developers can write generic and reusable code. It allows adding new features without altering existing code. It is a core principle of object-oriented programming. It improves the readability and scalability of programs. It is widely used in frameworks and APIs. Polymorphism enables a clean and logical code structure.

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

Compile-time polymorphism occurs when function calls are resolved at compile time. It is achieved through function overloading and operator overloading. Function overloading allows defining multiple functions with the same name but different parameters. For example:

```
cpp
CopyEdit
int add(int a, int b);
float add(float a, float b);
```

Operator overloading lets you redefine operators for user-defined types. Example: Overloading `+` in a `Complex` class. Since decisions are made during compilation, execution is faster. It improves code clarity by using the same function/operator name for similar actions. Compile-time polymorphism does not support late binding. It is widely used when behavior doesn't need to change at runtime. It promotes code reuse and simplicity. However, it's less flexible than runtime polymorphism.

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

Runtime polymorphism is resolved during program execution. It is implemented using inheritance and virtual functions. A base class declares a virtual function, and derived classes override it. For example:

```
cpp
CopyEdit
class Animal {
public: virtual void speak() { cout << "Animal speaks"; }
};
class Dog : public Animal {
public: void speak() override { cout << "Dog barks"; }
};
```

Using a base class pointer (`Animal* a = new Dog();`) calls the derived class method at runtime. This is due to dynamic dispatch via the virtual table (vtable). It allows code to work with different types using a common interface. It enables flexible program behavior. Runtime polymorphism supports plug-in architectures. It promotes extensibility and decouples code. It enables late binding, crucial for OOP design. However, it may be slower than compile-time polymorphism.

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

Static polymorphism is resolved at compile time, dynamic at runtime. Static uses function/operator overloading. Dynamic uses inheritance and virtual functions. Static polymorphism is faster due to early binding. Dynamic polymorphism uses virtual tables for late binding. Static polymorphism doesn't support overriding behavior at runtime. Dynamic allows derived classes to override base class behavior. Static is less flexible but more efficient. Dynamic provides flexibility and extensibility. Static is implemented by the compiler, dynamic by the runtime system. Static is used for similar operations with different types. Dynamic is used when behavior needs to vary at runtime. Both promote reusability and abstraction. Their use depends on performance vs. flexibility needs.

5. How is polymorphism implemented in C++?

Polymorphism is implemented using function overloading, operator overloading, and inheritance with virtual functions. Compile-time polymorphism uses overloaded functions and operators. For example:

```
cpp
CopyEdit
int add(int, int);
float add(float, float);
```

Dynamic polymorphism uses a virtual function in a base class. Derived classes override this function. The base class pointer or reference can invoke the overridden function at runtime. This is managed using a virtual table (vtable) and a virtual pointer (vptr). When an object is created,

vptr points to the vtable of its class. The vtable contains addresses of overridden functions. This mechanism supports runtime dispatch. Virtual functions are declared using the `virtual` keyword. Polymorphism promotes modular, extensible, and maintainable code.

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

Pointers are variables that store the memory address of another variable. They are declared using the `*` operator. Example:

```
cpp
CopyEdit
int a = 10;
int* p = &a;
```

Here, `p` stores the address of `a`, and `*p` accesses the value. Pointers support dynamic memory allocation and data structures like linked lists. They can point to any data type. They are used in arrays, functions, and objects. Dereferencing a pointer gives access to the data stored at that address. They enable efficient memory usage. Pointers can also be null or uninitialized, which may cause errors. They support complex structures like function pointers and pointer to pointer. Pointers are powerful but must be handled carefully to avoid segmentation faults.

7. Explain the syntax for declaring and initializing pointers.

To declare a pointer, use the `*` symbol before the pointer name. Example:

```
cpp
CopyEdit
int a = 5;
int* ptr = &a;
```

Here, `ptr` is a pointer to an integer and stores the address of `a`. You can also declare multiple pointers:

```
cpp
CopyEdit
int *p1, *p2;
```

Pointers can be initialized to `nullptr` to indicate they point to nothing:

```
cpp
CopyEdit
int* p = nullptr;
```

Always initialize pointers before use. You can assign one pointer to another:

```
cpp
CopyEdit
int* p2 = p;
```

The `&` operator is used to get the address of a variable. The `*` operator is used to dereference and access the value. Proper initialization prevents undefined behavior and crashes.

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

You can access the value pointed to by a pointer using the dereference operator `*`. For example:

```
cpp
CopyEdit
int a = 10;
int* p = &a;
cout << *p; // Outputs 10
```

Here, `*p` gives the value of `a`. Dereferencing a pointer retrieves the data stored at the memory address it points to. Be cautious while dereferencing null or uninitialized pointers, as it can cause runtime errors. Dereferencing is also used in pointer arithmetic and dynamic memory. You can use `*p = 20;` to modify the value of `a`. It allows indirect access to variables and is fundamental to pointer operations. It is also used in object and array pointer access.

9. Describe the concept of pointer arithmetic.

Pointer arithmetic involves performing operations on pointer values. Valid operations include addition, subtraction, increment, and decrement. If `int* p` points to an integer at address 1000, `p+1` points to address 1004 (assuming `int` is 4 bytes). This is due to type-aware arithmetic. Example:

```
cpp
CopyEdit
int arr[] = {10, 20, 30};
int* p = arr;
cout << *(p + 1); // Outputs 20
```

You can traverse arrays using pointers. You can also subtract pointers to find distance:

```
cpp
CopyEdit
```

```
int* q = p + 2;
int diff = q - p; // 2
```

Pointer arithmetic must only be done within the bounds of the same array. It's commonly used in dynamic arrays and buffer manipulation. Improper arithmetic can lead to segmentation faults.

10. What are the common pitfalls when using pointers?

Common pitfalls include using uninitialized or null pointers, which can cause crashes. Dangling pointers point to memory that has been freed. Memory leaks occur when dynamically allocated memory isn't released. Incorrect pointer arithmetic can cause undefined behavior. Double deletion of memory using `delete` can crash the program. Pointer type mismatches lead to incorrect data access. Overwriting memory using incorrect dereferencing or bounds violation is dangerous. Misusing pointers can corrupt data or crash the system. Avoid using `void*` without proper casting. Not checking for `nullptr` before dereferencing is risky. Use smart pointers in modern C++ to manage memory. Proper initialization, careful use, and cleanup are crucial

11. What is a virtual function and how does it work?

A virtual function in C++ is a member function in the base class that can be overridden in a derived class. It is declared using the `virtual` keyword. When accessed through a base class pointer or reference, the overridden function in the derived class is called (if it exists). This is known as runtime polymorphism. Example:

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

If `Base* ptr = new Derived(); ptr->show();` is called, it prints "Derived". This happens through a mechanism called the virtual table (vtable). Virtual functions allow dynamic dispatch. They are essential for interface design. If a virtual function isn't overridden, the base class version is used. They support extensibility and flexibility in object-oriented design. Virtual destructors ensure proper cleanup in polymorphic classes.

12. Explain the role of the virtual keyword in C++.

The `virtual` keyword tells the compiler to support late binding for a function. It allows derived classes to override a function, and ensures the correct function is called at runtime when using a base pointer. Without `virtual`, the base class function would always be called. It is used only in base class declarations. It is essential for enabling polymorphism. Functions declared as virtual can be overridden using the same signature. `virtual` supports dynamic dispatch using vtables. If a class has virtual functions, it gets a virtual table pointer (vptr). `virtual` is also used in virtual destructors to avoid memory leaks. It doesn't affect non-inherited or non-pointer calls. It can be combined with `override` for clarity and safety. Using `virtual` promotes flexible and extensible class hierarchies.

13. What is a pure virtual function and an abstract class?

A **pure virtual function** has no definition in the base class and must be overridden in derived classes. It is declared using `= 0` syntax:

```
cpp
CopyEdit
virtual void draw() = 0;
```

A class with at least one pure virtual function is called an **abstract class**. Abstract classes cannot be instantiated directly. They provide a blueprint for derived classes. Derived classes must implement all pure virtual functions to become concrete. Abstract classes are used to define interfaces and ensure consistent APIs. They promote design by contract. They allow partial implementation in the base class. You can still have non-pure member functions in an abstract class. Abstract classes support polymorphism and code reuse. They are crucial in large applications for defining consistent structures and behaviors.

14. Explain the concept of base class and derived class.

A base class (or parent class) is a general class from which other classes can inherit. A derived class (or child class) inherits attributes and behaviors of the base class. In C++, inheritance is specified as:

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

The base class provides common functionality. The derived class can reuse, extend, or override this functionality. Inheritance promotes code reuse and hierarchy modeling. Access specifiers

(`public`, `protected`, `private`) control what members are inherited and how. The derived class can also add new members. Inheritance supports polymorphism when virtual functions are involved. Constructors and destructors of base and derived classes execute in order. Base class constructors are called before derived ones. Inheritance helps in modeling real-world relationships and creating scalable software architectures.

15. What is inheritance and its types in C++?

Inheritance is a feature in C++ that allows a class (derived) to acquire properties and behaviors of another class (base). It supports reusability and modularity. C++ supports several types of inheritance:

1. **Single inheritance** – One base, one derived class.
2. **Multiple inheritance** – A class inherits from more than one base class.
3. **Multilevel inheritance** – A class inherits from a derived class.
4. **Hierarchical inheritance** – Multiple classes inherit from one base class.
5. **Hybrid inheritance** – Combination of two or more types.
   Example:

```
cpp
CopyEdit
class A {};
class B : public A {};
```

Inheritance enables polymorphism, improves maintainability, and represents real-world hierarchies. Care must be taken with access specifiers and ambiguity in multiple inheritance.

16. What is function overloading in C++?

Function overloading is when multiple functions have the same name but different parameter lists. It allows functions to perform similar tasks with different inputs. The compiler differentiates them based on the number and types of parameters. Example:

```
cpp
CopyEdit
int add(int a, int b);
float add(float a, float b);
```

This is resolved at compile time (static polymorphism). Overloading improves code readability and consistency. It eliminates the need for unique function names for similar tasks. Default arguments can also be part of overload resolution. Overloaded functions must differ in their

parameter signatures. Return type alone cannot distinguish overloaded functions. It enhances usability and maintains logical grouping of operations.

17. Explain operator overloading with example.

Operator overloading allows redefining the meaning of C++ operators for user-defined types (like classes). You can provide custom behavior for operators like +, −, ==, etc. Example:

```
cpp
CopyEdit
class Complex {
public:
  int real, imag;
  Complex operator+(const Complex& obj) {
    Complex res;
    res.real = real + obj.real;
    res.imag = imag + obj.imag;
    return res;
  }
};
```

This allows adding complex numbers using `c1 + c2`. Operator overloading makes classes intuitive to use. It improves code clarity. Not all operators can be overloaded (e.g., `::`, `.`). Overloading must be done carefully to maintain consistency and expected behavior.

18. What is the difference between overloading and overriding?

Overloading occurs when two or more functions have the same name but different parameter lists. It happens at compile time. It's a form of static polymorphism.
Overriding occurs when a derived class provides a specific implementation of a virtual function declared in the base class. It happens at runtime and supports dynamic polymorphism. Overloading doesn't require inheritance. Overriding requires a base and derived class relationship. Overloading can occur in the same class. Overriding must use virtual functions. Overloading is based on function signature. Overriding is based on inheritance hierarchy. Overloading helps in code consistency. Overriding helps in behavioral customization. Both enhance readability and modular design.

19. What is a friend function in C++?

A friend function is a non-member function that can access the private and protected members of a class. It is declared using the `friend` keyword inside the class. Example:

```
cpp
CopyEdit
class Box {
  private: int length;
  public:
    friend void printLength(Box);
};
```

Friend functions are useful for operations involving multiple classes. They are not called with object syntax (`object.function()`). They provide controlled access to class internals. They break encapsulation slightly, so they should be used judiciously. They can also be friend classes. Friend functions cannot be inherited. They improve flexibility in operator overloading and external operations.

20. What are the characteristics of friend functions?

- Declared using the `friend` keyword.
- Can access private and protected members.
- Not members of the class but have special access.
- Cannot be called using object notation (use normal function call).
- Useful for operator overloading involving multiple objects.
- Can be declared as friends in more than one class.
- Not inherited by derived classes.
- Defined outside the class like normal functions.
- Not subject to the same access control as member functions.
- Useful in situations where mutual access between classes is required.
- Should be used sparingly to avoid violating encapsulation.

21. What is a class template in C++?

A class template allows you to define a generic class that works with any data type. It helps create reusable code. Templates use the keyword `template` followed by template parameters. Example:

```
cpp
CopyEdit
template <class T>
class Box {
  T value;
```

```
public:
  void set(T v) { value = v; }
  T get() { return value; }
};
```

You can then create `Box<int>`, `Box<float>`, etc. Templates reduce code duplication and enhance flexibility. The actual type is specified at object creation. Class templates are widely used in the STL (like vector, list). They support strong type safety. Templates are instantiated at compile time. Complex logic can be generalized using templates. They also support multiple type parameters.

22. What is a function template in C++?

A function template defines a generic function that works with different data types. It avoids writing separate functions for each type. It uses `template <class T>` syntax. Example:

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

You can call `add<int>(3, 4)` or `add<float>(3.5, 2.5)`. The compiler generates the appropriate version at compile time. Function templates improve reusability and maintainability. They support type safety and reduce redundancy. They work well with user-defined and primitive types. Overloaded versions can coexist with template functions. Templates form the foundation of the Standard Template Library (STL).

23. What are the advantages of templates in C++?

Templates promote **code reusability** and **type independence**. A single template works with multiple data types. They reduce redundancy by avoiding duplicate code for different types. Templates provide **type safety** during compile time. They allow better **performance** since instantiation happens at compile time. They are essential in **generic programming**. Templates help create **STL containers** like `vector`, `map`, etc. They also support **metaprogramming** in C++. With templates, developers can create **flexible and extensible libraries**. They encourage abstraction. Templates simplify code maintenance and improve readability. Using templates avoids errors due to copy-paste implementations for different types. They support multiple type parameters for more generic designs.

24. What is exception handling in C++?

Exception handling in C++ is used to manage runtime errors gracefully. It uses three main keywords: `try`, `throw`, and `catch`. Code that might throw an error is placed in the `try` block. If an exception occurs, `throw` sends it to the corresponding `catch` block. Example:

```
cpp
CopyEdit
try {
  throw 10;
} catch (int e) {
  cout << "Error: " << e;
}
```

C++ supports multiple catch blocks for different exception types. Exception handling separates error-handling code from regular logic. It helps avoid program crashes and provides meaningful error messages. You can also throw user-defined types. The `catch(...)` block catches all types. Exception handling supports clean resource management. It is essential in building reliable software.

25. What is the syntax of try, catch, and throw in C++?

C++ uses the following syntax for exception handling:

```
cpp
CopyEdit
try {
  // Code that may throw exception
  throw exception_value;
} catch (Type1 arg) {
  // Handle exception of Type1
} catch (Type2 arg) {
  // Handle exception of Type2
}
```

The `try` block encloses risky code. The `throw` keyword generates an exception. The `catch` block handles the thrown exception. You can have multiple catch blocks for different types. A catch-all block `catch(...)` handles any type of exception. Exceptions can be basic types or class objects. Exception handling enables robust error management. It avoids program termination during unexpected errors. Catch blocks should follow the try block immediately.

26. What is the need for exception handling in C++?

Exception handling allows you to deal with runtime errors in a controlled way. It prevents program crashes by catching unexpected events. Without it, errors like divide-by-zero or file-not-found would terminate the program. It separates error-handling logic from core logic. It enables graceful failure and better debugging. Exceptions help maintain program flow and stability. They support custom messages for users and logs. They allow developers to anticipate and handle edge cases. Exception handling improves software robustness. It reduces the chances of undefined behavior. It's essential in critical applications like finance, healthcare, etc. It helps developers build fault-tolerant applications.

27. What are the types of exceptions in C++?

C++ exceptions can be broadly classified as:

1. **Standard exceptions** – Defined in the `<exception>` header like `std::bad_alloc`, `std::out_of_range`, etc.
2. **User-defined exceptions** – Custom exception classes created by the programmer.
3. **Synchronous exceptions** – Occur due to logical errors (e.g., divide by zero).
4. **Asynchronous exceptions** – External to the program (e.g., hardware failure, signals). C++ allows throwing any type (int, char, string, class). However, it's best practice to throw objects derived from `std::exception`. The hierarchy provides functions like `what()` for error messages. Using well-defined exception types improves clarity. A `catch(...)` block handles all types.

28. Explain the concept of multiple catch blocks.

Multiple catch blocks allow handling different types of exceptions differently. Each `catch` block handles a specific type. Syntax:

```
cpp
CopyEdit
try {
  // risky code
} catch (int e) {
  // handle int exception
} catch (char e) {
  // handle char exception
}
```

Only one matching catch block executes. Catch blocks are checked top to bottom. If no match is found, the exception propagates or leads to a crash. This approach provides precise error handling. You can also add a general catch-all using `catch(...)`. Ordering catch blocks

properly avoids ambiguity. It's useful when working with different exception types from templates or class hierarchies. It improves control and makes programs more resilient.

29. What is the use of catch(...) block?

The `catch(...)` block is a catch-all handler. It catches any type of exception thrown from the `try` block. It is useful when the exception type is unknown or for generic error messages. Example:

```
cpp
CopyEdit
try {
  throw 10.5;
} catch (...) {
  cout << "Unknown exception caught.";
}
```

It should be placed after all specific catch blocks. It provides a safety net when no matching handler exists. It helps avoid unhandled exceptions. However, it does not allow you to access the thrown value. Use it for logging or cleanup. It's useful in library code or critical systems to ensure stability.

30. How are exceptions handled in C++ using classes?

C++ allows you to create custom exception classes, usually derived from `std::exception`. This enables more descriptive and structured error handling. Example:

```
cpp
CopyEdit
class MyException : public std::exception {
public:
  const char* what() const noexcept override {
    return "Custom exception occurred";
  }
};
```

You can then throw and catch this object:

```
cpp
CopyEdit
throw MyException();
```

Using classes makes exceptions more meaningful. You can pass extra data through members. Inheritance allows grouping exception types. The `what()` function returns a descriptive message. Exception classes support better debugging and code organization. They are essential in robust, large-scale applications.

31. What is the Standard Template Library (STL)?

The Standard Template Library (STL) is a powerful library in C++ that provides generic classes and functions. It includes commonly used data structures and algorithms. STL is based on templates, so it works with any data type. It includes:

- **Containers** (like `vector`, `list`, `map`, `set`)
- **Algorithms** (like `sort()`, `find()`, `count()`)
- **Iterators** (like pointers to traverse containers)
  STL promotes code reusability and efficiency. It saves time by offering well-tested implementations. STL is type-safe and supports generic programming. It works seamlessly with user-defined types. Containers hold data, algorithms process data, and iterators connect the two. STL reduces development time and boosts performance in C++ programs.

32. What are the major components of STL?

STL has three major components:

1. **Containers** – Store collections of objects (e.g., `vector`, `list`, `deque`, `map`, `set`).
2. **Algorithms** – Perform operations like searching, sorting, merging, etc. (e.g., `sort()`, `find()`).
3. **Iterators** – Point to elements in containers and allow traversal (like pointers).
   Other important parts:

- **Functors (function objects)** – Objects that can act like functions.
- **Allocators** – Handle memory management for STL containers.
  STL supports a consistent interface across containers. Algorithms work with all containers via iterators. It is efficient, flexible, and reduces coding effort significantly.

33. What are containers in STL?

Containers are objects that store multiple elements of the same type. STL containers are of three types:

1. **Sequence Containers** – Linear storage (`vector`, `list`, `deque`).
2. **Associative Containers** – Key-value pairs (`set`, `map`, `multimap`, `multiset`).
3. **Unordered Containers** – Hash-based storage (`unordered_map`, `unordered_set`).
   Containers provide member functions for insertion, deletion, access, and traversal. They are templated, allowing flexibility in data types. Example:

```
cpp
CopyEdit
vector<int> v = {1, 2, 3};
```

Each container has its own strengths. For example, `vector` is fast for random access, while `list` is good for frequent insertions/deletions.

34. What is a vector in STL?

A `vector` is a dynamic array provided by STL. It stores elements in contiguous memory like arrays. Key features:

- Automatically resizes when elements are added/removed.
- Allows random access via index (`v[2]`).
- Offers functions like `push_back()`, `pop_back()`, `insert()`, `erase()`.
- Provides iterators for traversal.
  Example:

```
cpp
CopyEdit
vector<int> v;
v.push_back(10);
```

`vector` is efficient for insertions at the end. It is part of the `<vector>` header. Ideal when the size of data is unknown at compile time. Vectors are widely used in competitive programming and projects.

35. What is a list in STL?

A `list` in STL is a **doubly linked list** container. Unlike vectors, it allows fast insertions and deletions anywhere in the list. Key features:

- Not stored in contiguous memory.
- Provides bidirectional iterators.
- Slower random access (`no v[i]`).

- Functions: `push_back()`, `push_front()`, `insert()`, `remove()`, `sort()`.
  Example:

```
cpp
CopyEdit
list<int> l = {1, 2, 3};
l.push_front(0);
```

Lists are efficient for frequent modifications (insertions/deletions). Ideal when order changes often. It's included in the `<list>` header.

36. What is a map in STL?

A `map` is an associative container that stores key-value pairs in sorted order. Each key is unique. It uses a self-balancing binary search tree internally. Features:

- Syntax: `map<Key, Value>`.
- Elements are sorted by key.
- Fast lookup, insertion, and deletion (`O(log n)`).
- Access via key: `m[key]`.
  Example:

```
cpp
CopyEdit
map<int, string> m;
m[1] = "one";
```

Maps are ideal for dictionary-like data. They are in `<map>` header. To allow duplicate keys, use `multimap`. For unsorted versions, use `unordered_map`.

37. What is the difference between map and unordered_map?

Feature	`map`	`unordered_map`
Storage Order	Sorted by keys	Unordered (hash-based)
Implementation	Red-black tree	Hash table
Access Time	O(log n)	Average O(1), Worst O(n)
Key Uniqueness	Keys must be unique	Keys must be unique

Feature	map	unordered_map
Header	<map>	<unordered_map>
Use map when you need ordered data. Use unordered_map for faster access when order doesn't matter. map consumes more memory but is predictable. unordered_map is faster for lookups.		

38. What is a stack in STL?

A stack is a container adapter in STL that follows **Last In First Out (LIFO)**. You can only access the top element. Key functions:

- push() – add to top
- pop() – remove top
- top() – access top
- empty() – check if empty
- size() – number of elements
  Example:

```
cpp
CopyEdit
stack<int> s;
s.push(10);
```

stack is implemented using deque by default. It's used in function calls, expression evaluation, undo features, etc. It's declared using <stack> header.

39. What is a queue in STL?

A queue is a FIFO (First In First Out) container adapter. Elements are inserted at the back and removed from the front. Key operations:

- push() – add to rear
- pop() – remove from front
- front() – access front
- back() – access rear
- empty() – checks if queue is empty
  Example:

```
cpp
CopyEdit
```

```
queue<int> q;
q.push(5);
```

It’s ideal for scheduling tasks, buffering, etc. It's built on top of `deque`. Declared using `<queue>` header. Unlike `vector`, you cannot access elements by index.

40. What is a priority_queue in STL?

A `priority_queue` is a max-heap structure. The largest element is always on top. It's useful when you need quick access to the maximum (or minimum) value. Key operations:

- `push()` – inserts element
- `pop()` – removes top (highest priority)
- `top()` – accesses the top
- `empty()` – checks if empty
  Example:

```
cpp
CopyEdit
priority_queue<int> pq;
pq.push(10);
```

By default, it is a max-heap. For min-heap, use a custom comparator or `greater<int>`. Useful in algorithms like Dijkstra’s or Huffman encoding. Declared using `<queue>` header.

****41. What are file operations in C++?**

File operations** in C++ involve reading from and writing to files using file streams. The `<fstream>` header provides three main classes:

- `ifstream` for reading (input file stream),
- `ofstream` for writing (output file stream),
- `fstream` for both input and output.
  Basic file operations include:
- Opening a file using `.open()`,
- Reading/writing using <<, >>, or `.get()` / `.put()`,
- Closing a file using `.close()`.
  Example:

```
cpp
```

```
CopyEdit
ofstream fout("data.txt");
fout << "Hello!";
fout.close();
```

Error checking using `.fail()` is important. File modes like `ios::in`, `ios::out`, and `ios::app` control access. File handling allows data persistence across program runs.

42. What are the types of file streams in C++?

C++ supports three main types of file streams defined in the `<fstream>` header:

1. **ifstream (input file stream)** – For reading from files.
2. **ofstream (output file stream)** – For writing to files.
3. **fstream (file stream)** – For both reading and writing.
   These classes inherit from `istream` and `ostream`. Depending on the file mode (`ios::in`, `ios::out`, etc.), the file stream is opened in different ways. For example:

```
cpp
CopyEdit
ifstream fin("data.txt");
ofstream fout("log.txt");
```

Each stream offers functions like `.open()`, `.close()`, `.eof()`, `.get()`, `.put()`. Proper stream selection is important based on the required operation.

43. How do you open and close files in C++?

To **open a file**, use `.open()` or provide the filename in the constructor:

```
cpp
CopyEdit
ifstream fin;
fin.open("data.txt");
```

Or:

```
cpp
CopyEdit
ofstream fout("data.txt");
```

To **close a file**, use:

```
cpp
CopyEdit
fin.close();
fout.close();
```

Files can also be opened in specific modes like:

- `ios::in`, `ios::out`, `ios::app`, `ios::binary`.
  Always check if the file is opened successfully using `.is_open()` or `.fail()`.
  Closing a file flushes the buffer and releases resources. Not closing may lead to data loss or corruption.

44. What are the different file modes in C++?

File modes determine how a file is accessed. Common modes include:

- `ios::in` – Open for reading.
- `ios::out` – Open for writing.
- `ios::app` – Append at end.
- `ios::ate` – Go to end on open.
- `ios::trunc` – Truncate if file exists.
- `ios::binary` – Open in binary mode.
  Modes can be combined using `|`. Example:

```
cpp
CopyEdit
fstream file("log.txt", ios::in | ios::out);
```

Choosing the correct mode ensures proper access and prevents unintended data loss. For example, `ios::trunc` deletes existing data unless `ios::app` is used.

45. What is the difference between text and binary files in C++?

Feature	Text File	Binary File
Format	Human-readable (ASCII)	Machine-readable (raw bytes)
Size	Larger (includes formatting)	Smaller
Readability	Can be opened in text editor	Requires specific

Feature	Text File	Binary File
		programs
Speed	Slower	Faster for structured data
Usage	Logs, config files	Images, videos, compiled data

Binary files are opened with `ios::binary`. Text files can be opened normally. Binary file operations use `.read()` and `.write()` for block access.

46. How to write data into a file in C++?

To write data:

1. Include `<fstream>`.
2. Use `ofstream` or `fstream` with `ios::out`.
   Example:

```
cpp
CopyEdit
ofstream fout("data.txt");
fout << "Welcome to C++ file handling!";
fout.close();
```

You can also use `.put()` for character-by-character writing. Make sure to check if the file opens using `.fail()`. Writing can be done in binary using `.write()` for structured data. Always close the file after writing. You can append data using `ios::app`.

47. How to read data from a file in C++?

To read:

1. Use `ifstream` or `fstream` with `ios::in`.
2. Open the file and use >> or `.getline()` to read data.
   Example:

```
cpp
CopyEdit
ifstream fin("data.txt");
string line;
while (getline(fin, line)) {
  cout << line << endl;
```

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

You can also use `.get()` to read character by character. Always check `.eof()` or `.fail()` to detect file end or read failure. Reading binary data is done using `.read()` method.

48. What is the use of seekg() and seekp()?

- `seekg()` – Sets the get pointer (input position).
- `seekp()` – Sets the put pointer (output position).
  They allow moving to specific positions in a file. Useful for random access in binary files.
  Syntax:

```
cpp
CopyEdit
file.seekg(position);
file.seekp(position);
```

Offsets can be combined with:

- `ios::beg` – Beginning of file
- `ios::cur` – Current position
- `ios::end` – End of file
  Example:

```
cpp
CopyEdit
fin.seekg(10, ios::beg); // Move to 11th byte
```

These functions allow efficient file navigation.

49. What is the difference between get(), getline(), and read()?

- `get()` – Reads a single character.
- `getline()` – Reads a line of text into a string.
- `read()` – Reads a block of binary data.
  Examples:

```
cpp
CopyEdit
char c;
```

```
fin.get(c);

char line[100];
fin.getline(line, 100);

fin.read((char*)&obj, sizeof(obj));
```

Use `get()` for char-by-char input, `getline()` for reading strings until newline, and `read()` for binary file operations. Each serves different use cases based on input type and file mode.

50. How can files be used to store and retrieve class objects in C++?

To store class objects:

1. Define the class and object.
2. Use `ofstream` or `fstream` with `ios::binary | ios::out`.
3. Use `.write()` to write the object.
   Example:

```
cpp
CopyEdit
class Student { ... };
Student s;
ofstream fout("data.dat", ios::binary);
fout.write((char*)&s, sizeof(s));
```

To retrieve:

```
cpp
CopyEdit
ifstream fin("data.dat", ios::binary);
fin.read((char*)&s, sizeof(s));
```

Ensure there are no virtual functions or dynamic memory in the class for binary storage. This approach allows persistent storage of structured data.