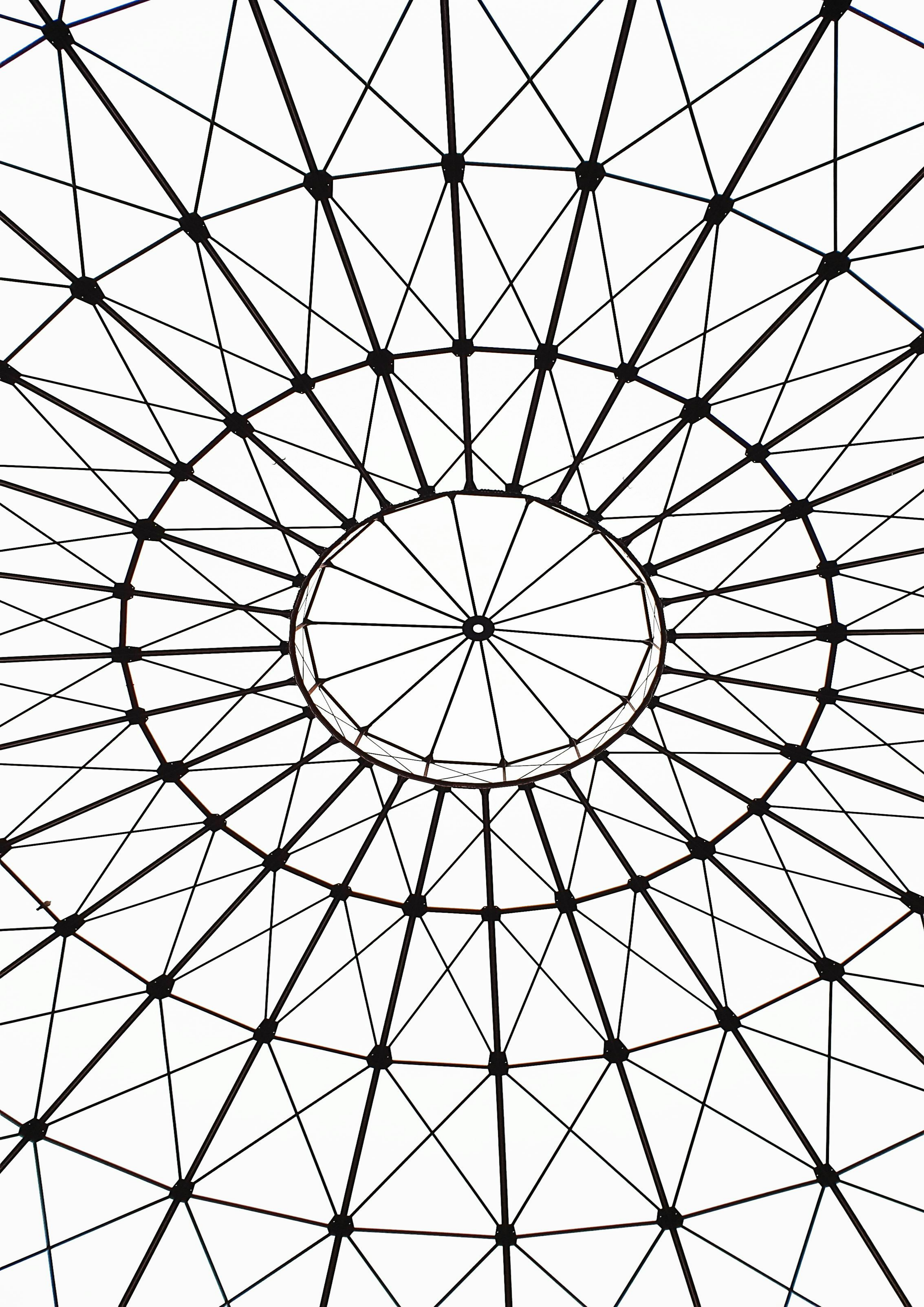

C++ Question Bank

Anchit Mulye

July 18, 2024

C++ Question Bank

Anchit Mulye

July 18, 2024

Contents

1 Basic of C++

1.1 What are the key features of C++?

- **Object-Oriented:** Supports object-oriented programming paradigms like classes, inheritance, polymorphism, and encapsulation.
- **Rich Standard Library:** Provides a comprehensive library that includes algorithms, containers, and functions for various tasks.
- **Performance:** Offers low-level control over hardware and memory, making it suitable for system-level programming and performance-critical applications.
- **Portability:** Designed to be platform-independent, with compilers available for most operating systems.
- **Compatibility:** Supports both procedural and object-oriented programming, enabling developers to choose the appropriate paradigm based on project requirements.
- **Memory Management:** Supports both static and dynamic memory allocations.

1.2 Explain the structure of a C++ program.

A typical C++ program consists of the following components:

- **Preprocessor Directives:** These are instructions to the preprocessor, which processes the source code before compilation. Directives start with `#` and include commands like `#include` for including header files.
- **Namespace Declaration:** Namespaces help avoid name collisions and organize code. The `namespace` keyword declares a scope where identifiers can exist.
- **Main Function:** Every C++ program must have a `main()` function, which serves as the entry point. Execution of the program begins from `main()`.
- **Variable Declarations:** Variables are declared to store data. They must be declared before they are used, specifying the data type.
- **Statements and Expressions:** C++ statements perform actions, and expressions compute values. These form the core logic of the program.
- **Functions:** Functions encapsulate reusable code. They are declared with a return type, name, parameters (optional), and a function body.
- **Classes and Objects (Optional):** C++ supports object-oriented programming. Classes define data structures and methods, while objects are instances of classes.
- **Comments:** Comments improve code readability and understanding. They can be single-line (`//`) or multi-line (`/* */`).
- **Headers and Libraries:** Header files (`#include`) provide declarations needed for compilation. Libraries extend functionality beyond standard C++.
- **End of Program:** The `return` statement in `main()` (if required) signifies the end of the program execution, returning an exit status to the operating system.

1.3 Explain the steps from source code to executable.

- **Source Code (`.cpp` files):** This is where you write your C++ program using a text editor or an Integrated Development Environment (IDE).
- **Preprocessing:**
  - **Tool:** Preprocessor

  - **Description:** Before compilation begins, the preprocessor handles directives starting with `#`, such as `#include` and `#define`. It includes header files and expands macros to generate an intermediary file (`.i`).

- **Compilation:**
  - **Tool: Compiler (`g++`, `clang++`, etc.)**
  - **Description:** The compiler translates the preprocessed source code (`.i` file) into assembly language code (`.s` file). It checks syntax, semantics, and type correctness during this phase.

- **Assembly:**
  - **Tool: Assembler (as)**
  - **Description:** The assembler converts the assembly code (`.s` file) into machine code or object code (`.o` file). Each line of assembly code corresponds directly to a machine instruction.

- **Linking:**
  - **Tool: Linker (ld)**
  - **Description:** The linker combines multiple object files (`.o` files) generated from the compilation step along with necessary libraries (like the C++ Standard Library) to create a single executable file. It resolves external references between files, assigns addresses to variables and functions, and generates the final executable (`.exe` on Windows, without extension on Unix-like systems).

- **Loading (Optional):**
  - **Tool:** Loader
  - **Description:** On some operating systems, an additional loading step may occur where the executable is loaded into memory for execution.

- **Execution:**
  - **Tool:** Operating System
  - **Description:** Finally, the operating system loads the executable into memory and starts executing the `main()` function, initiating the program.

Notes

- **Header Files:** These are included in the preprocessing step (`#include`) and provide declarations for functions, classes, and constants used in the program.
- **Libraries:** Besides standard libraries, you can link external libraries (static `.lib` or dynamic `.dll/.so`) to extend functionality.
- **Debugging Symbols:** Debug versions of executables may include symbols (`pdb` on Windows, `dSYM` on macOS) to aid in debugging.

1.4 What are data types in C++?

- **Primitive Data Types:** Integer, Float, Character, Boolean, Void
- **Derived Data Types:** Pointers, References, Functions, Arrays
- **User Defined Data Types:** Classes, Structures, Unions, Enumerations

1.5 What is call by value and call by reference?

- **Call by Value:** a copy of the actual argument is passed to the function.

Call by Value

```
void modifyValue(int value) {
    value = 100; // Change local copy
}

int main() {
    int a = 10;
    modifyValue(a); // Pass by value
    std::cout << "Value after: " << a << std::endl; // a remains 10
    return 0;
}
```

- **Call by Reference:** a reference to the actual argument is passed to the function.

Call by Reference

```
void modifyValue(int &value) {
    value = 100; // Change original value
}

int main() {
    int a = 10;
    modifyValue(a); // Pass by reference
    std::cout << "Value after: " << a << std::endl; // a becomes 100
    return 0;
}
```

- **Call by Pointer:** similar to call by reference, but with pointers.

Call by Pointer

```
void modifyValue(int *value) {
    *value = 100; // Change original value via pointer
}

int main() {
    int a = 10;
    modifyValue(&a); // Pass by pointer
    std::cout << "Value after: " << a << std::endl; // a becomes 100
    return 0;
}
```

1.6 What are differences between pass by value and pass by reference?

	Pass by Value	**Pass by Reference**
Memory	Higher memory usage due to copies	Lower memory usage, only references are passed
Performance	Potentially slower due to copying	Generally faster, no copying involved
Effects	No effect on original data	Modifies the original data
Safety	Safer, no unintended side effects	Riskier, careful handling required to avoid issues

1.7 What are differences between reference and pointer?

	Reference	**Pointer**
Syntax	`int &ref = variable`	`int *ptr = &variable`
Initialization	Must be initialized and can not be null	Can be declared without initialization and can be null
Reassignment	Reference cannot be changed	Can be reassigned to point to a different variable
Dereferencing	Automatically dereferenced	Explicit dereferencing using `*int value = *ptr`
Address	Does not have its own address	Has its own address and can store other addresses
Use Cases	Function parameters and return values; Operator overloading	Dynamic memory allocation; Low level memory manipulation

1.8 What are differences between prefix and postfix?

- **Prefix Increment/Decrement**
  - **Syntax:** `++variable` or `--variable`
  - **Operation:** The operator is applied before the value is used in the expression.

Prefix

```
int a = 5;
int b = ++a; // a is incremented to 6, then b is assigned the value 6
```

- **Postfix Increment/Decrement**
  - **Syntax:** `variable++` or `variable--`
  - **Operation:** The operator is applied after the value is used in the expression.

Postfix

```
int a = 5;
int b = a++; // b is assigned the value 5, then a is incremented to 6
```

1.9 What are operations on pointers?

- Declaration and Initialization
- Assignment
- Dereferencing
- Address-of Operator

- Pointer Arithmetic
- Comparison
- Null Pointers

Operations on Pointers

```cpp
#include <iostream>

int main() {
    int x = 10;
    int y = 20;
    int arr[] = {1, 2, 3, 4, 5};

    // Declaration and Initialization
    int *ptr1 = &x;
    int *ptr2 = &y;

    // Assignment
    ptr1 = ptr2; // ptr1 now points to y

    // Dereferencing
    std::cout << "Value at ptr1: " << *ptr1 << std::endl; // Output: 20
    *ptr1 = 30; // Changing the value of y to 30

    // Address-of Operator
    int *ptr3 = &x;

    // Pointer Arithmetic
    int *ptrArr = arr;
    std::cout << "First element: " << *ptrArr << std::endl; // Output: 1
    ptrArr++;
    std::cout << "Second element: " << *ptrArr << std::endl; // Output: 2

    // Comparison
    if (ptr1 != ptr2) {
        std::cout << "Pointers are different" << std::endl;
    } else {
        std::cout << "Pointers are the same" << std::endl; // Printed
    }

    // Null Pointer
    int *ptrNull = nullptr;
    if (ptrNull == nullptr) {
        std::cout << "ptrNull is a null pointer" << std::endl; // Printed
    }

    return 0;
}
```

1.10 What are different storage classes?

In C++, storage classes define the scope, visibility, and lifetime of variables and/or functions within a C++ program. There are four storage classes in C++:

- `auto`:
  - Default storage class for local variables.

- The variable is only visible within the block where it is defined.

- `register`:
  - Suggests that the variable be stored in a CPU register instead of RAM.
  - Used for variables that require fast access.
- `static`:
  - The variable retains its value between multiple function calls.
  - For global variables, the variable is limited to the file scope.
- `extern`:
  - Used to declare a global variable or function in another file.
  - Indicates that the variable is defined elsewhere.
- `mutable`:
  - Allows a member of an object to be modified even if the object is const.
  - Used primarily with class member variables.

1.11 What are access modifiers?

Access modifiers in C++ are keywords that set the access level or visibility of class members. There are three primary access modifiers:

- `public`:
  - Members are accessible from outside the class.
  - Used for functions and variables that need to be accessed by other classes.
- `protected`:
  - Members are accessible within the class and by derived class instances.
  - Used to allow derived classes to access the base class members while hiding them from the rest of the program.
- `private`:
  - Members are only accessible within the class itself.
  - Used to hide data and functions from outside the class.

1.12 What are inline functions

Inline functions are functions defined with the `inline` keyword, which suggests to the compiler to insert the function's code at the point of call rather than performing a traditional function call. This can improve performance by avoiding the overhead of a function call, especially for small, frequently called functions.

- Declared using the `inline` keyword.
- Useful for small functions that are called frequently.
- Can lead to faster execution but may increase the size of the binary.

Inline Function

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

1.13 What are differences between `static` and `volatile`?

- `static`:
  - The `static` keyword in C++ is used to declare variables that retain their value between function calls.
  - It can be applied to local variables, global variables, and class members.
  - A `static` variable inside a function retains its value between invocations of the function.
  - A `static` global variable or function has internal linkage, meaning it is only accessible within the file in which it is declared.
  - A `static` class member is shared among all instances of the class.
- `volatile`:
  - The `volatile` keyword is used to indicate that a variable's value may be changed at any time by something outside the control of the code section in which it appears.
  - This prevents the compiler from optimizing the code in ways that assume the variable's value is not changing unexpectedly.
  - It is commonly used in embedded systems, multithreading, and signal handling.

1.14 Explain the concept of RAII (Resource Acquisition Is Initialization).

Resource Acquisition Is Initialization (RAII) is a programming idiom used to manage resources such as memory, file handles, and network connections. It ties resource management to the lifetime of objects, ensuring that resources are properly released when objects go out of scope.

- Resources are acquired and released by constructors and destructors, respectively.
- Ensures exception safety by automatically releasing resources in the destructor.
- Commonly used with smart pointers, file streams, and lock guards.

1.15 What is exception handling in C++? Why is it used?

Exception handling in C++ provides a way to react to exceptional circumstances (errors) in a program. It is used to manage errors and exceptional conditions in a controlled way, allowing the program to continue running or gracefully terminate.

- Helps in separating error-handling code from regular code.
- Enables handling of runtime errors, preventing abrupt program termination.

1.16 Explain the `try`, `catch`, and `throw` blocks in C++.

- `try` block:
  - Contains code that might throw an exception.
  - If an exception is thrown, the control is transferred to the corresponding `catch` block.
- `catch` block:
  - Used to handle exceptions thrown by the `try` block.
  - Multiple `catch` blocks can be used to handle different types of exceptions.
- `throw` statement:
  - Used to throw an exception.
  - Can be used to throw exceptions of any type.

Exception Handling

```
    try {
        // Code that may throw an exception
        throw std::runtime_error("Error occurred");
    } catch (const std::runtime_error &e) {
        // Handle runtime error
        std::cerr << e.what() << std::endl;
    } catch (...) {
        // Handle any exception
        std::cerr << "Unknown error" << std::endl;
    }
```

1.17 What is the difference between runtime error and compile time error?

- **Runtime Error**:
  - Occurs during the execution of a program.
  - Examples include division by zero, null pointer dereference, and out-of-bounds array access.
  - Usually leads to program termination or undefined behavior.
- **Compile Time Error**:
  - Detected by the compiler during the compilation of the program.
  - Examples include syntax errors, type mismatches, and undeclared variables.
  - Prevents the program from compiling successfully.

1.18 How do you create a custom exception in C++?

Creating a custom exception in C++ involves defining a new class that inherits from the `std::exception` class or any of its derived classes.

Custom Exception

```
#include <exception>

class MyCustomException : public std::exception {
public:
    const char* what() const noexcept override {
        return "My custom exception occurred";
    }
};

int main() {
    try {
        throw MyCustomException();
    } catch (const MyCustomException &e) {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

1.19 How do you open and close a file in C++?

To open and close a file in C++, you use the file stream classes from the `<fstream>` library.

File Open and Close

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outFile("example.txt"); // Open file for writing
    if (outFile.is_open()) {
        outFile << "Writing to file\n";
        outFile.close(); // Close the file
    }

    std::ifstream inFile("example.txt"); // Open file for reading
    if (inFile.is_open()) {
        std::string line;
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }
        inFile.close(); // Close the file
    }

    return 0;
}
```

1.20 Explain the difference between `ifstream`, `ofstream`, and `fstream` in C++.

- `ifstream`:
  - Input file stream.
  - Used for reading from files.
- `ofstream`:
  - Output file stream.
  - Used for writing to files.
- `fstream`:
  - File stream.
  - Used for both reading from and writing to files.

1.21 How do you check if a file exists in C++?

To check if a file exists in C++, you can use the `std::ifstream` class.

File Exists

```cpp
#include <fstream>
#include <iostream>

bool fileExists(const std::string &filename) {
    std::ifstream file(filename);
    return file.good();
}

int main() {
    std::string filename = "example.txt";
    if (fileExists(filename)) {
        std::cout << "File exists.\n";
    } else {
        std::cout << "File does not exist.\n";
    }
    return 0;
}
```

1.22 What is the difference between `const char*`, `char* const`, and `const char* const`.

- `const char*`:
  - Pointer to a constant character.
  - The character pointed to by the pointer cannot be modified.
  - The pointer itself can be modified to point to another character.
  - Example: `const char* ptr = "Hello";`
- `char* const`:
  - Constant pointer to a character.
  - The pointer itself cannot be modified after initialization.
  - The character pointed to by the pointer can be modified.
  - Example: `char c = 'A'; char* const ptr = &c;`
- `const char* const`:
  - Constant pointer to a constant character.
  - Neither the pointer nor the character it points to can be modified.
  - Example: `const char* const ptr = "Hello";`

1.23 How to return a string from a function in C++?

- **Method 1: Returning a String Object**

As an Object

```
#include <iostream>
#include <string>

std::string getString() {
    std::string message = "Hello, World!";
    return message;
}

int main() {
    std::string result = getString();
    std::cout << "Returned string: " << result << std::endl;
    return 0;
}
```

- **Method 2: Returning a String by Reference**

As a Reference

```
#include <iostream>
#include <string>

const std::string& getString() {
    static std::string message = "Hello, World!";
    return message;
}

int main() {
    const std::string& result = getString();
    std::cout << "Returned string: " << result << std::endl;
    return 0;
}
```

1.24 What are namespaces in C++? Explain how they work.

Namespaces are a feature in C++ that allows you to group a set of related classes, functions, and variables under a single name. This helps to avoid name collisions in large projects where multiple libraries might have functions or variables with the same names. By encapsulating these entities within a namespace, you ensure that their names are unique within the scope of that namespace. Here is a basic example to illustrate how namespaces work in C++:

Namespace

```
#include <iostream>

namespace MyNamespace {
    void display() {
        std::cout << "Hello from MyNamespace!" << std::endl;
    }
}

int main() {
    MyNamespace::display();
    return 0;
}
```

- **Declaration:** The keyword `namespace` is used to declare a namespace. In this example, `MyNamespace` is a namespace that contains the `display` function.

- **Scope Resolution Operator:** To call a function or access a variable within a namespace, you use the scope resolution operator `::`. In `MyNamespace::display()`, the `::` operator is used to access the `display` function within `MyNamespace`.

Types of Namespaces

- Regular Namespaces
- Anonymous Namespaces
- Nested Namespaces
- Inline Namespaces (C++11 and later)
- Standard Namespace

1.25 Explain different agruments of `main()`.

In C++, the `main()` function can have different argument signatures depending on the platform and compiler. Common signatures include:

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

- `argc`: Argument count, which is the number of command-line arguments passed to the program.
- `argv`: Argument vector, an array of strings containing the command-line arguments.
- `envp`: Environment variables passed to the program (in some implementations).

`main()` Function

```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "Argument count: " << argc << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << "Argument " << i << ": " << argv[i] << std::endl;
    }
    return 0;
}
```

1.26 What is difference between shallow copy and deep copy?

In C++, shallow copy and deep copy are ways of copying objects:

- **Shallow Copy**: This involves copying the values of data members from one object to another. If the object contains pointers to dynamically allocated memory, only the pointers are copied, not the data they point to. Both objects then point to the same memory locations, which can lead to issues if one object modifies the data.
- **Deep Copy**: This involves copying all the data pointed to by the pointers as well. It allocates new memory for the copied object and copies the values pointed to by the pointers. This way, each object has its own copy of the data, preventing unintended side effects due to shared data.

Shallow Copy

```
#include <iostream>

class ShallowCopyExample {
private:
    int *data;
public:
    ShallowCopyExample(int d) {
        data = new int(d);
    }
    // Shallow copy constructor
    ShallowCopyExample(const ShallowCopyExample &other) {
        data = other.data; // Only copies the pointer
    }
};

int main() {
    ShallowCopyExample obj1(10);
    ShallowCopyExample obj2 = obj1; // Shallow copy happens here
    // obj1 and obj2 now share the same data pointer
    return 0;
}
```

Deep Copy

```
#include <iostream>

class DeepCopyExample {
private:
    int *data;
public:
    DeepCopyExample(int d) {
        data = new int(d);
    }
    // Deep copy constructor
    DeepCopyExample(const DeepCopyExample &other) {
        data = new int(*other.data); // Allocates new memory and copies data
    }
    ~DeepCopyExample() {
        delete data;
    }
};

int main() {
    DeepCopyExample obj1(10);
    DeepCopyExample obj2 = obj1; // Deep copy happens here
    // obj1 and obj2 have separate copies of data
    return 0;
}
```

1.27 How function calls works in C++?

In C++, function calls involve several steps:

1. **Function Definition**: The function is defined, specifying its return type, name, parameters, and body.

2. **Function Call**: When a function is called, control transfers to the function's entry point (its address).
3. **Function Execution**: Inside the function, the statements within its body are executed in sequence.
4. **Return Value**: If the function has a return type other than `void`, it returns a value back to the caller.
5. **Return to Caller**: Control returns to the caller, continuing execution from where the function was called.

2 C vs C++

2.1 Differentiate between C and C++

Aspect	C	C++
Paradigm	Procedural	Multi-paradigm (procedural, OOP, generic)
Features	Simple, minimalistic	Object-oriented, templates, exceptions
Standard Libraries	Standard C Library	Standard Template Library (STL)
Compatibility	More portable	Includes C features; additional features may not be universally supported
Execution Speed	Often slightly faster	Higher-level abstractions may incur overhead
Memory Management	Manual (malloc, free)	Manual and automatic (RAII, smart pointers)
Usage	System programming, embedded systems	Larger-scale software, GUI applications

C

```c
#include <stdio.h>

int main() {
    printf("Hello, C!\n");
    return 0;
}
```

C++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, C++!" << std::endl;
    return 0;
}
```

2.2 What is the difference between `cout` and `printf` in C++?

Aspect	`cout`	`printf`
Usage	Part of C++ Standard Library (`<iostream>`)	Part of C Standard Library (`<stdio.h>`)
Output	Formatted output to standard output	Formatted output to standard output
Syntax	Uses `<<` insertion operator	Uses format specifiers (`%`)
Type Safety	Type-safe; checks types at compile-time	Format specifiers must match argument types
C++ Compatibility	Native to C++, supports namespaces, exceptions	From C, compatible with C++ but lacks some C++ features
Performance	Generally slower due to object-oriented nature	Generally faster for large-scale formatted output
Buffer	It is fully buffered	It is line buffered
Implementation	It is an object of `ostream` class	It is a function

2.3 What is the difference between `new` and `malloc()` in C++?

Aspect	`new`	`malloc()`
Language	C++	C
Header File	`<new>` (automatically included in C++)	`<stdlib.h>`
Return Type	Returns pointer to allocated memory of specific type	Returns `void*` (generic pointer)
Initialization	Calls constructor for objects (if applicable)	Memory allocated but no initialization
Type Safety	Type-safe; allocates memory for specific data type	Not type-safe; requires casting for proper use

new

```
int* ptr = new int;
MyClass* obj = new MyClass();

delete ptr;
delete obj;
```

malloc

```
int* ptr = (int*)malloc(sizeof(int)); // No equivalent for constructor invocation

free(ptr);
free(obj); // assuming obj is a pointer
```

2.4 How do you link a C++ program to C functions?

In C++, functions and variables are typically subject to name mangling—a process that encodes function signatures into unique names to support function overloading and type-safe linkage. However, when interfacing with C functions, which do not undergo name mangling, the `extern "C"` linkage specification is used to ensure compatibility and proper linking.

- **Name Mangling:**
  - C++ compilers encode function signatures into mangled names to handle function overloading and maintain type safety.
- **C Functions:**
  - C functions do not undergo name mangling and have straightforward function names.
- `extern "C"` **Linkage Specification:**
  - By using `extern "C"` around C function declarations in C++ code, you inform the compiler to disable name mangling for these functions. This allows the linker to correctly resolve and link function calls to their corresponding C function definitions.
- **Compilation and Linking:**
  - When compiling a C++ program that includes headers with `extern "C"` declarations, the compiler ensures that function names are not mangled.
  - This enables the linker to correctly match calls from C++ code to the unmangled names of C functions defined elsewhere.

C Function to C++

```
// C header file example.h
#ifdef __cplusplus
extern "C" {
#endif

void c_function(); // Declaration of C function

#ifdef __cplusplus
}
#endif
```

2.5 Differences between struct in C and C++.

Aspect	C	C++
Default Access Specifier	No default; members default to `public`	Default access specifier is `public`
Keyword Usage	Requires `struct` keyword	Does not require `struct` keyword
Member Functions	Cannot have member functions	Can have member functions (methods)
Inheritance	Does not support inheritance	Supports single and multiple inheritance
Constructor	No constructors	Can have constructors and destructors
Access Control	No access control (all members public by default)	Supports access specifiers (`public`, `private`, `protected`)

Notes

Keyword Usage: In C, you must use the `struct` keyword when declaring a struct variable (`struct Point p;`). In C++, the `struct` keyword is optional (`Point p;`).

2.6 What does extern ”C” int func(int *, Foo) accomplish?

In C++, the `extern "C"` declaration is used to specify that the function `func` should use C-style linkage rather than C++ linkage. This affects how the function's name is represented (or mangled) by the compiler, which is crucial for interoperability between C++ code and code written in other languages like C.

- **Name Mangling:**
  - C++ compilers often ”mangle” function names, meaning they encode additional information about a function's parameters and return type into the function's name. This allows for function overloading and is used to implement features like namespaces.
- **C-style Linkage:**
  - When you declare a function with `extern "C"`, you're telling the compiler to use C-style linkage for that function. This linkage does not perform name mangling, resulting in a plain function name without any additional encoding.
- **Purpose:**
  - The `extern "C"` declaration is typically used when you need to interface with code that has been compiled using a C compiler. Since C compilers do not perform name mangling, using `extern "C"` ensures that the function's name remains unchanged and can be correctly referenced from C code.

Extern C

```
// C++ code
extern "C" {
    int func(int* ptr, Foo obj);  // Declares func with C-style linkage
}
```

3 Advanced C++

3.1 What is a dangling pointer? How do you avoid it?

A dangling pointer in C++ refers to a pointer that points to a memory location that has been freed or deleted. Accessing or dereferencing such a pointer can lead to undefined behavior.
To avoid dangling pointers:

- Always initialize pointers when declaring them.
- Avoid dereferencing pointers after they have been deleted or gone out of scope.
- Use smart pointers or nullptr to manage memory and avoid manual deallocation issues.

Dangling Pointer

```
int* ptr = new int(10); // Allocate memory
std::cout << "Value: " << *ptr << std::endl;
delete ptr; // Deallocate memory
std::cout << "Value after delete: " << *ptr << std::endl; // Undefined behavior
```

3.2 Explain the concept of smart pointers in C++.

Smart pointers in C++ are classes that manage the lifetime of dynamically allocated objects, ensuring proper resource management and avoiding memory leaks. They automatically release memory when it's no longer needed, either through reference counting or unique ownership.
Examples include `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`.

3.3 What is memory leakage? How does C++ manage memory?

Memory leakage in C++ occurs when allocated memory is not properly deallocated after use, leading to a gradual depletion of available memory.
C++ manages memory through:

- Manual allocation and deallocation using `new` and `delete` operators.
- Smart pointers (`std::unique_ptr`, `std::shared_ptr`) for automatic memory management.
- RAII (Resource Acquisition Is Initialization) principle for deterministic destruction of objects and freeing of resources.

3.4 What is difference between `delete` and `delete[]` operators in C++?

In C++, `delete` is used to deallocate memory allocated for a single object, while `delete[]` is used to deallocate memory allocated for arrays of objects. Using `delete[]` ensures that the correct number of destructors are called for each element in the array.

3.5 What is STL? Exlpain its components.

STL (Standard Template Library) is a collection of classes and functions in C++ that provides general-purpose classes and algorithms, such as containers (like vectors, lists, maps), algorithms (like sorting, searching), and iterators.
Components of STL:

- Containers: Sequence containers (vector, list, deque), associative containers (set, map), container adaptors (stack, queue).
- Algorithms: Sorting, searching, modifying algorithms.
- Iterators: Generalized pointers that allow traversal of containers.

3.6 What is an iterator? Explain the types of iterators in C++.

An iterator in C++ is an object that allows iteration through elements of a container. It acts like a generalized pointer.
Types of iterators:

- Input iterators: Read-only access and single pass traversal.
- Output iterators: Write-only access and single pass traversal.
- Forward iterators: Read and write access, multi-pass traversal in one direction.
- Bidirectional iterators: Read and write access, multi-pass traversal in both directions.
- Random access iterators: Read and write access with arithmetic operations for random access.

3.7 What are templates in C++? Why are they useful?

Templates in C++ allow generic programming by enabling the creation of functions and classes that operate on types parameterized by other types. They enable code reusability and flexibility by allowing algorithms to work on different data types without rewriting code.

3.8 Explain function templates with an example.

Function templates in C++ define functions that operate on generic types. Here's an example of a function template that computes the maximum of two values:

Function Template

```
template<typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

This template can be instantiated with different types (`int`, `double`, etc.) to create specific functions.

3.9 What is difference between class templates and function templates?

Class templates define blueprints for classes that can work with any data type, while function templates define blueprints for functions that can work with any data type. Class templates define entire classes that are parameterized by one or more types, whereas function templates define functions that are parameterized by one or more types.

Class Template

```
template<typename T>
class Container {
private:
    T element;
public:
    Container(T arg) : element(arg) {}
    T getElement() { return element; }
};
```

3.10 What is specialization in templates?

Template specialization in C++ allows you to provide a different implementation for a template when instantiated with specific types. It allows customization of template behavior for particular data types or scenarios.

Specialization in Template

```
template<>
const char* max<const char*>(const char* a, const char* b) {
    return strcmp(a, b) > 0 ? a : b;
}
```

3.11 How do you create a generic function in C++?

To create a generic function in C++, you use function templates. Here's an example of a generic function to swap two values:

Generic Function

```
template<typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

This function template works for any data type.

3.12 What are lambdas in C++? Provide an example.

Lambdas in C++ are anonymous functions that can be defined inline. They provide a concise way to write function objects or closures. They capture variables from their enclosing scope by value or reference.
Example of a lambda that adds two numbers:

Lambda Function

```
auto add = [](int a, int b) {
    return a + b;
};
```

- Capture Everything by Value (`[=]`)
- Capture Everything by Reference (`[&]`)
- Capture Specific Variables by Value (`[a]`)
- Capture Specific Variables by Reference (`[&a]`)
- Mixed Capture (`[=, &b]`)
- Capture by Value and Reference in C++14 and Later (`[=a, &b]`)

3.13 Explain the difference between `std::function` and function pointers in C++

`std::function` in C++ is a polymorphic function wrapper that can store, copy, and invoke any Callable target—functions, lambdas, bind expressions, or other function objects.
Function pointers, on the other hand, directly point to a function's address and have limited flexibility compared to `std::function`.

- Function Pointers: allow you to store the address of a function and call it indirectly.

Function Pointer

```
#include <iostream>

// A simple function
void printNumber(int number) {
    std::cout << "Number: " << number << std::endl;
}

int main() {
    // Declare a function pointer
    void (*funcPtr)(int) = &printNumber;

    // Call the function through the pointer
    funcPtr(42);

    return 0;
}
```

- `std::function`: is a versatile function wrapper provided by the C++ Standard Library. It can store any callable target, such as function pointers, lambda expressions, and function objects.

std::function

```
#include <iostream>
#include <functional> // For std::function and std::bind
#include <vector>
#include <algorithm> // For std::for_each

void printSum(int a, int b) {
    std::cout << "Sum: " << a + b << std::endl;
}

int main() {
    using namespace std::placeholders; // For _1, _2, etc.

    // Create a std::function with a lambda expression
    std::function<void(int)> func = [](int number) {
        std::cout << "Number: " << number << std::endl;
    };

    func(10);

    // Use std::bind to create a new function with one argument
    std::function<void(int)> boundFunc = std::bind(printSum, _1, 10);
    boundFunc(5);

    // Use std::function with a standard algorithm
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::for_each(vec.begin(), vec.end(), func);

    return 0;
}
```

3.14 What is concurrency in C++?

Concurrency in C++ refers to the ability of a program to perform multiple tasks simultaneously. It involves executing multiple threads or processes concurrently to improve program efficiency and responsiveness.

3.15 Explain the difference between threads and processes.

In C++, threads are lightweight processes within a single process that share memory and resources, allowing concurrent execution. Processes, on the other hand, are independent instances of a program that run in separate memory spaces and communicate through inter-process communication mechanisms.

3.16 What is a mutex? How do you use it in C++?

A mutex (mutual exclusion) in C++ is a synchronization primitive used to protect shared data from being simultaneously accessed by multiple threads. It allows only one thread at a time to lock the mutex and access the shared resource.
Example usage of a mutex:

Mutex

```
#include <mutex>

std::mutex mtx;

void critical_section() {
    mtx.lock();
    // Perform critical operations
    mtx.unlock();
}
```

3.17 What are different types of mutex in C++?

- `std::mutex`: The basic mutex class. It provides the `lock`, `unlock`, and `try_lock` methods.
- `std::timed_mutex`: Extends std::mutex with the ability to attempt locking with a timeout.
- `std::recursive_mutex`: A mutex that can be locked multiple times by the same thread. Each lock must be matched by an unlock operation.
- `std::recursive_timed_mutex`: Extends `std::recursive_mutex` with the ability to attempt locking with a timeout.
- `std::shared_mutex`: A mutex that allows multiple threads to share simultaneous read access while only one thread can hold write access.
- `std::shared_timed_mutex`: Extends `std::shared_mutex` with the ability to attempt locking with a timeout.

3.18 What are the advantages of using `std::thread` over `pthread`?

`std::thread` in C++ provides a portable and standardized interface for creating and managing threads across different platforms. It encapsulates platform-specific details and simplifies thread management compared to using platform-specific libraries like `pthread`.

- **Standardization:** `std::thread` is part of the C++ Standard Library (`<thread>` header), ensuring portability across different platforms and compilers.
- **Integration:** `std::thread` seamlessly integrates with other C++ features like exceptions, RAII (Resource Acquisition Is Initialization), and lambda expressions.
- **Ease of Use:** `std::thread` provides a more modern and intuitive interface compared to `pthread`, making it easier to create and manage threads.

- **Type Safety:** `std::thread` supports type-safe function and object binding, reducing errors related to thread creation and management.

3.19 How do you handle race conditions in C++?

Race conditions in C++ are handled using synchronization techniques such as mutexes, locks, and atomic operations. These techniques ensure that shared resources are accessed in a controlled manner, preventing conflicting accesses by multiple threads.

- **Mutexes:** Use `std::mutex` (or other synchronization primitives like `std::lock_guard`, `std::unique_lock`) to protect shared resources. Only one thread can lock the mutex at a time, ensuring mutual exclusion.

  Mutexes

  ```
  #include <mutex>
  std::mutex mtx;
  int shared_data = 0;

  void incrementSharedData() {
      std::lock_guard<std::mutex> lock(mtx);
      shared_data++;
  }
  ```

- **Atomic Types:** Use `std::atomic` for simple variables like integers that require atomic access operations. Operations on std::atomic variables are guaranteed to be thread-safe.

  Atomic Types

  ```
  #include <atomic>
  std::atomic<int> atomic_data(0);

  void incrementAtomicData() {
      atomic_data++;
  }
  ```

- **Thread Synchronization:** Use condition variables (`std::condition_variable`) to notify waiting threads about changes in shared data, avoiding busy-waiting and improving efficiency.

Thread Synchronization

```
#include <condition_variable>
std::mutex mtx;
std::condition_variable cv;
bool data_ready = false;

void processData() {
    std::unique_lock<std::mutex> lock(mtx);
    // Wait until data is ready
    cv.wait(lock, [] { return data_ready; });
    // Process data
}

void setDataReady() {
    {
        std::lock_guard<std::mutex> lock(mtx);
        data_ready = true;
    }
    cv.notify_one(); // Notify waiting thread
}
```

- **Design Considerations:** Use thread-safe data structures (e.g., std::queue with mutex protection) and minimize shared state between threads to reduce the likelihood of race conditions.

3.20 Explain move semantics in C++11. When and why is it used?

Move semantics in C++11 allows the efficient transfer of resources (like dynamically allocated memory) from one object to another without unnecessary copying. It's used to optimize performance by reducing the overhead of copying large objects, especially in situations involving temporary objects and function returns.

Move Semantics

```
// Move constructor
MyClass(MyClass&& other) noexcept
    : data(std::move(other.data)) {
    other.data = nullptr; // Invalidate the moved-from object
}
```

3.21 What are rvalue references? How are they related to move semantics?

Rvalue references in C++ allow binding to temporary objects (rvalues) and are denoted by `&&`. They enable move semantics by distinguishing between objects that can be safely moved from (rvalues) and objects that should not be moved from (lvalues).

rvalue

```
// Function taking an rvalue reference parameter
void processValue(int&& value) {
    // Use value
}
```

3.22 What is template metaprogramming? Proivde an example.

Template metaprogramming in C++ involves using templates to perform computations at compile-time, rather than run-time. It allows for powerful compile-time optimizations and code generation.
Example of template metaprogramming to calculate factorial:

Template Metaprogramming

```
template<unsigned n>
struct factorial {
    static const unsigned value = n * factorial<n - 1>::value;
};

template<>
struct factorial<0> {
    static const unsigned value = 1;
};

// Usage
unsigned fact = factorial<5>::value; // fact will be 120
```

3.23 Explain `noexcept` specifier in C++. When and why it is used?

The `noexcept` specifier in C++ indicates that a function does not throw exceptions. It's used to improve performance and enable certain optimizations, as well as to provide information to the compiler and programmers about exception safety guarantees.

`noexcept`

```
void func() noexcept {
    // Function body
}
```

3.24 What are the improvements introduced in C++14, C++17, and C++20?

- **C++14:** introduced several enhancements over C++11, focusing mainly on improving usability and flexibility without introducing major breaking changes. Some key features include:
  - **Generic lambdas:** Lambdas can now have auto-declared parameters, allowing them to be used with different argument types.
  - **Relaxed constexpr restrictions:** constexpr functions can now contain a broader range of statements including loops and conditional statements.
  - **Variable templates:** Templates can now be used to define variables as well as functions and classes.
  - **Binary literals and digit separators:** Support for binary literals (e.g., '0b1101') and digit separators (e.g., '1'000'000') improves readability.
  - **`std::make_unique`:** Added to complement `std::make_shared`, facilitating the creation of `unique_ptr` instances.
- **C++17** C++17 builds upon C++14, introducing several new features and improvements to the language:
  - **Structured bindings:** Allows decomposition of objects into their components directly in a simpler way.
  - **`std::optional`:** Provides a type-safe way to represent optional objects, avoiding null pointer issues.
  - **Parallel algorithms:** Standardized execution policies for algorithms, enabling easier parallelization.
  - **if constexpr:** Introduces constexpr if, allowing conditional compilation based on compile-time conditions.
  - **`std::variant`:** A type-safe union that provides an alternative to traditional C-style unions.
  - **Filesystem library:** Adds std::filesystem for portable file system operations, facilitating file and directory management.

- **C++20** introduces significant new features aimed at improving developer productivity and code clarity:
  - **Concepts:** A major addition enabling concepts as a way to specify and constrain template parameters.
  - **Ranges:** Introduces a comprehensive library for working with ranges of elements, replacing iterators in many cases.
  - **Coroutines:** Introduces coroutines, allowing functions to be suspended and resumed at specific points.
  - **Modules:** Introduces module support, improving build times and reducing header file dependencies.
  - **Three-way comparison:** Simplifies and standardizes comparison operations with the introduction of the spaceship operator (‘¡=¿‘).
  - **Calendar and timezone support:** Enhances std::chrono with support for calendars and timezones, improving date and time handling.

4 Object Oriented Programming (OOP)

4.1 What is Class and Object in C++?

- **Class**: In C++, a class is a user-defined data type that encapsulates data and functions into a single unit. It serves as a blueprint for creating objects. For example:

  Class

  ```
  class Car {
      public:
          string brand;
          int year;
  };
  ```

- **Object**: An object is an instance of a class. It represents a real-world entity based on the class blueprint. For example:

  Object

  ```
  Car myCar;
  myCar.brand = "Toyota";
  myCar.year = 2023;
  ```

4.2 What are differences between C++ struct and C++ class?

In C++, the struct and class keywords are largely interchangeable, with the main difference being default access level (public for struct, private for class). Example:

Struct vs Class

```
struct Rectangle {
    int width, height; // Members are public by default
};

class Circle {
    int radius; // Members are private by default
public:
    void setRadius(int r) {
        radius = r;
    }
};
```

4.3 What is object slicing?

Object slicing occurs when a derived class object is assigned to a base class object. The derived class-specific attributes and methods are "sliced off," leaving only the base class attributes and methods.

Object Slicing

```
class Base {
public:
    int baseVar;
};

class Derived : public Base {
public:
    int derivedVar;
};

Base baseObj;
Derived derivedObj;
baseObj = derivedObj; // Object slicing: derivedVar is sliced off
```

4.4 What are the four pillars of OOP? Explain each with an example.

The four pillars of Object-Oriented Programming (OOP) are:

1. **Encapsulation:** Bundling of data and methods that operate on the data into a single unit (class).

   Example of Encapsulation

   ```
   class Employee {
   private:
       std::string name;
       double salary;
   public:
       Employee(std::string empName, double empSalary)
           : name(empName), salary(empSalary) {}
       double getSalary() {
           return salary;
       }
   };
   ```

2. **Abstraction:** Hiding of complex implementation details and exposing only relevant operations.

Example of Abstraction

```
class Shape {
public:
    virtual double calculateArea() = 0; // Pure virtual function
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double calculateArea() override {
        return 3.14 * radius * radius;
    }
};

class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double calculateArea() override {
        return width * height;
    }
};
```

3. **Inheritance:** Mechanism by which a class can inherit properties and behavior from another class.

Example of Inheritance

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

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

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

4. **Polymorphism:** Ability to process objects differently depending on their data type or class.

Example of Polymorphism

```
class Animal {
public:
    virtual void makeSound() {
        std::cout << "Some generic animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Bark" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Meow" << std::endl;
    }
};

void animalSound(Animal *animal) {
    animal->makeSound();
}
```

4.5 What is inheritance? Explain the types of inheritance in C++.

Inheritance is a mechanism in which a class (derived class) inherits properties and behaviors from another class (base class). Types of inheritance in C++ include:

- **Single inheritance:** One class inherits from only one base class.
- **Multiple inheritance:** One class inherits from multiple base classes.
- **Multilevel inheritance:** Deriving a class from another derived class.
- **Hierarchical inheritance:** Multiple derived classes from a single base class.

4.6 What is polymorphism? Explain with an example.

Polymorphism allows objects of different classes to be treated as objects of a common superclass. Example:

Polymorphism

```
class Animal {
public:
    virtual void makeSound() {
        cout << "Animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Bark" << endl;
    }
};
```

4.7 What are types of polymorphism? Explain with an example. Explain the cost associated with each type.

Polymorphism in C++ can be broadly classified into two types: compile-time polymorphism and runtime polymorphism.

- **Compile-time Polymorphism** Compile-time polymorphism is achieved through function overloading and operator overloading. This type of polymorphism is resolved during compile time.
  - **Cost:** Compile-time polymorphism has no runtime overhead since all the decisions are made during compilation. The main cost associated is the increase in binary size due to multiple function definitions.

  Compile-time Polymorphism using Function Overloading

  ```
  #include <iostream>

  class Print {
  public:
      void show(int i) {
          std::cout << "Integer: " << i << std::endl;
      }

      void show(double d) {
          std::cout << "Double: " << d << std::endl;
      }

      void show(std::string s) {
          std::cout << "String: " << s << std::endl;
      }
  };

  int main() {
      Print p;
      p.show(5);
      p.show(5.5);
      p.show("Hello");
      return 0;
  }
  ```

- **Runtime Polymorphism** Runtime polymorphism is achieved through inheritance and virtual functions. This type of polymorphism is resolved during runtime.

- **Cost:** Runtime polymorphism introduces a runtime overhead due to the dynamic binding mechanism (vtable lookup). This can also lead to slightly larger binary sizes and slower execution compared to compile-time polymorphism.

Runtime Polymorphism using Virtual Functions

```
#include <iostream>

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

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

int main() {
    Base* b;
    Derived d;
    b = &d;

    b->show(); // Calls Derived's show()
    return 0;
}
```

4.8 What is encapsulation? Why is it important?

Encapsulation is bundling data and methods that operate on the data into a single unit (class), hiding the internal details of how an object operates. It promotes reusability and allows for better control over data, enhancing security and reducing complexity.

4.9 What is abstraction? Give example of abstraction in C++.

Abstraction is the process of hiding complex implementation details and exposing only relevant operations to the user. Example:

Abstraction

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

class Circle : public Shape {
public:
    void draw() override {
        // Draw circle implementation
    }
};
```

4.10 What is difference between function overloading and function overriding?

- **Function Overloading** Function overloading is a feature in C++ that allows multiple functions to have the same name with different parameters. The correct function to be called is determined at compile time based on the function signature.

  Key Points:

  - Same function name with different parameters.
  - Resolved at compile time.

  Function Overloading

  ```
  #include <iostream>

  class Print {
  public:
      void show(int i) {
          std::cout << "Integer: " << i << std::endl;
      }

      void show(double d) {
          std::cout << "Double: " << d << std::endl;
      }

      void show(std::string s) {
          std::cout << "String: " << s << std::endl;
      }
  };

  int main() {
      Print p;
      p.show(5);          // Calls show(int)
      p.show(5.5);        // Calls show(double)
      p.show("Hello");    // Calls show(std::string)
      return 0;
  }
  ```

- **Function Overriding** Function overriding allows a derived class to provide a specific implementation of a function that is already defined in its base class. The function in the base class must be declared with the `virtual` keyword. The correct function to be called is determined at runtime.

  Key Points:

  - Same function name and parameters in both base and derived classes.
  - Requires `virtual` keyword in the base class.
  - Resolved at runtime.

Function Overriding

```
#include <iostream>

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

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

int main() {
    Base* b;
    Derived d;
    b = &d;

    b->show(); // Calls Derived's show()
    return 0;
}
```

4.11 What is difference between virtual and pure virtual?

A virtual function is a member function in a base class that can be overridden in a derived class, while a pure virtual function is a virtual function declared in a base class with no implementation and must be overridden in derived classes.

- **Normal Functions:** Regular member functions that cannot be overridden in derived classes.
- **Virtual Functions:** Member functions that can be overridden in derived classes and are resolved at runtime.
- **Pure Virtual Functions:** Member functions with no implementation in the base class, must be overridden in derived classes, making the base class abstract.

Normal Function

```
    #include <iostream>

    class Base {
    public:
        void show() {
            std::cout << "Normal function in Base class" << std::endl;
        }
    };

    class Derived : public Base {
    public:
        void show() {
            std::cout << "Normal function in Derived class" << std::endl;
        }
    };

    int main() {
        Base b;
        b.show(); // Calls Base's show()

        Derived d;
        d.show(); // Calls Derived's show()
        return 0;
    }
```

Virtual Function

```
#include <iostream>

class Base {
public:
    virtual void show() {
        std::cout << "Virtual function in Base class" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Virtual function in Derived class" << std::endl;
    }
};

int main() {
    Base* b = new Derived;
    b->show(); // Calls Derived's show() because of virtual function
    delete b;
    return 0;
}
```

Pure Virtual Function

```
#include <iostream>

class Base {
public:
    virtual void show() = 0; // Pure virtual function
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Pure virtual function overridden in Derived class" << std::endl;
    }
};

int main() {
    // Base b; // Error: Cannot instantiate abstract class
    Base* b = new Derived;
    b->show(); // Calls Derived's show()
    delete b;
    return 0;
}
```

4.12 What are abstract classes?

Abstract classes in C++ are classes that contain at least one pure virtual function. They cannot be instantiated on their own and are meant to be subclassed with concrete implementations of their pure virtual functions.

4.13 What is operator overloading?

Operator overloading allows operators to be redefined so that they work on objects of user-defined classes. For example, overloading the + operator to perform addition for custom objects.

Operator Overloading

```
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

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

    // Overload the << operator for output
    friend std::ostream& operator << (std::ostream& out, const Complex& c) {
        out << c.real << " + " << c.imag << "i";
        return out;
    }
};
```

4.14 What is virtual inheritance?

Virtual inheritance in C++ is used to resolve ambiguity caused by multiple inheritance by ensuring that only one instance of a base class exists in the inheritance hierarchy. In C++, the diamond problem occurs when a class inherits from two classes that both inherit from a common base class. This creates ambiguity in the derived class about which inherited class to use for a particular member inherited from the base class.

Virtual Inheritance

```
class A {
public:
    void foo() { std::cout << "A::foo()" << std::endl; }
};

class B : virtual public A {  // Virtual inheritance
public:
    void bar() { std::cout << "B::bar()" << std::endl; }
};

class C : virtual public A {  // Virtual inheritance
public:
    void baz() { std::cout << "C::baz()" << std::endl; }
};

class D : public B, public C {
    // No longer inherits foo() ambiguously
};
```

- **Advantages of Virtual Inheritance**
  - **Avoids Ambiguity:** By ensuring that only one instance of a base class exists within the inheritance hierarchy, virtual inheritance prevents ambiguity in accessing inherited members.
  - **Space Efficiency:** It reduces memory overhead by eliminating redundant base class instances that would occur without virtual inheritance.
- **Disadvantages of Virtual Inheritance**
  - **Complexity:** It adds complexity to the design and understanding of class relationships, especially in larger inheritance hierarchies.
  - **Performance Overhead:** Resolving virtual inheritance involves additional runtime overhead due to pointer adjustments and increased method lookup time.

4.15 What is compile time and run time polymorphism?

- **Compile-time Polymorphism:** Achieved through function overloading and operator overloading. Resolved at compile-time based on the types of arguments or operands.
- **Run-time Polymorphism:** Achieved through function overriding and virtual functions. Resolved at runtime based on the actual type of the object pointed to by the base class pointer or reference.

4.16 What is constructor and destructor?

- **Constructor** A constructor in C++ is a special member function that is automatically called when an object of a class is created. It is used to initialize the object's state. There are several types of constructors:
  - **Default Constructor:** Automatically initializes objects when no initial value is specified explicitly.

Default Constructor

```
class Example {
public:
    Example() {
        // Default constructor
    }
};
```

  - **Parameterized Constructor:** Takes parameters to initialize the object with user-defined values.

Parameterized Constructor

```
class Example {
public:
    int value;
    Example(int val) {
        value = val;  // Parameterized constructor
    }
};
```

  - **Copy Constructor:** Creates a new object as a copy of an existing object.

Copy Constructor

```
class Example {
public:
    int value;
    Example(const Example &obj) {
        value = obj.value;  // Copy constructor
    }
};
```

  - **Constructor Delegation (C++11 and later):** Calling one constructor from another within the same class.

Constructor Delegation

```
class Example {
public:
    int value;
    Example() : Example(0) {}  // Delegating constructor
    Example(int val) {
        value = val;  // Parameterized constructor
    }
};
```

- **Destructor** A destructor in C++ is a special member function that is automatically called when an object goes out of scope or is explicitly deleted. It is used to release resources acquired by the object during its lifetime. Types of destructors include:

  - **Default Destructor:** Automatically provided if no user-defined destructor is specified.

Default Destructor

```
class Example {
public:
    ~Example() {
        // Default destructor
    }
};
```

– **Virtual Destructor:** Ensures that the derived class objects are destructed properly by invoking their destructors first and then the base class destructors.

Virtual Destructor

```
class Base {
public:
    virtual ~Base() {
        // Virtual destructor
    }
};
class Derived : public Base {
public:
    ~Derived() {
        // Derived class destructor
    }
};
```

– **Pure Virtual Destructor:** Declared as virtual and assigned a value 0, which must be implemented in the derived class.

Pure Virtual Destructor

```
class Base {
public:
    virtual ~Base() = 0;  // Pure virtual destructor
};
Base::~Base() {}  // Implementation in derived class
```

– **Non-Virtual Destructor:** Used when you do not need to delete a derived class object using a pointer to the base class.

Non-Virtual Destructor

```
class Base {
public:
    ~Base() {
        // Non-virtual destructor
    }
};
```

4.17 Why there are no virtual constructor?

In C++, constructors cannot be declared as virtual because constructors are responsible for initializing objects and setting up their vtables. However, the vtable itself is set up during the object construction process based on the type of the object being constructed. If constructors were virtual, there would be ambiguity regarding which constructor to call during object creation, as virtual functions are resolved based on the vtable, which is not fully set up until after construction is complete.

4.18 When copy constructor is called and when copy assignment operator?

- The copy constructor is called when a new object is initialized with an existing object of the same class, either by value or as a parameter to a function. It is invoked explicitly or implicitly during initialization. For example:

  Copy Constructor

  ```
  class Example {
  public:
      int value;
      // Copy constructor
      Example(const Example& obj) {
          value = obj.value;
      }
  };

  Example obj1;  // Default constructor
  Example obj2 = obj1;  // Copy constructor called here
  ```

- The copy assignment operator (operator=) is called when an already initialized object is assigned a new value from another object of the same type. For example:

  Copy Assignment Operator

  ```
  Example obj3;
  obj3 = obj1;  // Copy assignment operator called here
  ```

4.19 What is the order in which objects are constructed and destructed?

- Objects in C++ are constructed in the following order:
  - Base classes are constructed before derived classes.
  - Members are constructed in the order they are declared in the class.
- Objects are destructed in the reverse order of construction:
  - Members are destructed in the reverse order of their construction.
  - Derived classes are destructed before base classes.

4.20 When constructor should use initialization list and when assignment?

- Constructors should use initialization lists when initializing data members of a class, especially when those members are objects themselves or have a cost associated with their construction. Initialization lists ensure that data members are initialized before the body of the constructor is executed, which can be more efficient and avoids potential issues with uninitialized variables.

- Assignment inside the constructor body should be used when initialization requires complex logic that cannot be performed in the initialization list or when initialization depends on runtime conditions.

4.21 What is friend class and friend function?

- **Friend Class:** A friend class in C++ is a class that is granted access to the private and protected members of another class. It is declared using the friend keyword within the class definition. For example:

Friend Class

```
class A {
private:
    int data;
    friend class B;  // B is a friend class of A
};

class B {
public:
    void accessA(A& obj) {
        obj.data = 10;  // B can access private member data of A
    }
};
```

- **Friend Function:** A friend function is a function that is not a member of a class but is granted access to its private and protected members. It is declared as a friend using the friend keyword inside the class definition or as a standalone declaration in the enclosing namespace. For example:

Friend Function

```
class A {
private:
    int data;
    friend void printData(A& obj);  // printData is a friend function of A
};

void printData(A& obj) {
    std::cout << obj.data << std::endl;  // can access private member data
}
```

4.22 Explain working of virtual table.

A virtual table (vtable) is used in C++ to support dynamic dispatch of virtual functions for polymorphic classes. Each class with at least one virtual function has its own vtable, which is a static array of pointers to virtual functions. When an object is created, a pointer to its vtable (called the vpointer or vptr) is added as a hidden member of the object. This vptr points to the vtable of the class of the object.

5 Standard Template Library (STL)

5.1 What is the difference between `std::vector` and `std::array`?

Aspect	`std::vector`	`std::array`
Size flexibility	`std::vector` can dynamically resize itself.	`std::array` has a fixed size determined at compile time.
Access	Elements are accessed via iterators or indices.	Supports direct access via array indexing.
Storage location	Typically allocates memory on the heap, managed automatically.	Allocates memory on the stack or as a member of a class.

5.2 What is the difference between `std::map` and `std::unordered_map`?

Aspect	`std::map`	`std::unordered_map`
Ordering	Stores elements in a sorted order based on keys.	Does not maintain any particular order.
Performance	Operations like search, insertion, and deletion are typically slower due to ordered storage.	Provides faster average-time performance for search, insertion, and deletion operations.
Implementation	Uses a binary search tree (usually Red-Black Tree).	Uses hash tables for storage.

5.3 What is the difference between `std::vector` and `std::list`?

Aspect	`std::vector`	`std::list`
Memory allocation	Stores elements contiguously in memory, allowing fast access but slower insertion/deletion at the middle.	Uses a doubly linked list, enabling fast insertion/deletion but slower access.
Iterators	Supports random access iterators, allowing efficient traversal and access to elements.	Supports bidirectional iterators, suitable for sequential traversal.
Size	Uses more memory due to its contiguous storage.	Uses extra memory for pointers in the linked list structure.

5.4 What is the difference between `std::cout` and `std::cerr`?

Aspect	`std::cout`	`std::cerr`
Buffering	Buffered by default, meaning output may not appear immediately on the console.	Unbuffered, ensuring immediate display of error messages.
Usage	Used for general output.	Specifically used for error messages or critical warnings.
Redirection	Can be redirected to a file or another stream.	Typically directed to the standard error output and is not redirected by default.

5.5 What are the differences between `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`?

These are smart pointers in C++ with different ownership and behavior characteristics:

- `std::unique_ptr`: Provides exclusive ownership of an object, allows efficient transfer of ownership, and is move-only (cannot be copied).
- `std::shared_ptr`: Manages shared ownership of an object through reference counting, automatically deallocates the object when no more references exist.

- `std::weak_ptr`: Observes an object managed by `std::shared_ptr` without affecting its reference count, avoiding cyclic dependencies and potential memory leaks.

Smart Pointers

```
std::unique_ptr<int> ptr(new int(10));

std::shared_ptr<int> ptr1(new int(20));
std::shared_ptr<int> ptr2 = ptr1;

std::shared_ptr<int> sharedPtr(new int(30));
std::weak_ptr<int> weakPtr = sharedPtr;
```

5.6 What is `std::sort`?

`std::sort` is a standard library function in C++ used for sorting elements in a container:

- **Functionality:** Sorts the elements in ascending order by default, or using a custom comparator.
- **Complexity:** Average-case time complexity is O(N log N), making it efficient for large datasets.
- **Usage:** Can be applied to various containers like `std::vector`, `std::array`, and arrays, providing a flexible and optimized sorting mechanism.

`std::sort` Example

```
std::vector<int> vec = {5, 2, 9, 1, 5, 6};

std::sort(vec.begin(), vec.end()); // Sort in ascending order
```

5.7 What is `std::move`?

`std::move` is a utility function in C++11 that converts its argument into an rvalue, enabling efficient move semantics:

- **Purpose:** Facilitates the transfer of resources (like ownership of dynamically allocated memory) from one object to another, avoiding unnecessary copying.
- **Usage:** Often used with move constructors and move assignment operators to optimize performance, especially with large or non-copyable objects.

`std::move` Example

```
std::string str = "Hello, World!";
std::string new_str = std::move(str); // Transfer ownership

std::cout << "New string: " << new_str << std::endl;
std::cout << "Original string: " << str << std::endl; // str is now empty
```

5.8 Explain the concept of `constexpr` in C++11. How is it different from `const`?

`constexpr` in C++11 denotes that an object or function is evaluated at compile-time:

- **Variables:** Defines constants that must be initialized with constant expressions and can be used in contexts requiring constant expressions.
- **Functions:** Specifies that a function can be evaluated at compile-time, enabling more efficient computation and better optimization.

- **Difference from `const`:** While `const` ensures that the value of an object does not change, `constexpr` additionally ensures that the value is known at compile-time, allowing its use in compile-time computations and contexts.

`constexpr` Example

```
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
constexpr int result = factorial(5); // evaluated at compile-time
```