https://chatgpt.com/share/6899bf43-dac0-8013-bec0-0f9e9b9415a0

https://chatgpt.com/share/6899bf43-dac0-8013-bec0-0f9e9b9415a0

Garbage Collector = Automatic memory manager that frees unused objects to avoid memory leaks.

1. **Encapsulation** – Wrapping data and methods into a single unit (class).
2. **Abstraction** – Hiding implementation details, showing only essential features.
3. **Inheritance** – Reusing properties and methods of one class in another.
4. **Polymorphism** – Ability of a method or object to behave in multiple ways (compile-time & runtime).

✅ Interview Short Answer for Operations in Containers:

- **Vector/Deque** → Insert at end/front, random access, erase by index.
- **List** → Insert/delete anywhere fast, but no random access.
- **Set/Map** → Sorted, unique elements/pairs.
- **Unordered Set/Map** → Faster but no order.

🎯 Summary (Interview Style Answer)

👉 STL containers are **ready-made data structures** in C++.

- **Sequential** → (`vector` , `deque` , `list` , `forward_list`) → store in linear order.
- **Associative** → (`set` , `multiset` , `map` , `multimap`) → sorted order using tree.
- **Unordered** → (`unordered_set` , `unordered_map` , etc.) → fast hashing, no order.
- **Adaptors** → (`stack` , `queue` , `priority_queue`) → special-purpose containers.

✅ Use Case Rule of Thumb:

- Random Access → `vector`
- Frequent Middle Insert/Delete → `list`
- Unique Sorted Data → `set`
- Key-Value Mapping → `map`
- Faster Access (no order) → `unordered_map`
- LIFO → `stack` , FIFO → `queue` , Max-Min → `priority_queue`

🔥 Summary for Interview:

- `new` → Allocates memory + calls constructor.
- `delete` → Deallocates memory + calls destructor.
- `new[]` / `delete[]` → For arrays.
- Placement `new` → Construct object at specific memory location.
- In **modern C++**, prefer **smart pointers** for automatic memory management.

cpp Copy code

```cpp
#include <memory>
#include <iostream>
using namespace std;

class Demo {
public:
    Demo() { cout << "Demo constructor\n"; }
    ~Demo() { cout << "Demo destructor\n"; }
};

int main() {
    unique_ptr<Demo> d1 = make_unique<Demo>();   // auto delete
    shared_ptr<Demo> d2 = make_shared<Demo>();   // reference counting
}
```

🔹 What is a Smart Pointer?

A **smart pointer** is a C++ object that acts like a pointer but **automatically manages memory**.
It ensures that when the object goes out of scope, the memory is released (RAII – Resource Acquisition Is Initialization principle).

C++ provides these main smart pointers in `<memory>`:

1. `unique_ptr` → Exclusive ownership (only one smart pointer can own the object).
2. `shared_ptr` → Shared ownership (multiple smart pointers can share ownership).
3. `weak_ptr` → Non-owning reference (used with `shared_ptr` to avoid circular references).

What is the difference between C and C++?

> "C is a procedural programming language, while C++ is multi-paradigm, supporting both procedural and object-oriented programming. C++ adds features like classes, inheritance, polymorphism, function overloading, and better data security, which are not present in C."

What are the main features of C++?

> "C++ is a powerful, object-oriented, and platform-independent language. It supports features like classes, inheritance, polymorphism, function/operator overloading, dynamic memory management, and a rich Standard Template Library, while still allowing low-level programming like C."

What is the difference between declaration and definition?

Key Difference Table

Aspect	Declaration	Definition
Memory	No memory allocated (for vars)	Allocates memory or provides body
Purpose	Tells compiler about existence	Actually creates/implements it
Example	`extern int a;`	`int a = 5;`

Example Short Answer in Interview

> "A declaration tells the compiler about the variable or function's name and type without allocating memory, while a definition actually allocates memory or provides the function's body. Every definition is a declaration, but not every declaration is a definition."

What is the use of the main() function in C++?

Answer:

> In C++, the `main()` function is the **entry point** of every program. It is where program execution begins and ends. The operating system calls `main()` when the program starts, and the value it returns is sent back to the operating system to indicate whether the program ended successfully (`0`) or with an error (non-zero).

Key Points:

1. **Execution Starts Here** – The first function to run in a C++ program.
2. **Return Value** – Usually returns `0` for success or another value for errors.
3. **Signature Variants** –

```cpp
int main()           // No command-line arguments
int main(int argc, char* argv[]) // With command-line arguments
```

4. **Mandatory in Every C++ Program** – Without `main()`, the program won't run (except in embedded systems with a custom entry point).

Short Interview Answer

> "The `main()` function is the entry point of a C++ program. Execution starts from here, and its return value tells the operating system whether the program ended successfully or with an error."

What are header files? Give some examples.

Short Interview Answer

> "Header files in C++ contain function, class, and constant declarations that are included in programs using `#include`. They make code reusable and modular. Examples include `<iostream>`, `<cmath>`, `<string>`, and user-defined files like `myheader.h`."

What is the difference between #include and #include "filename"?

> "`#include <file>` searches only in system directories, used for standard library headers. `#include "file"` searches the current directory first, then system directories, and is used for user-defined headers."

In C++, the `#include` directive is used to include header files in a program, but the way the file is searched depends on the type of brackets used.

1. `#include <filename>`
   - **Meaning**: Tells the compiler to look for the header file **only in the standard library directories** (system-defined paths).
   - **Use**: For standard library headers like `<iostream>`, `<cmath>`, `<string>`.
   - **Example**:

```cpp
#include <iostream> // Searches system directories
```

2. `#include "filename"`
   - **Meaning**: Tells the compiler to **first search in the current working directory** (where your program is), and if not found, then it searches the system library directories.
   - **Use**: For user-defined header files or project-specific files.
   - **Example**:

```cpp
#include "myheader.h" // Searches current folder first
```

Explain namespace in C++.

Short Interview Answer

> "A namespace in C++ is a container for identifiers that helps avoid naming conflicts and organize code. The standard library uses the `std` namespace, and we can create our own namespaces for project-specific code."

how its handle name conflict ?4

Short Interview Line

> "Namespaces prevent name conflicts by giving each identifier its own scope, so two items with the same name in different namespaces don't clash."

cpp Copy Edit

```cpp
#include <iostream>
using namespace std;

namespace LibA {
    void print() { cout << "From Library A"; }
}

namespace LibB {
    void print() { cout << "From Library B"; }
}

int main() {
    LibA::print(); // Calls function from LibA
    LibB::print(); // Calls function from LibB
}
```

✅ **Reason**: Each `print()` is inside its own namespace (`LibA` and `LibB`), so they're **treated as separate functions** even though they have the same name.

What is the difference between signed and unsigned integers?

Example:

cpp Copy Edit

```cpp
#include <iostream>
using namespace std;

int main() {
    int signedNum = -10;       // Allowed
    unsigned int unsignedNum = 10; // Allowed
    // unsignedNum = -5;       // ❌ Error: negative not allowed
}
```

Short Interview Answer

> "A signed integer can store both positive and negative values, while an unsigned integer stores only non-negative values, effectively doubling the positive range for the same size."

What is the size of int, float, double, char on your compiler?

Data Type	Typical Size	Example Range (Signed)
`int`	4 bytes	-2,147,483,648 to 2,147,483,647
`float`	4 bytes	~ ±3.4 × 10^38 (6–7 decimal digits)
`double`	8 bytes	~ ±1.7 × 10^308 (15–16 digits)
`char`	1 byte	-128 to 127 (signed) or 0–255 (unsigned)

Explain type casting in C++ with examples.

Short Interview Answer

> "Type casting in C++ means converting one data type to another. It can be implicit (automatic) or explicit (manual) using C-style casts, function-style casts, or C++ cast operators like `static_cast`. Example: `(double)a / b` converts `a` to double for accurate division."

1. Implicit Type Casting (Type Conversion / Type Promotion)

- Done **automatically** by the compiler.
- Converts smaller types to larger types to prevent data loss.
- Example:

```cpp
int a = 5;
double b = a; // int automatically converted to double
```

2. Explicit Type Casting (Type Casting by Programmer)

- Done **manually** using casting operators.

Methods:

1. C-style Cast

```cpp
double pi = 3.14;
int x = (int)pi; // pi becomes 3
```

2. Function-style Cast

```cpp
double pi = 3.14;
int x = int(pi); // pi becomes 3
```

What is the difference between const, constexpr, and #define?

Short Interview Answer

> "`const` creates a read-only variable that can be set at runtime or compile time. `constexpr` ensures the value is evaluated at compile time. `#define` is a preprocessor macro that replaces text before compilation, but it's not type-safe and has no scope."

1. `const` – Constant variable (value cannot be modified after initialization)

cpp Copy Edit

```cpp
const int x = 10; // Cannot change x later
```

2. `constexpr` – Compile-time constant (must be evaluated at compile time)

cpp Copy Edit

```cpp
constexpr int square(int n) { return n * n; }
int arr[square(3)]; // Allowed because result known at compile time
```

3. `#define` – Preprocessor macro (simple text substitution)

cpp Copy Edit

```cpp
#define PI 3.14159
double area = PI * r * r;
```

What is the difference between ++i and i++?

Example:

cpp Copy Edit

```cpp
int i = 5;
int x = ++i;  // i becomes 6, then x = 6
```

Result:

- `i = 6`
- `x = 6`

Quick Summary Table:

Operation	Action Order	Example (`i=5`)	Final i	Assigned Value
`++i`	Increment → Use	`x = ++i;` → `x=6`	6	6
`i++`	Use → Increment	`x = i++;` → `x=5`	6	5

💡 **Tip:**

- Use `++i` in loops when you **don't** need the old value — it's slightly more efficient.
- Use `i++` when you **need** the old value before increment.

What is the difference between struct and class in C++?

0

> In C++, both `struct` and `class` are user-defined types that can have variables and functions.
> The main difference is that **struct members are public by default**, while **class members are private by default**.
> Also, **structs default to public inheritance** and **classes to private inheritance**.

Explain this pointer.

The `this` **pointer** is like the class's personal ID card—it tells the function, *"Hey, I'm working on THIS object right now!"*

What is the difference between call by value and call by reference?

1. Call by Value

- **Definition:** A copy of the actual value is passed to the function.
- **Effect:** Changes made inside the function **do not affect** the original variable.
- **When to Use:** When you want to protect the original data from being changed.

Example:

cpp Copy Edit

```cpp
#include <iostream>
using namespace std;

void changeValue(int x) {
    x = 50; // only changes local copy
}

int main() {
    int num = 10;
    changeValue(num);
    cout << "Value of num: " << num; // Output: 10 (unchanged)
    return 0;
}
```

2. Call by Reference

- **Definition:** The function receives the **address** of the variable (using `&`), so it can modify the original value directly.
- **Effect:** Changes made inside the function **do affect** the original variable.
- **When to Use:** When you want the function to modify the original data or avoid copying large data.

Example:

cpp Copy Edit

```cpp
#include <iostream>
using namespace std;

void changeValue(int &x) {
    x = 50; // modifies original variable
}

int main() {
    int num = 10;
    changeValue(num);
    cout << "Value of num: " << num; // Output: 50 (changed)
    return 0;
}
```

What are inline functions?

What is an Inline Function in C++?

An **inline function** is a function where the compiler replaces the function call with the **actual code of the function** during compilation.
This avoids the overhead of a **function call** (like jumping to function code, passing parameters, and returning back).

```
inline return_type function_name(parameters) {
    // function body
}
```

Example

cpp Copy Edit

```cpp
#include <iostream>
using namespace std;

inline int square(int x) {
    return x * x;  // Function is small, so can be inline
}

int main() {
    cout << "Square of 5: " << square(5) << endl;
    cout << "Square of 8: " << square(8) << endl;
    return 0;
}
```

What is the difference between function overloading and overriding?

> **Function overloading** is when multiple functions have the same name but different parameter lists, decided at compile time.
> **Function overriding** is when a derived class redefines a base class function with the same signature, decided at runtime using virtual functions.

What are default arguments in C++?

Short Interview Answer

"In C++, default arguments are values assigned to function parameters that are used when the caller does not pass those arguments. They must be assigned from right to left and simplify function calls by avoiding unnecessary overloading."

Syntax:

```cpp
#include <iostream>
using namespace std;

void greet(string name = "Guest", int age = 18) {
    cout << "Hello " << name << ", age: " << age << endl;
}

int main() {
    greet();                    // Uses both default values
    greet("Saurabh");           // Uses default for age
    greet("Rahul", 25);         // No default used
}
```

What are the four pillars of OOP?

> The four pillars of OOP are **Encapsulation** (data hiding), **Abstraction** (show essential details, hide complexity), **Inheritance** (reusability), and **Polymorphism** (same interface, different behavior).

Explain class and object with examples.

Short Interview Answer

> **A class** is a blueprint for creating objects, defining variables and methods.
> **An object** is an instance of a class with its own values for the variables.

What is the difference between struct and class in C++?

> In C++, both `struct` and `class` are user-defined types that can have variables and functions.
> The main difference is that **struct members are public by default**, while **class members are private by default**.
> Also, **structs default to public inheritance** and **classes to private inheritance**.

What is a constructor? What is a destructor?

Key Points

- Name is **same as the class name**.
- **No return type** (not even `void`).
- Can be overloaded (multiple constructors with different parameters).
- Can have **default arguments**.
- Runs **only once** when the object is created.

2. Destructor

A **destructor** is a special function in a class that is **automatically called** when the object goes out of scope or is deleted.
It is mainly used to **free resources** (like memory, files, network connections).

Key Points

- Name is the **same as the class name** but with a **tilde (~)** in front.
- **No parameters** and **no return type**.
- Cannot be overloaded (only **one destructor** is allowed).
- Runs **only once** when the object is destroyed.

What is the difference between copy constructor and assignment operator?

1. Copy Constructor

- **Purpose**: Initializes a new object as a copy of an existing object.
- **When Called**:
  - When a new object is created from an existing object **at the time of declaration**.
  - When passing an object **by value** to a function.
  - When returning an object **by value** from a function.
- **Syntax**:

```cpp
ClassName(const ClassName &obj);
```

Explain this pointer.

✅ **In short:**
The `this` **pointer** is like the class's personal ID card—it tells the function, *"Hey, I'm working on THIS object right now!"*

What is virtual function and pure virtual function?

1. Virtual Function

- A **virtual function** is a member function in the base class that you expect to **override in derived classes**.
- When you call a virtual function **through a base class pointer/reference**, C++ will decide **at runtime** which version of the function to call (base or derived).
- This is called **Dynamic Polymorphism**.

Key points:

- Declared using the `virtual` keyword in the base class.
- If a derived class overrides it, the **derived class version** will be called.
- If not overridden, the **base class version** will be used.

2. Pure Virtual Function

- A **pure virtual function** is a **virtual function with no definition** in the base class — it forces derived classes to provide their own implementation.
- Declared by assigning `= 0` in the declaration.
- A class with at least **one pure virtual function** becomes an **abstract class**.
- You **cannot create objects** of abstract classes.

Difference Table

Feature	Virtual Function	Pure Virtual Function
Definition in Base	Has a body	No body (`=0`)
Override in Derived	Optional	**Mandatory**
Abstract Class?	Not required	Required
Object Creation	Possible	Not possible for base
Purpose	Provide default behavior that can be overridden	Force derived classes to implement

Explain friend function and friend class.

1. Friend Function

A **friend function** is a function that is **not a member** of a class but is allowed to access the class's **private** and **protected** members.

Difference Table

Feature	Friend Function	Friend Class
Definition	A single function that can access private/protected members of a class.	An entire class whose all member functions can access private/protected members of another class.
Scope	Limited to that specific function.	All functions of that class get access.
Declaration	`friend void funcName(ClassName obj);`	`friend class ClassName;`
Use Case	For specific operations like operator overloading.	When two classes are closely related and need full access to each other's data.

Difference between malloc/free and new/delete.

✅ **In short:**

- Use `new` / `delete` in C++ for objects (safe & calls constructors/destructors).
- Use `malloc` / `free` only for raw memory allocation, mostly in C code or when very low-level control is needed.

✅ **Interview Tip:**

If asked **"Which should I use in C++?"** — say:

> *"In C++, `new/delete` is preferred because they are type-safe, call constructors/destructors, and integrate with C++ object-oriented features. `malloc/free` should be avoided unless working with legacy C code or specific low-level memory needs."*

What is a memory leak? How do you avoid it?

Memory Leak:

A **memory leak** occurs when a program allocates memory dynamically (using `new` / `malloc`) but fails to release it (`delete` / `free`) after use.

This means the memory remains occupied even though it's no longer needed, eventually reducing available memory and possibly causing the program or system to crash.

Example:

cpp Copy Edit

```cpp
int* ptr = new int(5);
// Forgot to delete ptr
```

What is the difference between stack and heap memory?

💡 **Interview Tip:**

If they ask follow-up:

- Stack is for **static/deterministic** memory needs (size known at compile time).
- Heap is for **dynamic/non-deterministic** memory needs (size decided at runtime).

Explain dangling pointer.

A **dangling pointer** is a pointer that still holds the memory address of a variable or object that has already been **deleted or gone out of scope**.
Since the memory is no longer valid, using a dangling pointer can lead to **undefined behavior**, crashes, or data corruption.

How to avoid dangling pointers

✅ **Set pointer to `nullptr` after deleting**

cpp Copy Edit

```cpp
delete p;
p = nullptr;
```

✅ **Avoid returning addresses of local variables**

✅ **Use smart pointers (`std::unique_ptr`, `std::shared_ptr`) – they manage memory automatically.**

💡 **Interview tip:**

> "A dangling pointer points to memory that has been freed or is out of scope. It's dangerous because dereferencing it causes undefined behavior. To avoid it, set pointers to `nullptr` after freeing, avoid returning addresses of local variables, and use smart pointers in modern C++."

What is static keyword used for (in variables, functions, and classes)?

The static keyword makes variables retain their value across calls, makes class variables and functions shared across all objects, and limits global functions/variables to file scope. There is no direct static class in C++.

2. Static Functions

- **Inside a class:**
  - Can be called **without creating an object** of the class.
  - Cannot access non-static members directly.

```cpp
class MyClass {
public:
    static void staticFunc() {
        cout << "Static function called\n";
    }
};
```

- **At global/namespace level:**
  - Limits the function scope to the **current file** (internal linkage).

1. Static Variables

- **Inside a function:**
  - Retains its **value between function calls** (preserves state).
  - Initialized only once.

```cpp
void func() {
    static int count = 0;
    count++;
    cout << count << endl;
}
```

- **Inside a class:**
  - Shared by **all objects** of the class (only one copy exists).
  - Accessed using `ClassName::variable` or through objects.

3. Static Classes

- C++ does **not have static classes** like C# or Java.
- However, a class with all static members can behave like a static class.

What is mutable keyword?

The mutable keyword allows a member variable to be modified even if it belongs to a const object.

Explanation

- The `mutable` keyword allows a **class data member** to be modified even if it is inside an object that is declared as `const`.
- Normally, in a `const` object, you **cannot change** any member variables. But `mutable` **members are an exception** to this rule.
- Useful for things like **caching**, **reference counting**, or **lazy evaluation**, where you want to modify some internal state without changing the logical state of the object.

What are templates? Give an example of a function template.

Short Interview Answer

> Templates allow writing generic functions or classes that work with any data type. For example, a function template can compare two values and return the maximum regardless of their type.

What is exception handling in C++?

What is Exception Handling in C++?

- Exception handling is a mechanism to **detect and handle runtime errors** (exceptions) so that the normal flow of the program can be maintained.
- It uses three keywords:
  - `try` — block where you write code that might throw an exception.
  - `throw` — used to **throw** an exception when an error occurs.
  - `catch` — block to **handle** the exception thrown.

What are the components of STL?

STL has **four main components**:

1. **Containers**
   - Data structures to store collections of objects.
   - Examples: `vector`, `list`, `deque`, `set`, `map`, `unordered_map`, etc.
2. **Algorithms**
   - Functions to perform operations like searching, sorting, counting, manipulating on containers.
   - Examples: `sort()`, `find()`, `accumulate()`, `binary_search()`, etc.
3. **Iterators**
   - Objects that point to elements in containers and allow traversal.
   - They work like pointers and provide a common interface for different containers.
   - Types: input iterator, output iterator, forward iterator, bidirectional iterator, random access iterator.
4. **Function Objects (Functors)**
   - Objects that can be called like functions.
   - Used to customize algorithms, e.g., custom sorting criteria.
   - Example: `std::greater<int>()`, or your own class with `operator()`.

2. Order of Elements

- `map` stores elements in **sorted order** based on keys.
- `unordered_map` stores elements in **no particular order**.

How do you iterate over a vector?

Access elements of a container (array, vector, map, etc.) one by one without knowing the internal structure**.**

```
    vector<int> v = {10, 20, 30, 40};
    for (size_t i = 0; i < v.size(); i++) {
        cout << v[i] << " ";
    }
    return 0;
}
```

Copy Edit

2. **Using iterators:**

```cpp
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
```

Copy Edit

3. **Using range-based for loop (C++11 and later):**

```cpp
for (int x : v) {
    cout << x << " ";
}
```

Copy Edit

Initializer list is used to initialize member variables before the constructor body executes. It is mandatory for const and reference data members, because they must be initialized at the time of object creation and cannot be assigned later.

syntax :

```
class Demo {
   const int x;   // const → must be initialized
   int &ref;      // reference → must be initialized
public:
   Demo(int a, int &r) : x(a), ref(r) {   // ✅ initializer list
      // constructor body
   }
};
```

-- Inline function it replace function call with actual code then how --

```
inline int add(int a, int b) {
   return a + b;
}
int main() {
   int x = add(2, 3);   // compiler replaces → int x = 2 + 3;
   int y = add(5, 6);   // compiler replaces → int y = 5 + 6;
   int x = 2 + 3;  // if not inline
   int y = 5 + 6;
}
```

👉 this = a pointer that always points to the **object which called the function**.

File Handling It is Flow of Data :

Stream -Input-istream-cin | and ouput-ostream-cout Stream
cin it is object of istream - cout it is object of output stream

```
int main(){
        ofstream outfile(“myfile.txt”,ios::mode);
}
```

ofstream if file in exist then it open if not exist then it create

Mode	**Meaning**
ios::in	Open for reading
ios::out	Open for writing (overwrite file)
ios::app	Open for appending (add data at end)
ios::binary	Open file in binary mode
ios::trunc	Delete old content before writing
ios::ate	Open file and move pointer at end