|  |
| --- |
| **Day 1** C++ Overview Environment Setup Basic Program Structure Data Types and Variables Operators Input/Output    Conditional Statements Loops Functions Recursion Modular Programming |
| **Day 2** Arrays Strings Pointers Functions and Pointers Pointers to Pointers and Dynamic Memory Intro    Structures Introduction to Data Structures |
| **Day 3** Dynamic Memory Management Classes and Objects Constructors and Destructors Inheritance Polymorphism Encapsulation and Abstraction  New, new [], new dynamic  **Day 4** Boost Libraries Overview Introduction to Boost: philosophy and modularity Boost Smart Pointers Boost Filesystem Boost Asio (intro only) Boost Regex Boost Bind and Lambda Templates and STL (Standard Template Library) Function templates vs class templates  Variable Template Template specialization & partial specialization Variadic templates SFINAE and enable\_if Type traits and decltype, auto Containers: vector, list, map, unordered\_map, set Iterators and algorithms Functors, lambdas, and std::function Smart pointers (unique\_ptr, shared\_ptr) Exception handling in STL  **Day 5** Windows Programming with MFC & ATL MFC architecture and message maps Creating a simple MFC dialog-based application Event handling and UI controls Document/View architecture (overview) COM basics and ATL’s role Creating a simple COM component using ATL ATL Smart pointers (CComPtr, CComBSTR) ATL vs MFC: When to use what Using STL/Boost in MFC/ATL applications Exception safety and RAII in GUI apps Debugging and memory management tools (basics) Performance tips and best practices (basics) Project/Practice Ideas (mini project : Hands On) |

**Day 1**

C++ Overview  
Environment Setup  
Basic Program Structure  
Data Types and Variables  
Operators  
Input/Output

Conditional Statements  
Loops  
Functions  
Recursion  
Modular Programming

**1. Introduction**

* **C++** is a **general-purpose, high-performance programming language** created by **Bjarne Stroustrup** in **1979** as an extension of C.
* It supports **multiple paradigms**:
  + **Procedural programming** (like C)
  + **Object-Oriented Programming (OOP)**
  + **Generic programming** (templates)
  + **Functional programming** (lambdas, std::function)

**2. Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Compiled language** | Converts code into machine code for speed. |
| **Strongly typed** | Enforces type safety, but allows explicit casting. |
| **OOP support** | Classes, inheritance, polymorphism, encapsulation, abstraction. |
| **Memory management** | Manual (new, delete) and automatic (RAII, smart pointers). |
| **Standard Template Library (STL)** | Ready-made data structures & algorithms. |
| **Multi-paradigm** | Supports OOP, procedural, generic, and functional styles. |
| **Low-level access** | Can directly manipulate hardware & memory. |

**3. C++ Program Structure**

#include <iostream> // Preprocessor directive

using namespace std; // Namespace

int main() { // Entry point

cout << "Hello, C++!" << endl; // Output

return 0; // Exit status

}

**Flow**: Preprocessing → Compilation → Linking → Execution.

**4. Core Concepts**

**A. Data Types**

* **Basic**: int, float, double, char, bool
* **Derived**: arrays, pointers, references
* **User-defined**: struct, class, enum

**B. Operators**

* Arithmetic (+ - \* / %)
* Relational (== != < > <= >=)
* Logical (&& || !)
* Bitwise (& | ^ << >>)
* Assignment (= += -= etc.)

**C. Control Statements**

* Conditional: if, if-else, switch
* Looping: for, while, do-while, range-based for
* Jump: break, continue, goto (discouraged)

**5. Object-Oriented Features**

* **Class & Object** – Encapsulates data & behavior
* **Inheritance** – Code reuse and hierarchy (public, protected, private)
* **Polymorphism** – Overloading (compile-time), overriding (run-time via virtual)
* **Encapsulation** – Access modifiers (private, public, protected)
* **Abstraction** – Pure virtual functions & abstract classes

Example:

class Shape {

public:

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

};

class Circle : public Shape {

public:

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

};

**6. Memory Management**

* **Manual**:
* int\* p = new int(5);
* delete p;
* **Smart Pointers** (std::unique\_ptr, std::shared\_ptr, std::weak\_ptr)
* **RAII** (Resource Acquisition Is Initialization) — allocate in constructor, release in destructor.

**7. Standard Template Library (STL)**

* **Containers**: vector, list, map, set, unordered\_map
* **Algorithms**: sort(), find(), count\_if()
* **Iterators**: traverse containers

#include <vector>

#include <algorithm>

vector<int> v = {4, 1, 3};

sort(v.begin(), v.end());

**8. Modern C++ Enhancements (C++11 to C++23)**

* **C++11**: auto, range-based for, lambdas, nullptr, smart pointers, move semantics
* **C++14**: generic lambdas, binary literals
* **C++17**: if constexpr, structured bindings, std::optional
* **C++20**: concepts, ranges, coroutines, modules
* **C++23**: std::expected, improved constexpr, pattern matching proposals

**9. Typical Use Cases**

* Game development (Unreal Engine)
* High-performance computing
* Operating systems, embedded systems
* Real-time simulations
* Financial systems

**10. Advantages & Disadvantages**

✅ **Pros**:

* Fast execution
* Rich library support
* Control over hardware/memory
* Multi-paradigm flexibility

⚠️ **Cons**:

* Complex syntax
* Manual memory management risk
* Longer compile times

**1. Install Visual Studio 2022 with C++ Support**

1. **Download Visual Studio 2022**
   * Go to the official site: https://visualstudio.microsoft.com/downloads/
   * Choose **Community Edition** (free) unless you have Professional/Enterprise.
2. **Run the installer**
   * When the **Visual Studio Installer** opens, it shows **Workloads** (feature sets).
3. **Select C++ Development Workloads**
   * Check **"Desktop development with C++"**.
   * Optional but useful:
     + **Game development with C++** (if using Unreal Engine or DirectX)
     + **Linux development with C++** (if targeting WSL/Linux)
4. **Select optional components**
   * MSVC v143 - C++ compiler toolset (default for VS2022)
   * Windows 10/11 SDK (latest)
   * C++ CMake tools for Windows (optional but good)
   * C++ ATL/MFC (optional if needed for legacy GUI apps)
5. **Install & wait**
   * Click **Install** (or **Install while downloading** for faster start).

**2. Verify Installation**

1. **Open Visual Studio 2022**.
2. Go to **Help → About Microsoft Visual Studio**.
3. Ensure **Desktop development with C++** is listed in Installed components.

**3. Create Your First C++ Project**

1. **Launch Visual Studio**.
2. **File → New → Project**.
3. Select:
   * **Console App** (C++) → Click **Next**.
4. Enter:
   * **Project Name**: HelloCPP
   * **Location**: Choose a folder
   * Uncheck **Place solution and project in the same directory** (optional)
5. Click **Create**.

**4. Write a Simple Program**

Replace the template code with:

#include <iostream>

using namespace std;

int main() {

cout << "Hello, C++ in Visual Studio 2022!" << endl;

return 0;

}

**5. Build and Run**

* **Build**: Press Ctrl + Shift + B or **Build → Build Solution**.
* **Run**: Press Ctrl + F5 (**Start Without Debugging**)  
  *(This keeps the console window open after running.)*

**6. Configure Useful Settings**

**Keep Console Open in Debug Mode**

1. Go to **Project → Properties**.
2. Navigate to:  
   Configuration Properties → Linker → System.
3. Add system("pause"); at the end of main() **OR** use **Ctrl+F5** instead of F5.

**Enable C++20/23 Features**

1. **Right-click Project → Properties**.
2. Configuration Properties → C/C++ → Language.
3. **C++ Language Standard**: Set to /std:c++20 or /std:c++latest.

**Change Output Path**

1. **Right-click Project → Properties**.
2. Configuration Properties → General.
3. Set **Output Directory** where you want .exe files to be saved.

**7. Testing Your Setup**

You can check the compiler version by adding:

#include <iostream>

int main() {

std::cout << \_\_cplusplus << "\n";

}

For **C++20**, you should see 202002L.

Here’s the **basic structure of a C++ program** — the skeleton that almost all programs follow.

**1. Structure Layout**

// 1. Preprocessor Directives

#include <iostream> // Include library for input/output

// 2. Namespace Declaration

using namespace std; // Avoids writing std:: before cout, cin, etc.

// 3. main() Function - Entry Point

int main() {

// 4. Variable Declarations & Statements

cout << "Hello, C++!" << endl; // Output statement

return 0; // 5. Return statement - end of program

}

**2. Sections Explained**

| **Section** | **Purpose** |
| --- | --- |
| **Preprocessor Directives** | Start with #. They include files, define macros, etc. (#include <iostream>) |
| **Namespace** | Groups identifiers to avoid name conflicts (using namespace std;). |
| **main() Function** | The execution starts here. Must return int. |
| **Statements** | Actual instructions to execute. |
| **Return Statement** | Returns status to the OS (0 means success). |

**3. Example with Comments**

#include <iostream> // Preprocessor directive for I/O functions

using namespace std; // Use standard namespace

int main() {

// Declare a variable

string name = "World";

// Display message

cout << "Hello, " << name << "!" << endl;

// Exit program successfully

return 0;

}

**4. Execution Flow**

1. **Preprocessing** → Handles #include, macros.
2. **Compilation** → Converts C++ to object code.
3. **Linking** → Combines with libraries to create .exe.
4. **Execution** → Runs the program.

**1. What is a Data Type?**

A **data type** tells the compiler:

* **What kind of data** a variable will hold.
* **How much memory** to allocate.
* **What operations** can be performed on it.

**2. Categories of Data Types**

**A. Fundamental (Primitive) Types**

| **Type** | **Example** | **Size\*** | **Meaning** |
| --- | --- | --- | --- |
| int | 42 | 4 bytes | Integer numbers |
| float | 3.14f | 4 bytes | Single-precision decimal |
| double | 3.14159 | 8 bytes | Double-precision decimal |
| char | 'A' | 1 byte | Single character |
| bool | true | 1 byte | Boolean (true/false) |
| void | — | 0 bytes | No value (used in functions) |

\*Size can vary based on compiler/architecture (above is typical for 64-bit).

**B. Derived Types**

* **Arrays**: int arr[5] = {1, 2, 3, 4, 5};
* **Pointers**: int \*p = &x;
* **References**: int &ref = x;

**C. User-Defined Types**

* **struct**
* **class**
* **enum**
* **typedef / using aliases**

**3. Variables**

A **variable** is a named memory location that stores data.

**Syntax:**

data\_type variable\_name = value; // optional initialization

**Examples:**

int age = 25; // integer

float height = 5.9f; // floating-point

char grade = 'A'; // character

bool isActive = true; // boolean

**4. Rules for Naming Variables**

* Must start with a **letter** or \_
* Can contain letters, digits, and \_
* **Case-sensitive** (age and Age are different)
* Cannot be a **reserved keyword** (like int, class)

**5. Variable Initialization**

**A. Direct Initialization**

int a = 10;

**B. Brace Initialization (C++11)**

int b{20};

**C. Copy Initialization**

int c(30);

**6. Constants**

Use const to make a variable read-only:

const double PI = 3.14159;

Use constexpr for compile-time constants (C++11+):

constexpr int MAX = 100;

**7. Type Modifiers**

| **Modifier** | **Meaning** |
| --- | --- |
| signed | Can hold positive and negative |
| unsigned | Only positive |
| short | Uses less memory |
| long | Uses more memory (long long for even bigger) |

Example:

unsigned int positiveNumber = 100;

long long bigNumber = 9876543210;

**8. auto Type Deduction (C++11+)**

auto x = 10; // int

auto y = 3.14; // double

auto z = "Hello"; // const char\*

**9. Example Program**

#include <iostream>

using namespace std;

int main() {

int age = 30; // integer

double weight = 65.5; // floating point

char grade = 'A'; // character

bool isPassed = true; // boolean

const double PI = 3.14159; // constant

cout << "Age: " << age << endl;

cout << "Weight: " << weight << endl;

cout << "Grade: " << grade << endl;

cout << "Passed: " << isPassed << endl;

cout << "PI: " << PI << endl;

return 0;

}

**1. What is an Operator?**

An **operator** is a symbol that tells the compiler to perform a specific operation on operands (variables, constants, expressions).

Example:

int x = 10 + 5; // '+' is an operator, x is assigned 15

**2. Types of Operators in C++**

**A. Arithmetic Operators**

| **Operator** | **Meaning** | **Example (a=10, b=3)** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division | a / b | 3 |
| % | Modulus (remainder) | a % b | 1 |

**B. Relational (Comparison) Operators**

| **Operator** | **Meaning** | **Example (a=10, b=3)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater/equal | a >= b | true |
| <= | Less/equal | a <= b | false |

**C. Logical Operators**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| && | AND | (a > 5 && b < 5) | true |
| ` |  | ` | OR |
| ! | NOT | !(a > b) | false |

**D. Assignment Operators**

| **Operator** | **Example** | **Equivalent to** |
| --- | --- | --- |
| = | x = 5 | — |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| \*= | x \*= 3 | x = x \* 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |

**E. Increment / Decrement Operators**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| ++x | Pre-increment | Increase, then use |
| x++ | Post-increment | Use, then increase |
| --x | Pre-decrement | Decrease, then use |
| x-- | Post-decrement | Use, then decrease |

**F. Bitwise Operators (operate on binary level)**

| **Operator** | **Meaning** | **Example (a=5 (0101), b=3 (0011))** | **Result (binary)** |
| --- | --- | --- | --- |
| & | AND | a & b → 1 (0001) | 1 |
| ` | ` | OR | `a |
| ^ | XOR | a ^ b → 6 (0110) | 6 |
| ~ | NOT | ~a → Inverts bits | depends on system |
| << | Left shift | a << 1 → 10 (1010) | 10 |
| >> | Right shift | a >> 1 → 2 (0010) | 2 |

**G. Conditional (Ternary) Operator**

**Syntax:**

condition ? value\_if\_true : value\_if\_false;

**Example:**

int x = 10, y = 20;

int max = (x > y) ? x : y; // max = 20

**H. sizeof Operator**

Returns the size of a type in bytes.

cout << sizeof(int); // usually 4

**I. Type Casting Operators**

int a = 10;

double b = (double)a; // explicit cast

**3. Example Program**

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 3;

cout << "Addition: " << a + b << endl;

cout << "Greater?: " << (a > b) << endl;

cout << "Logical AND: " << (a > 5 && b < 5) << endl;

cout << "Bitwise OR: " << (a | b) << endl;

cout << "Ternary Max: " << ((a > b) ? a : b) << endl;

return 0;

}

**1. I/O in C++**

C++ uses the **iostream** library for console-based input and output.

| **Stream** | **Purpose** | **Object** |
| --- | --- | --- |
| **Output** | Display data to console | cout |
| **Input** | Take data from user | cin |
| **Error Output** | Display errors | cerr |
| **Log Output** | Unbuffered messages | clog |

**2. Output using cout**

* **Syntax:**

cout << data;

* **Example:**

#include <iostream>

using namespace std;

int main() {

cout << "Hello, C++!" << endl; // endl = newline

cout << "Value: " << 42 << "\n"; // \n = newline

return 0;

}

**Key Points:**

* << is the **insertion operator** (sends data to output stream).
* Can chain multiple << operators.

**3. Input using cin**

* **Syntax:**

cin >> variable;

* **Example:**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age; // input from user

cout << "You entered: " << age << endl;

return 0;

}

**Key Points:**

* >> is the **extraction operator** (takes data from input stream).
* Stops reading at a whitespace for string unless you use getline().

**4. Taking a Whole Line with getline()**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name;

cout << "Enter your full name: ";

getline(cin, name); // reads full line including spaces

cout << "Hello, " << name << "!" << endl;

return 0;

}

**5. Error Output with cerr**

* **For error messages** (unbuffered, appears immediately):

cerr << "Error: Invalid input!" << endl;

**6. Logging with clog**

* **For logs/warnings** (buffered output, flushed later):

clog << "Processing data..." << endl;

**7. Example Program (All I/O Streams)**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name;

int age;

cout << "Enter your name: ";

getline(cin, name);

cout << "Enter your age: ";

cin >> age;

cout << "\n--- Output ---\n";

cout << "Name: " << name << "\n";

cout << "Age: " << age << "\n";

clog << "Info: Program executed successfully." << endl;

cerr << "Warning: This is just a demo error message." << endl;

return 0;

}

**1. What are Conditional Statements?**

They allow you to **execute certain parts of the code only when a condition is true**.

C++ supports:

1. if
2. if...else
3. if...else if...else
4. Nested if
5. switch

**2. if Statement**

**Syntax:**

if (condition) {

// code if condition is true

}

**Example:**

int x = 10;

if (x > 5) {

cout << "x is greater than 5\n";

}

**3. if...else Statement**

**Syntax:**

if (condition) {

// executes if true

} else {

// executes if false

}

**Example:**

int age = 16;

if (age >= 18) {

cout << "Eligible to vote\n";

} else {

cout << "Not eligible to vote\n";

}

**4. if...else if...else Ladder**

Used for **multiple conditions**.

int marks = 85;

if (marks >= 90) {

cout << "Grade A\n";

} else if (marks >= 75) {

cout << "Grade B\n";

} else if (marks >= 50) {

cout << "Grade C\n";

} else {

cout << "Fail\n";

}

**5. Nested if**

An if inside another if.

int num = 20;

if (num > 0) {

if (num % 2 == 0) {

cout << "Positive even number\n";

}

}

**6. switch Statement**

Best when checking a single variable against multiple constant values.  
**Syntax:**

switch (expression) {

case value1:

// code

break;

case value2:

// code

break;

default:

// code if no match

}

**Example:**

int day = 3;

switch (day) {

case 1: cout << "Monday\n"; break;

case 2: cout << "Tuesday\n"; break;

case 3: cout << "Wednesday\n"; break;

default: cout << "Invalid day\n";

}

**Rules:**

* switch expression must be int, char, or enum (C++17 allows some constexpr strings in limited contexts).
* break stops execution from falling through to next case.

**7. Conditional (Ternary) Operator**

Shorthand for if...else.

int a = 10, b = 20;

int max = (a > b) ? a : b; // max = 20

**8. Example Program**

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

if (num > 0) {

cout << "Positive\n";

} else if (num < 0) {

cout << "Negative\n";

} else {

cout << "Zero\n";

}

switch (num) {

case 0: cout << "Zero via switch\n"; break;

case 1: cout << "One via switch\n"; break;

default: cout << "Other number via switch\n";

}

return 0;

}

**1. What is a Loop?**

A **loop** is used to execute a block of code **multiple times** until a certain condition becomes false.

**2. Types of Loops in C++**

**A. for Loop**

Best when you **know how many times** to run.  
**Syntax:**

for (initialization; condition; update) {

// code

}

**Example:**

for (int i = 1; i <= 5; i++) {

cout << i << " ";

}

**Output:**

1 2 3 4 5

**B. while Loop**

Runs **while a condition is true**.  
**Syntax:**

while (condition) {

// code

}

**Example:**

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

**C. do...while Loop**

Runs the code **at least once**, then checks condition.  
**Syntax:**

do {

// code

} while (condition);

**Example:**

int i = 1;

do {

cout << i << " ";

i++;

} while (i <= 5);

**D. Range-based for Loop (C++11+)**

Best for iterating **over containers**.  
**Syntax:**

for (data\_type var : container) {

// code

}

**Example:**

#include <vector>

vector<int> nums = {10, 20, 30};

for (int x : nums) {

cout << x << " ";

}

**3. Loop Control Statements**

| **Statement** | **Use** |
| --- | --- |
| break | Exit loop immediately |
| continue | Skip current iteration and move to next |
| goto | Jump to a label (not recommended) |

**Example:**

for (int i = 1; i <= 5; i++) {

if (i == 3) continue; // skip 3

if (i == 5) break; // stop at 5

cout << i << " ";

}

**Output:**

1 2 4

**4. Nested Loops**

A loop inside another loop.

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 2; j++) {

cout << "(" << i << "," << j << ") ";

}

}

**Output:**

(1,1) (1,2) (2,1) (2,2) (3,1) (3,2)

**5. Example Program**

#include <iostream>

using namespace std;

int main() {

// for loop

for (int i = 1; i <= 5; i++) {

cout << i << " ";

}

cout << endl;

// while loop

int j = 1;

while (j <= 5) {

cout << j << " ";

j++;

}

cout << endl;

// do-while loop

int k = 1;

do {

cout << k << " ";

k++;

} while (k <= 5);

return 0;

}

**1. What is a Function?**

A **function** is a block of code that performs a specific task.  
You can **call** the function when needed, instead of repeating code.

**2. Types of Functions**

1. **Built-in functions** → Already provided by C++ (e.g., sqrt(), pow()).
2. **User-defined functions** → Created by the programmer.

**3. Function Syntax**

returnType functionName(parameter1, parameter2, ...) {

// function body

return value; // optional if returnType is void

}

**4. Example of a Simple Function**

#include <iostream>

using namespace std;

// Function declaration

void greet() {

cout << "Hello, C++!" << endl;

}

int main() {

greet(); // Function call

return 0;

}

**Output:**

Hello, C++!

**5. Function with Parameters**

#include <iostream>

using namespace std;

void printSum(int a, int b) {

cout << "Sum = " << (a + b) << endl;

}

int main() {

printSum(5, 3);

return 0;

}

**Output:**

Sum = 8

**6. Function Returning a Value**

#include <iostream>

using namespace std;

int square(int num) {

return num \* num;

}

int main() {

int result = square(4);

cout << "Square = " << result << endl;

return 0;

}

**Output:**

Square = 16

**7. Function Declaration & Definition (Prototype)**

If you define a function **after** main(), you must declare it first.

#include <iostream>

using namespace std;

int square(int); // Function declaration

int main() {

cout << square(5);

return 0;

}

// Function definition

int square(int num) {

return num \* num;

}

**8. Default Parameters**

#include <iostream>

using namespace std;

void greet(string name = "User") {

cout << "Hello, " << name << endl;

}

int main() {

greet(); // Uses default value

greet("Alice"); // Uses given value

return 0;

}

**Output:**

Hello, User

Hello, Alice

**9. Pass by Value vs Pass by Reference**

**Pass by Value (copy is passed)**

void changeValue(int x) {

x = 100;

}

**Pass by Reference (original is changed)**

void changeValue(int &x) {

x = 100;

}

**10. Recursion (Function calling itself)**

#include <iostream>

using namespace std;

int factorial(int n) {

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

int main() {

cout << factorial(5); // 120

return 0;

}

✅ Functions make code **reusable, modular, and easy to maintain**.

**1. What is Recursion?**

Recursion is when a **function calls itself** to solve a problem in smaller steps until it reaches a **base case** (stop condition).

It’s like standing between two mirrors — you see infinite reflections, but eventually, you stop because you can’t see forever.

**2. Structure of a Recursive Function**

Every recursive function has:

1. **Base Case** – stops the recursion.
2. **Recursive Case** – the function calls itself with smaller/simpler input.

**General Syntax:**

returnType functionName(parameters) {

if (base\_condition) {

return base\_result;

}

return functionName(smaller\_problem);

}

**3. Example: Factorial**

Mathematically:

n! = n × (n-1) × (n-2) × ... × 1

Or recursively:

n! = n × (n-1)!

**Code:**

#include <iostream>

using namespace std;

int factorial(int n) {

if (n <= 1) return 1; // Base case

return n \* factorial(n - 1); // Recursive case

}

int main() {

cout << "Factorial of 5 = " << factorial(5) << endl;

return 0;

}

**Output:**

Factorial of 5 = 120

**4. Example: Fibonacci Sequence**

Mathematically:

F(0) = 0, F(1) = 1

F(n) = F(n-1) + F(n-2)

**Code:**

#include <iostream>

using namespace std;

int fibonacci(int n) {

if (n == 0) return 0; // Base case 1

if (n == 1) return 1; // Base case 2

return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case

}

int main() {

for (int i = 0; i < 10; i++)

cout << fibonacci(i) << " ";

return 0;

}

**Output:**

0 1 1 2 3 5 8 13 21 34

**5. Example: Sum of First N Natural Numbers**

#include <iostream>

using namespace std;

int sum(int n) {

if (n == 0) return 0; // Base case

return n + sum(n - 1); // Recursive case

}

int main() {

cout << sum(10); // Output: 55

return 0;

}

**6. How Recursion Works (Call Stack)**

For factorial(3):

factorial(3)

→ 3 \* factorial(2)

→ 2 \* factorial(1)

→ 1 (base case)

**Call Stack:**

factorial(3) → factorial(2) → factorial(1)

Functions are paused until the innermost call finishes, then results are returned **backwards**.

**7. Tail vs Non-Tail Recursion**

* **Tail Recursion** → Recursive call is the last statement.  
  Example:
* void countdown(int n) {
* if (n == 0) return;
* cout << n << " ";
* countdown(n - 1); // last action
* }
* **Non-Tail Recursion** → Work is done after the recursive call.  
  Example:
* void reverseCountdown(int n) {
* if (n == 0) return;
* reverseCountdown(n - 1);
* cout << n << " "; // work after recursion
* }

**8. Pros & Cons**

✅ Clear, elegant solution for repetitive patterns.  
✅ Reduces complex problems into smaller sub-problems.

⚠️ Can be **slow** (overhead of function calls).  
⚠️ Can cause **stack overflow** if base case is missing or too deep.

**1. What is Modular Programming?**

Modular programming is a **program design technique** where a program is divided into **separate, independent modules** (parts), each responsible for a specific functionality.

Think of it like **LEGO blocks** — you build small blocks (modules) and then combine them to create a big program.

**2. Why Use Modular Programming?**

✅ **Readability** – Easier to understand.  
✅ **Reusability** – Use the same module in different projects.  
✅ **Maintainability** – Fix one module without breaking others.  
✅ **Team Work** – Different people can work on different modules at the same time.

**3. Modular Programming in C++**

In C++, we typically separate:

1. **Header files (.h or .hpp)** → contain function declarations, class definitions, constants.
2. **Source files (.cpp)** → contain actual function implementations.
3. **Main program file** → contains main() function and calls the modules.

**4. Example: Modular Calculator**

We’ll make a calculator using three files:

**File 1: calculator.h (Header file – function declarations)**

// calculator.h

#ifndef CALCULATOR\_H

#define CALCULATOR\_H

double add(double a, double b);

double subtract(double a, double b);

double multiply(double a, double b);

double divide(double a, double b);

#endif

**File 2: calculator.cpp (Implementation file – function definitions)**

// calculator.cpp

#include "calculator.h"

double add(double a, double b) {

return a + b;

}

double subtract(double a, double b) {

return a - b;

}

double multiply(double a, double b) {

return a \* b;

}

double divide(double a, double b) {

if (b == 0) return 0; // simple error handling

return a / b;

}

**File 3: main.cpp (Main program – uses the module)**

// main.cpp

#include <iostream>

#include "calculator.h"

using namespace std;

int main() {

double x, y;

cout << "Enter two numbers: ";

cin >> x >> y;

cout << "Addition: " << add(x, y) << endl;

cout << "Subtraction: " << subtract(x, y) << endl;

cout << "Multiplication: " << multiply(x, y) << endl;

cout << "Division: " << divide(x, y) << endl;

return 0;

}

**5. How to Compile Multiple Files**

If you save all files in the same folder:

g++ main.cpp calculator.cpp -o calculator

./calculator

**6. Advantages of This Approach**

* Each .cpp file can be **compiled separately**.
* Changing calculator.cpp doesn’t require changing main.cpp.
* Code becomes **organized** and **scalable**.

**7. Real-World Analogy**

* **Restaurant kitchen** = Each chef (module) specializes in one task.
* **Head chef** = Main program (main.cpp) coordinates all chefs.

Day 2

Arrays

Strings

Pointers

Functions and Pointers

Pointers to Pointers and Dynamic Memory Intro

Structures

Introduction to Data Structures

**1. What is an Array?**

An **array** is a collection of elements of the **same data type** stored in **contiguous memory locations**.

* Each element can be accessed using its **index** (starting from 0).

Think of an array like a row of **mailboxes** with numbers on them — you can quickly access a specific one by its number.

**2. Array Declaration**

dataType arrayName[size];

Example:

int numbers[5]; // array of 5 integers

**3. Initialization**

int numbers[5] = {10, 20, 30, 40, 50}; // fully initialized

int nums[] = {1, 2, 3}; // size automatically determined

int arr[5] = {1, 2}; // remaining elements are 0

**4. Accessing Elements**

cout << numbers[0]; // prints first element

numbers[2] = 100; // updates 3rd element

**5. Example: Basic Array Usage**

#include <iostream>

using namespace std;

int main() {

int marks[5]; // array of size 5

cout << "Enter 5 marks: ";

for (int i = 0; i < 5; i++) {

cin >> marks[i];

}

cout << "You entered: ";

for (int i = 0; i < 5; i++) {

cout << marks[i] << " ";

}

return 0;

}

**Output (example):**

Enter 5 marks: 10 20 30 40 50

You entered: 10 20 30 40 50

**6. Types of Arrays**

**a) One-Dimensional Array**

int arr[4] = {1, 2, 3, 4};

Stored in memory like:

[1][2][3][4]

**b) Two-Dimensional Array (Matrix)**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

Stored in memory like:

Row 0 → 1 2 3

Row 1 → 4 5 6

**c) Multidimensional Arrays**

Can be 3D, 4D, etc., but rarely used directly — usually replaced with vectors or classes.

**7. Example: 2D Array**

#include <iostream>

using namespace std;

int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 3; j++) {

cout << matrix[i][j] << " ";

}

cout << endl;

}

return 0;

}

**Output:**

1 2 3

4 5 6

**8. Key Points**

* **Index starts at 0**.
* Memory is **contiguous** — useful for performance.
* If you go **out of bounds**, you get **undefined behavior** (dangerous!).
* For dynamic size arrays, prefer **std::vector**.

**9. Common Operations**

| **Operation** | **Example** |
| --- | --- |
| Traversing | Loop through array elements |
| Updating | arr[1] = 50; |
| Summing elements | sum += arr[i]; |
| Searching | Linear or binary search |
| Sorting | sort(arr, arr + n); |

**1. What is a String?**

A **string** is a sequence of characters stored together.  
In C++, we have **two main ways** to work with strings:

1. **C-style strings** → arrays of characters ending with a '\0' (null character).
2. **std::string class** (modern way) → easier to use, comes from <string>.

**2. C-Style Strings**

**Declaration**

char name[6] = "Alice"; // 5 chars + '\0'

* If you forget space for the '\0', behavior becomes undefined.
* You must use #include <cstring> for functions like strlen(), strcpy(), etc.

**Example**

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

strcat(str1, str2); // concatenate

cout << "Concatenated: " << str1 << endl;

cout << "Length: " << strlen(str1) << endl;

return 0;

}

**Output:**

Concatenated: HelloWorld

Length: 10

**3. Modern C++ Strings (std::string)**

* Safer, easier to manipulate.
* Part of the C++ Standard Library.
* Automatically handles size and '\0'.

**Declaration**

#include <string>

string name = "Alice";

**4. Basic Operations**

#include <iostream>

#include <string>

using namespace std;

int main() {

string firstName = "John";

string lastName = "Doe";

// Concatenation

string fullName = firstName + " " + lastName;

cout << "Full name: " << fullName << endl;

// Length

cout << "Length: " << fullName.length() << endl;

// Access character

cout << "First letter: " << fullName[0] << endl;

// Modify

fullName[0] = 'M';

cout << "Modified: " << fullName << endl;

return 0;

}

**Output:**

Full name: John Doe

Length: 8

First letter: J

Modified: Mohn Doe

**5. Common String Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| length() / size() | Returns number of characters | s.length() |
| append(str) | Adds to end of string | s.append("World") |
| substr(pos, len) | Extract substring | s.substr(0, 4) |
| find(str) | Finds substring position | s.find("abc") |
| erase(pos, len) | Removes part of string | s.erase(2, 3) |
| insert(pos, str) | Inserts string at position | s.insert(3, "xyz") |

**6. Example: Palindrome Check**

#include <iostream>

#include <string>

using namespace std;

int main() {

string str;

cout << "Enter a string: ";

cin >> str;

string rev = string(str.rbegin(), str.rend());

if (str == rev)

cout << "Palindrome" << endl;

else

cout << "Not a palindrome" << endl;

return 0;

}

**7. Key Points**

* Prefer **std::string** over **C-style strings** for safety.
* std::string can be used directly with I/O streams (cin, cout).
* For reading sentences with spaces → use getline(cin, str).

**1. What is a Pointer?**

A **pointer** is a variable that stores the **memory address** of another variable.  
Think of it as a *"GPS coordinate"* for where the data lives in memory.

**2. Declaring a Pointer**

int x = 10;

int\* ptr = &x; // ptr stores address of x

* & → “address of” operator.
* \* (in declaration) → means the variable is a pointer.

**3. Accessing Data**

#include <iostream>

using namespace std;

int main() {

int x = 10;

int\* p = &x;

cout << "Value of x: " << x << endl; // 10

cout << "Address of x: " << &x << endl; // memory address

cout << "Pointer p stores: " << p << endl; // same as &x

cout << "Value via pointer: " << \*p << endl; // 10

return 0;

}

* \*p → **dereference operator** → gets the value stored at the address.

**4. Changing Value Through Pointer**

int x = 5;

int\* p = &x;

\*p = 20; // changes x directly

cout << x; // 20

A pointer lets you modify the original variable it points to.

**5. Pointer to Different Data Types**

int\* p1; // pointer to int

double\* p2; // pointer to double

char\* p3; // pointer to char

* The type tells the compiler how many bytes to read when dereferencing.

**6. Null Pointer**

A pointer that points to **nothing**.

int\* p = nullptr; // safe to initialize empty pointer

**7. Pointer Arithmetic**

You can increment/decrement pointers:

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

int\* p = arr; // points to arr[0]

cout << \*p << endl; // 10

p++; // now points to arr[1]

cout << \*p << endl; // 20

⚠ Moves based on **size of type** (for int\*, moves 4 bytes each step on most systems).

**8. Pointers and Arrays**

Arrays automatically decay to pointers in most contexts:

int arr[3] = {1, 2, 3};

int\* p = arr;

cout << p[0] << " " << p[1]; // 1 2

**9. Pointer to Pointer**

You can have a pointer to another pointer.

int x = 42;

int\* p = &x;

int\*\* pp = &p;

cout << \*\*pp; // 42

**10. Dynamic Memory Allocation**

* **new** → allocates memory.
* **delete** → frees memory.

int\* p = new int; // allocates memory for int

\*p = 100;

cout << \*p;

delete p; // free memory

For arrays:

int\* arr = new int[5];

delete[] arr; // must use [] for arrays

**11. Example: Swapping Values Using Pointers**

#include <iostream>

using namespace std;

void swapValues(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 5, y = 10;

swapValues(&x, &y);

cout << "x: " << x << ", y: " << y;

return 0;

}

**Output:**

x: 10, y: 5

✅ **Key Takeaways**

* A pointer stores a memory address.
* Use & to get address, \* to get value at address.
* Always initialize pointers (nullptr if empty).
* Remember to delete memory allocated with new.

**1. Passing Pointers to Functions**

A function can take a pointer as an argument to work directly with the original variable.

**Example: Change a Value in a Function**

#include <iostream>

using namespace std;

void updateValue(int\* ptr) {

\*ptr = \*ptr + 5; // change original variable

}

int main() {

int x = 10;

updateValue(&x); // pass address

cout << "Updated x: " << x; // 15

}

**Key Idea** → Passing a pointer allows the function to modify the original variable.

**2. Swapping Values Using Pointers**

void swapValues(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

Call with:

swapValues(&x, &y);

**3. Array and Pointer Parameters**

Since arrays decay to pointers when passed to a function, you can handle arrays via pointers.

void printArray(int\* arr, int size) {

for(int i = 0; i < size; i++)

cout << arr[i] << " ";

}

int main() {

int nums[] = {1, 2, 3, 4};

printArray(nums, 4); // nums is treated as int\*

}

**4. Returning a Pointer from a Function**

⚠ **Warning**: Never return a pointer to a local variable (it goes out of scope).

**Safe Way** → Use new (caller must delete later):

int\* createNumber() {

int\* p = new int(42);

return p;

}

int main() {

int\* num = createNumber();

cout << \*num; // 42

delete num;

}

**5. Pointer to a Function**

You can store the address of a function in a pointer and call it.

**Syntax:**

returnType (\*pointerName)(parameterTypes);

**Example:**

#include <iostream>

using namespace std;

void greet() {

cout << "Hello from function pointer!\n";

}

int main() {

void (\*funcPtr)() = greet; // store address

funcPtr(); // call via pointer

}

**6. Passing a Function Pointer to Another Function**

This allows **callbacks**.

#include <iostream>

using namespace std;

void sayHello() { cout << "Hello!\n"; }

void sayBye() { cout << "Goodbye!\n"; }

void runFunction(void (\*func)()) {

func(); // call the passed function

}

int main() {

runFunction(sayHello);

runFunction(sayBye);

}

**7. Function Pointers with Parameters**

#include <iostream>

using namespace std;

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

int multiply(int a, int b) { return a \* b; }

int main() {

int (\*op)(int, int);

op = add;

cout << "Sum: " << op(5, 3) << endl; // 8

op = multiply;

cout << "Product: " << op(5, 3) << endl; // 15

}

**8. Modern Alternative → std::function**

Instead of raw function pointers, C++11+ offers **type-safe callable wrappers**:

#include <iostream>

#include <functional>

using namespace std;

void greet() { cout << "Hi!\n"; }

int main() {

function<void()> f = greet;

f();

}

Much safer and more flexible (works with lambdas too).

✅ **Key Takeaways**

* Pointers allow functions to modify original data.
* Arrays passed to functions are essentially pointers.
* You can store and pass around function addresses.
* Function pointers enable callbacks and strategy-like behavior.
* Modern C++ prefers std::function for safety.

**1. Pointer to a Pointer (int \*\*)**

A **pointer to a pointer** stores the address of another pointer, which in turn stores the address of a variable.  
Think: variable → pointer → pointer to pointer.

**Example:**

#include <iostream>

using namespace std;

int main() {

int value = 42;

int\* p = &value; // pointer to int

int\*\* pp = &p; // pointer to pointer to int

cout << "Value: " << value << endl; // 42

cout << "Via pointer: " << \*p << endl; // 42

cout << "Via pointer to pointer: " << \*\*pp << endl; // 42

return 0;

}

**Memory Structure**

value (42) <-- \*p <-- \*\*pp

↑ address of value ↑ address of p

p -------------------

↑ address of p

pp -------------------

**2. Why Use Pointers to Pointers?**

* **Dynamic 2D arrays** (int\*\* matrix)
* **Modifying a pointer itself** inside a function
* **Linked lists and complex data structures** (when you need to update head pointer)
* **Command-line arguments** (char \*\*argv in main())

**3. Passing Pointer to Pointer to a Function**

This allows changing the pointer itself.

#include <iostream>

using namespace std;

void allocateMemory(int\*\* ptr) {

\*ptr = new int(99); // allocate and assign

}

int main() {

int\* p = nullptr;

allocateMemory(&p);

cout << "Allocated value: " << \*p << endl; // 99

delete p; // cleanup

}

**4. Introduction to Dynamic Memory in C++**

By default, variables are stored in:

* **Stack** → automatically managed, fixed size
* **Heap** → manually managed, dynamic size

**Dynamic memory** = you request memory at runtime (using new) and release it with delete.

**4.1 Allocating and Releasing a Single Variable**

int\* p = new int; // allocate memory for 1 int

\*p = 10;

delete p; // free memory

**4.2 Allocating and Releasing an Array**

int\* arr = new int[5]; // 5 ints in heap

for (int i = 0; i < 5; i++)

arr[i] = i + 1;

delete[] arr; // must use delete[] for arrays

**4.3 Dynamic 2D Array with Pointers to Pointers**

int rows = 3, cols = 4;

int\*\* matrix = new int\*[rows];

for (int i = 0; i < rows; i++)

matrix[i] = new int[cols];

// Assign values

for (int i = 0; i < rows; i++)

for (int j = 0; j < cols; j++)

matrix[i][j] = (i + 1) \* (j + 1);

// Cleanup

for (int i = 0; i < rows; i++)

delete[] matrix[i];

delete[] matrix;

**4.4 Common Pitfalls**

* **Memory leaks**: Forgetting to delete allocated memory
* **Dangling pointers**: Using pointers after delete
* **Double deletion**: Calling delete twice on same pointer

✅ **Key Takeaways**

* Pointer to pointer = chain of references to a value.
* Useful in multi-level data structures and when modifying pointers inside functions.
* new / delete → manage heap memory manually.
* Always match new with delete and new[] with delete[].

**1. What is a Structure?**

A **structure** (struct) in C++ is a user-defined data type that groups **different types of variables** under one name.  
It’s like a **custom container** for related data.

**2. Basic Syntax**

#include <iostream>

using namespace std;

struct Student {

int rollNo;

string name;

float marks;

};

int main() {

Student s1; // Create a Student variable

s1.rollNo = 101;

s1.name = "Alice";

s1.marks = 89.5;

cout << "Roll No: " << s1.rollNo << endl;

cout << "Name: " << s1.name << endl;

cout << "Marks: " << s1.marks << endl;

}

**3. Initializing Structures**

Student s2 = {102, "Bob", 75.4}; // Direct initialization

**4. Accessing Members**

* **Dot operator (.)** → for normal variables
* **Arrow operator (->)** → for pointers to structs

Student s3 = {103, "Charlie", 92.0};

Student\* ptr = &s3;

cout << ptr->name << endl; // Charlie

**5. Array of Structures**

Student arr[2] = {

{201, "David", 88.0},

{202, "Eva", 91.5}

};

for (int i = 0; i < 2; i++) {

cout << arr[i].name << " scored " << arr[i].marks << endl;

}

**6. Structure with Functions**

Structures can have **member functions** (just like classes).

struct Employee {

int id;

string name;

double salary;

void display() {

cout << id << " - " << name << " : " << salary << endl;

}

};

int main() {

Employee e = {1, "John", 50000};

e.display();

}

**7. Pointer to Structure**

Employee e2 = {2, "Mary", 55000};

Employee\* ptr = &e2;

ptr->salary = 60000; // Modify via pointer

ptr->display();

**8. Structures vs Classes in C++**

| **Feature** | **struct** | **class** |
| --- | --- | --- |
| Default Access | public | private |
| Inheritance | Yes | Yes |
| Functions Allowed | Yes | Yes |

✅ **Key Points**

* Use structures to group **related variables of different types**.
* Access members via **.** or **->** (pointer).
* In modern C++, structs and classes are almost the same except **default access level**.

**1. What is a Data Structure?**

A **data structure** is a way of organizing, managing, and storing data so it can be used efficiently.

Think of it like **different storage containers**:

* A **stack** is like a pile of plates (last in, first out).
* A **queue** is like a line at a ticket counter (first in, first out).
* A **linked list** is like a chain where each link points to the next.
* A **tree** is like a family tree with parent-child relationships.

**2. Why Use Data Structures?**

* **Efficiency** → Faster searching, insertion, deletion.
* **Organization** → Keep related data together.
* **Scalability** → Handle large amounts of data without performance loss.
* **Reusability** → Once implemented, can be used anywhere.

**3. Categories of Data Structures**

**A. Primitive (Basic) Data Structures**

Directly supported by C++:

* int, float, char, bool, double

**B. Non-Primitive Data Structures**

Organized into two main types:

**1. Linear Data Structures**

Data is arranged **sequentially**.

* **Array** – Fixed-size, indexed storage.
* **Linked List** – Elements linked using pointers.
* **Stack** – LIFO structure.
* **Queue** – FIFO structure.
* **Deque** – Double-ended queue.

**2. Non-Linear Data Structures**

Data is arranged in **hierarchical or network form**.

* **Trees** – Hierarchical relationships (e.g., Binary Tree, BST).
* **Graphs** – Nodes connected by edges (e.g., social networks).

**4. Operations on Data Structures**

* **Traversal** – Visiting elements.
* **Insertion** – Adding elements.
* **Deletion** – Removing elements.
* **Searching** – Finding elements.
* **Sorting** – Arranging in order.
* **Merging** – Combining datasets.

**5. Real-Life Analogy**

Imagine managing a **library**:

* **Array** → Shelf with fixed number of books.
* **Linked List** → Chain of book references.
* **Stack** → Books piled on top of each other.
* **Queue** → People waiting for issuing books.
* **Tree** → Book categories and subcategories.
* **Graph** → Network of related topics.

**6. Example in C++**

#include <iostream>

using namespace std;

int main() {

// Array example

int numbers[5] = {10, 20, 30, 40, 50};

cout << "Array elements: ";

for (int i = 0; i < 5; i++) {

cout << numbers[i] << " ";

}

return 0;

}

✅ **Key Takeaways**

* Data structures are the **building blocks** of efficient programming.
* Choosing the right data structure **depends on the problem**.
* Understanding them is essential before moving into **algorithms**.

Day 3

Dynamic Memory Management

Classes and Objects

Constructors and Destructors

Inheritance

Polymorphism

Encapsulation and Abstraction

New, new [], new dynamic

**1. What is Dynamic Memory Management?**

* In C++, **memory** can be allocated either:
  1. **Compile-time (Static)** → Size fixed before program runs (e.g., int arr[10]).
  2. **Run-time (Dynamic)** → Size decided while program runs (e.g., based on user input).

Dynamic memory allows:

* Creating variables, arrays, or objects **on the fly**.
* Using only the memory you need.
* Avoiding waste when size is unknown at compile-time.

**2. Memory Areas in C++**

* **Stack** → Stores local variables, automatically managed.
* **Heap (Free Store)** → Stores dynamically allocated memory, **manually managed** by the programmer.

**3. Operators for Dynamic Memory**

**A. new Operator**

Allocates memory **from the heap** and returns a pointer.

int\* ptr = new int; // allocates memory for one int

\*ptr = 10; // assign value

**B. delete Operator**

Frees the memory allocated with new.

delete ptr;

ptr = nullptr; // good practice to avoid dangling pointer

**4. Allocating Arrays Dynamically**

int size;

cout << "Enter array size: ";

cin >> size;

int\* arr = new int[size]; // dynamically allocated array

for (int i = 0; i < size; i++) {

arr[i] = i + 1;

}

cout << "Array elements: ";

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

delete[] arr; // free memory

arr = nullptr;

⚠ Use delete[] when freeing arrays.

**5. Allocating for Objects**

class Student {

public:

string name;

Student(string n) : name(n) {}

void display() { cout << "Name: " << name << endl; }

};

Student\* s = new Student("Alice");

s->display();

delete s;

**6. Common Problems**

* **Memory Leak** → Not freeing allocated memory.
* **Dangling Pointer** → Using a pointer after freeing it.
* **Double Delete** → Deleting the same memory twice.

**7. Best Practices**

* Always pair new with delete and new[] with delete[].
* Set pointers to nullptr after deleting.
* Consider **smart pointers** (std::unique\_ptr, std::shared\_ptr) to automate cleanup.

**8. Example: Dynamic 2D Array**

#include <iostream>

using namespace std;

int main() {

int rows, cols;

cout << "Enter rows and columns: ";

cin >> rows >> cols;

int\*\* matrix = new int\*[rows]; // array of row pointers

for (int i = 0; i < rows; i++) {

matrix[i] = new int[cols]; // allocate each row

}

// Fill the matrix

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

matrix[i][j] = i + j;

}

}

// Display the matrix

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

cout << matrix[i][j] << " ";

}

cout << endl;

}

// Free memory

for (int i = 0; i < rows; i++) {

delete[] matrix[i];

}

delete[] matrix;

}

✅ **Key Takeaways**

* new → Allocate memory at runtime.
* delete → Free memory.
* Always avoid **leaks** and **dangling pointers**.
* Smart pointers in modern C++ make memory management safer.

**1. What is a Class?**

A **class** is a **blueprint** for creating objects.  
It defines **properties** (data members) and **behaviors** (member functions) in **one unit**.

Think:  
📜 *Class = Recipe*  
🍪 *Object = Cookie made from that recipe*

**2. What is an Object?**

An **object** is an **instance of a class**.  
You can create multiple objects from the same class, each with its own copy of the data members.

**3. Basic Syntax**

#include <iostream>

using namespace std;

// Class definition

class Car {

public: // access specifier

string brand; // data member

int speed; // data member

void display() { // member function

cout << "Brand: " << brand << ", Speed: " << speed << " km/h" << endl;

}

};

int main() {

Car c1; // object creation

c1.brand = "Tesla";

c1.speed = 120;

c1.display();

Car c2; // another object

c2.brand = "BMW";

c2.speed = 150;

c2.display();

}

**4. Access Specifiers**

* **public** → Accessible from anywhere.
* **private** → Accessible only inside the class (default).
* **protected** → Accessible in the class and its derived classes.

Example:

class Example {

private:

int x; // only class methods can access

public:

void setX(int val) { x = val; }

int getX() { return x; }

};

**5. Constructors**

A **constructor** is a special function that initializes objects automatically when created.

* Same name as the class
* No return type
* Can be **default**, **parameterized**, or **copy constructor**

Example:

class Student {

public:

string name;

int age;

// Default constructor

Student() {

name = "Unknown";

age = 0;

}

// Parameterized constructor

Student(string n, int a) {

name = n;

age = a;

}

void display() {

cout << name << " (" << age << " years old)" << endl;

}

};

int main() {

Student s1; // calls default constructor

s1.display();

Student s2("Alice", 20); // calls parameterized constructor

s2.display();

}

**6. Destructors**

A **destructor** cleans up resources when an object is destroyed.

* Same name as the class but prefixed with ~
* No parameters, no return type
* Automatically called when object goes out of scope

Example:

class Demo {

public:

Demo() { cout << "Object created\n"; }

~Demo() { cout << "Object destroyed\n"; }

};

int main() {

Demo d; // constructor called

} // destructor automatically called here

**7. this Pointer**

Inside a class, this refers to the **current object**.

Example:

class Box {

public:

int length;

Box(int length) {

this->length = length; // differentiate between parameter and data member

}

};

**8. Objects with Dynamic Memory**

You can allocate objects using new and deallocate with delete.

Car\* c = new Car();

c->brand = "Audi";

c->speed = 200;

c->display();

delete c;

**9. Combining Concepts**

#include <iostream>

using namespace std;

class Employee {

private:

string name;

double salary;

public:

Employee(string n, double s) {

name = n;

salary = s;

}

void giveRaise(double percent) {

salary += salary \* (percent / 100);

}

void display() {

cout << "Name: " << name << ", Salary: " << salary << endl;

}

};

int main() {

Employee e1("John", 50000);

e1.display();

e1.giveRaise(10);

e1.display();

}

**Key Takeaways**

* **Class** = blueprint, **Object** = instance.
* Use **access specifiers** to control visibility.
* **Constructors** auto-initialize, **Destructors** auto-cleanup.
* Objects can be **stack-based** (automatic) or **heap-based** (dynamic).
* Always pair new with delete to avoid memory leaks.

**1. Constructor in C++**

A **constructor** is a special member function of a class that **automatically executes when an object is created**.  
It is mainly used to **initialize objects**.

**Key Points**

* Same name as the class.
* No return type (not even void).
* Can be **overloaded** (multiple constructors with different parameters).
* Automatically called when an object is created.

**2. Types of Constructors**

**A. Default Constructor**

No parameters.

#include <iostream>

using namespace std;

class Student {

public:

string name;

int age;

// Default constructor

Student() {

name = "Unknown";

age = 0;

}

void display() {

cout << name << " (" << age << " years old)" << endl;

}

};

int main() {

Student s1; // default constructor called

s1.display();

}

**Output:**

Unknown (0 years old)

**B. Parameterized Constructor**

Takes arguments to initialize data members.

class Student {

public:

string name;

int age;

Student(string n, int a) { // parameterized constructor

name = n;

age = a;

}

void display() {

cout << name << " (" << age << " years old)" << endl;

}

};

int main() {

Student s2("Alice", 20);

s2.display();

}

**Output:**

Alice (20 years old)

**C. Copy Constructor**

Creates a new object as a **copy of an existing object**.

class Student {

public:

string name;

int age;

Student(string n, int a) {

name = n;

age = a;

}

// Copy constructor

Student(const Student &s) {

name = s.name;

age = s.age;

}

void display() {

cout << name << " (" << age << " years old)" << endl;

}

};

int main() {

Student s1("Bob", 22);

Student s2 = s1; // copy constructor called

s2.display();

}

**Output:**

Bob (22 years old)

**3. Destructor in C++**

A **destructor** is a special member function that **automatically executes when an object goes out of scope**.  
It is mainly used to **clean up resources** like memory or file handles.

**Key Points**

* Same name as class, prefixed with ~.
* No parameters and no return type.
* Automatically called once per object.

**4. Example of Destructor**

#include <iostream>

using namespace std;

class Demo {

public:

Demo() { cout << "Constructor called\n"; }

~Demo() { cout << "Destructor called\n"; }

};

int main() {

Demo d1; // constructor called

Demo d2; // constructor called

} // destructor called for d2 and then d1

**Output:**

Constructor called

Constructor called

Destructor called

Destructor called

**5. Dynamic Objects and Destructor**

For objects allocated with new, destructor is called only with delete.

Demo\* d = new Demo(); // constructor called

delete d; // destructor called

**6. Constructor Overloading Example**

#include <iostream>

using namespace std;

class Rectangle {

public:

int length, width;

Rectangle() { length = 0; width = 0; } // default

Rectangle(int l, int w) { length = l; width = w; } // parameterized

void area() { cout << "Area: " << length \* width << endl; }

};

int main() {

Rectangle r1; // default

Rectangle r2(5, 3); // parameterized

r1.area();

r2.area();

}

**Output:**

Area: 0

Area: 15

**7. Key Takeaways**

* **Constructor** → initialize object; automatically called when object is created.
* **Destructor** → cleanup resources; automatically called when object is destroyed.
* Can overload constructors for flexibility.
* Copy constructor allows creating object as a copy of another.
* Destructors are essential when using **dynamic memory** (new/delete) to prevent **memory leaks**.

**1. What is Inheritance?**

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

Think:

* **Base Class** → Parent, provides common features
* **Derived Class** → Child, reuses base features and can add new ones

**2. Why Use Inheritance?**

* **Reusability** → Avoid writing the same code again.
* **Extensibility** → Add or override features in derived classes.
* **Hierarchy Representation** → Model real-world relationships.

**3. Syntax**

class Base {

// base class members

};

class Derived : access\_specifier Base {

// derived class members

};

**Access Specifiers for Inheritance:**

* public → public members remain public, protected remain protected
* protected → public & protected members become protected
* private → public & protected members become private

**4. Simple Example**

#include <iostream>

using namespace std;

// Base class

class Vehicle {

public:

string brand = "Generic Vehicle";

void honk() { cout << "Beep beep!\n"; }

};

// Derived class

class Car : public Vehicle {

public:

string model = "Sedan";

};

int main() {

Car myCar;

myCar.honk(); // inherited from Vehicle

cout << myCar.brand << " " << myCar.model << endl;

}

**Output:**

Beep beep!

Generic Vehicle Sedan

**5. Types of Inheritance**

| **Type** | **Description** |
| --- | --- |
| **Single** | One derived class from one base class |
| **Multiple** | One derived class inherits from multiple base classes |
| **Multilevel** | Chain of inheritance (grandparent → parent → child) |
| **Hierarchical** | Multiple derived classes inherit from one base class |
| **Hybrid** | Combination of two or more types |

**A. Multiple Inheritance**

class Engine {

public:

void start() { cout << "Engine started\n"; }

};

class Wheels {

public:

void rotate() { cout << "Wheels rotating\n"; }

};

class Car : public Engine, public Wheels {

};

int main() {

Car myCar;

myCar.start();

myCar.rotate();

}

**B. Multilevel Inheritance**

class Animal {

public:

void eat() { cout << "Eating\n"; }

};

class Mammal : public Animal {

public:

void breathe() { cout << "Breathing\n"; }

};

class Dog : public Mammal {

public:

void bark() { cout << "Barking\n"; }

};

int main() {

Dog d;

d.eat();

d.breathe();

d.bark();

}

**6. Access Specifier Effects**

class Base {

public: int x;

protected: int y;

private: int z;

};

class Derived : public Base {

void show() {

cout << x; // OK (public)

cout << y; // OK (protected)

// cout << z; // Error (private)

}

};

**7. Constructors and Inheritance**

* Base class constructor executes **first**, then derived constructor.

class Base {

public:

Base() { cout << "Base constructor\n"; }

};

class Derived : public Base {

public:

Derived() { cout << "Derived constructor\n"; }

};

int main() {

Derived d;

}

**Output:**

Base constructor

Derived constructor

* **Parameterized base constructor**

class Base {

public:

Base(int x) { cout << "Base: " << x << endl; }

};

class Derived : public Base {

public:

Derived(int x) : Base(x) { cout << "Derived: " << x << endl; }

};

**8. Key Takeaways**

* Inheritance = code **reusability and hierarchy**.
* Use **access specifiers** carefully (public, protected, private).
* Constructors of base class execute **before** derived class.
* Supports **single, multiple, multilevel, hierarchical, hybrid** inheritance.
* Avoid ambiguity in multiple inheritance with **virtual inheritance** if needed.

**1. What is Polymorphism?**

**Polymorphism** literally means **“many forms”**.  
In C++, it allows a **single interface** (function or operator) to behave differently based on **context or object type**.

Two types of polymorphism:

1. **Compile-time (Static)** → Function overloading, operator overloading
2. **Runtime (Dynamic)** → Function overriding with virtual keyword

**2. Compile-Time Polymorphism**

**A. Function Overloading**

Same function name, different parameters.

#include <iostream>

using namespace std;

class Calculator {

public:

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

double add(double a, double b) { return a + b; }

};

int main() {

Calculator calc;

cout << calc.add(5, 3) << endl; // 8

cout << calc.add(2.5, 3.1) << endl; // 5.6

}

**B. Operator Overloading**

Redefine operators for user-defined types.

class Point {

public:

int x, y;

Point(int a, int b) { x = a; y = b; }

// Overload +

Point operator+(Point p) {

return Point(x + p.x, y + p.y);

}

};

int main() {

Point p1(1, 2), p2(3, 4);

Point p3 = p1 + p2;

cout << p3.x << ", " << p3.y; // 4, 6

}

**3. Runtime Polymorphism**

**A. Function Overriding**

* Occurs in **derived class** with **same function name and signature** as base class.
* **Virtual functions** are used to achieve **dynamic dispatch**.

#include <iostream>

using namespace std;

class Base {

public:

virtual void show() { // virtual enables runtime polymorphism

cout << "Base class\n";

}

};

class Derived : public Base {

public:

void show() override {

cout << "Derived class\n";

}

};

int main() {

Base\* b;

Derived d;

b = &d;

b->show(); // Calls Derived class function at runtime

}

**Output:**

Derived class

**B. Virtual Functions**

* Declared using virtual in base class.
* Enable **method overriding** in derived classes.
* Enables **late binding** at runtime.

**C. Pure Virtual Functions (Abstract Class)**

* Declared using = 0
* Makes class **abstract** → cannot instantiate directly.

class Shape {

public:

virtual void draw() = 0; // pure virtual

};

class Circle : public Shape {

public:

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

};

int main() {

Shape\* s = new Circle();

s->draw(); // Drawing Circle

delete s;

}

**4. Key Takeaways**

* **Polymorphism** = ability of one function/operator to behave differently.
* **Compile-time** → Function & operator overloading
* **Runtime** → Function overriding using virtual
* **Abstract classes** and **pure virtual functions** enforce method implementation in derived classes.
* Crucial for **extensibility, code reuse, and flexible design**.

**1. Encapsulation**

**Definition**

Encapsulation is the process of **binding data (variables) and functions (methods) together** into a single unit, usually a **class**, and **restricting direct access to some of its components**.

**Key Idea**

* Hide internal details.
* Access data **only through public methods** (getters/setters).

**Syntax Example**

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // hidden from outside

public:

void setBalance(double b) {

if (b >= 0) balance = b;

else cout << "Invalid balance!\n";

}

double getBalance() {

return balance;

}

};

int main() {

BankAccount account;

account.setBalance(1000);

cout << "Balance: " << account.getBalance() << endl;

}

**Output:**

Balance: 1000

**Advantages**

* Protects data from invalid access.
* Reduces code complexity.
* Improves maintainability.

**2. Abstraction**

**Definition**

Abstraction is the process of **hiding unnecessary details** and **showing only essential features** of an object.

Think:

* You **know how to drive a car** (interface)
* You **don’t need to know the engine mechanics** (implementation hidden)

**How Achieved in C++**

1. **Abstract Classes** → Classes with at least one **pure virtual function**.
2. **Interfaces** (using abstract classes) → Define **what** a class should do, not **how**.

**Example of Abstraction**

#include <iostream>

using namespace std;

class Shape { // Abstract class

public:

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

};

class Circle : public Shape {

public:

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

};

class Rectangle : public Shape {

public:

void draw() override { cout << "Drawing Rectangle\n"; }

};

int main() {

Shape\* s1 = new Circle();

Shape\* s2 = new Rectangle();

s1->draw(); // Drawing Circle

s2->draw(); // Drawing Rectangle

delete s1;

delete s2;

}

**Output:**

Drawing Circle

Drawing Rectangle

**3. Differences Between Encapsulation and Abstraction**

| **Feature** | **Encapsulation** | **Abstraction** |
| --- | --- | --- |
| Purpose | Hide internal data | Hide implementation details |
| Achieved by | Access specifiers (private, public) | Abstract classes, pure virtual functions |
| Focus | Data protection | Showing only essential features |
| Example | Getters and setters | Abstract class Shape with draw() |

**4. Key Takeaways**

* **Encapsulation** = protects **data** and ensures controlled access.
* **Abstraction** = hides **implementation details** and shows only what is necessary.
* Both work together to **improve modularity, readability, and maintainability**.
* Abstract classes and virtual functions are the **main tools for abstraction** in C++.

**1. new Operator (Single Object)**

* Allocates **memory for a single object** on the heap.
* Returns a **pointer** to the allocated memory.
* Must be deallocated using delete.

**Example:**

#include <iostream>

using namespace std;

int main() {

int\* p = new int; // allocate one int

\*p = 42; // assign value

cout << "Value: " << \*p << endl; // 42

delete p; // free memory

p = nullptr; // avoid dangling pointer

}

**2. new[] Operator (Array Allocation)**

* Allocates **memory for an array** of objects or variables.
* Must be deallocated using delete[].

**Example:**

int size = 5;

int\* arr = new int[size]; // allocate array of 5 ints

for (int i = 0; i < size; i++)

arr[i] = i + 1;

for (int i = 0; i < size; i++)

cout << arr[i] << " "; // 1 2 3 4 5

delete[] arr; // free array memory

arr = nullptr;

**3. new for Objects**

You can also allocate **class objects dynamically**.

class Student {

public:

string name;

int age;

void display() {

cout << name << " (" << age << " years old)\n";

}

};

int main() {

Student\* s = new Student(); // dynamic object

s->name = "Alice";

s->age = 20;

s->display();

delete s; // free memory

s = nullptr;

}

**4. new for Arrays of Objects**

int n = 3;

Student\* students = new Student[n]; // array of 3 objects

students[0].name = "Bob"; students[0].age = 21;

students[1].name = "Charlie"; students[1].age = 22;

students[2].name = "David"; students[2].age = 23;

for (int i = 0; i < n; i++)

students[i].display();

delete[] students; // free array memory

students = nullptr;

**5. Using new with Parameterized Constructors**

class Rectangle {

int length, width;

public:

Rectangle(int l, int w) { length = l; width = w; }

void area() { cout << "Area: " << length \* width << endl; }

};

int main() {

Rectangle\* r = new Rectangle(5, 3); // dynamic object

r->area(); // Area: 15

delete r;

}

**6. Key Points / Differences**

| **Feature** | **new** | **new[]** |
| --- | --- | --- |
| Usage | Single variable/object | Array of variables/objects |
| Deallocation | delete ptr; | delete[] ptr; |
| Returns | Pointer to object | Pointer to first element |

* Always **match new with delete** and **new[] with delete[]**.
* Helps manage memory **dynamically** when sizes are unknown at compile-time.

Day 4

Boost Libraries Overview

Introduction to Boost: philosophy and modularity

Boost Smart Pointers

Boost Filesystem

Boost Asio (intro only)

Boost Regex

Boost Bind and Lambda

Templates and STL (Standard Template Library)

Function templates vs class templates

Variable Template

Template specialization & partial specialization

Variadic templates

SFINAE and enable\_if

Type traits and decltype, auto

Containers: vector, list, map, unordered\_map, set

Iterators and algorithms

Functors, lambdas, and std::function

Smart pointers (unique\_ptr, shared\_ptr)

Exception handling in STL

**1. What is Boost?**

* **Boost** is a set of **peer-reviewed, portable C++ libraries**.
* Designed to **extend functionality of C++**, often used before new features are added to the standard library.
* Many Boost libraries are now part of **C++11/14/17/20** standards.

**2. Key Features**

* **High-quality**: Peer-reviewed and widely tested.
* **Cross-platform**: Works on Windows, Linux, macOS.
* **Modular**: Use only the libraries you need.
* **Standards-driven**: Many libraries inspired C++ standard features (e.g., std::shared\_ptr, std::regex).

**3. Installation**

1. Download Boost: https://www.boost.org/
2. Unzip or build libraries (for compiled ones).
3. Include Boost in your project:
   * Header-only libraries → just include headers.
   * Compiled libraries → link with .lib (Windows) or .a/.so (Linux).
4. Configure **include path** and **linker** in Visual Studio or your IDE.

**4. Categories of Boost Libraries**

**A. Smart Pointers**

* boost::shared\_ptr, boost::scoped\_ptr, boost::weak\_ptr
* Manage memory automatically, safer than raw pointers.

**B. Containers**

* boost::array, boost::unordered\_map, boost::multi\_array
* Useful when STL containers are not enough.

**C. Algorithms and Utilities**

* boost::algorithm → string algorithms, case conversion, trimming.
* boost::lexical\_cast → safe type conversions.
* boost::optional → represents optional values like null.

**D. Date & Time**

* boost::date\_time → work with dates, times, durations, and time zones.

**E. Regular Expressions**

* boost::regex → powerful regex support, often more flexible than std::regex.

**F. Filesystem**

* boost::filesystem → cross-platform file system operations.

**G. Threads and Synchronization**

* boost::thread → portable multi-threading support.
* boost::mutex, boost::condition\_variable → thread synchronization.

**H. Serialization**

* boost::serialization → save and load C++ objects to/from files or streams.

**I. Graph Library**

* boost::graph → supports complex graph structures and algorithms.

**5. Example: Using boost::shared\_ptr**

#include <iostream>

#include <boost/shared\_ptr.hpp>

using namespace std;

using namespace boost;

class Test {

public:

void show() { cout << "Hello from Test\n"; }

};

int main() {

shared\_ptr<Test> ptr(new Test()); // automatic memory management

ptr->show();

} // memory automatically freed when ptr goes out of scope

**6. Advantages of Boost**

* Reduces **manual memory management**.
* Cross-platform **portability**.
* Provides **advanced features** not yet in STL.
* Many libraries are **header-only**, easy to include.

**7. Commonly Used Boost Libraries**

* boost::shared\_ptr / boost::scoped\_ptr
* boost::optional
* boost::filesystem
* boost::thread
* boost::regex
* boost::algorithm

✅ **Key Takeaways**

* Boost is a **high-quality extension of C++**.
* Helps with **memory management, file system, regex, threads, and more**.
* Many Boost libraries influenced **modern C++ standard library**.
* Can be **header-only or compiled**, depending on library type.

**1. What is Boost?**

* **Boost** is a collection of **peer-reviewed, portable C++ libraries**.
* Provides **well-tested, reusable, and efficient components** that complement the C++ Standard Library.
* Many Boost libraries have influenced **modern C++ standards** (e.g., std::shared\_ptr, std::regex).

**2. Philosophy of Boost**

Boost follows a set of guiding principles that make it reliable and widely adopted:

**A. Quality and Peer Review**

* Each library undergoes **peer review by C++ experts**.
* Ensures **high-quality code**, performance, and consistency.

**B. Portability**

* Designed to **work across multiple platforms** (Windows, Linux, macOS).
* Abstracts OS-specific details, so developers can focus on logic, not platform quirks.

**C. Reusability**

* Libraries are **generic and flexible**.
* Can be reused in **different projects** without modification.

**D. Complement Standard Library**

* Boost fills **gaps in the standard library**.
* Often serves as a **proving ground** for new C++ features.

**E. Efficiency**

* Designed to be **high-performance**, leveraging templates and modern C++ idioms.

**F. Modern C++ Practices**

* Uses **templates, RAII, smart pointers, and generic programming** techniques.
* Encourages **safe and maintainable C++ code**.

**3. Modularity of Boost**

Boost is **modular**, meaning you can use **only the libraries you need**.

**A. Header-Only Libraries**

* No separate compilation required.
* Just include the headers, e.g., #include <boost/optional.hpp>.

**B. Compiled Libraries**

* Some libraries require **building and linking** (like boost.thread or boost.system).
* Boost provides b2 or bjam to **build these libraries**.

**C. Library Categories**

Boost libraries are **organized into categories** for easier usage:

| **Category** | **Examples** |
| --- | --- |
| Smart Pointers | shared\_ptr, scoped\_ptr |
| Containers & Arrays | array, multi\_array, unordered\_map |
| Algorithms & Utilities | algorithm, lexical\_cast, optional |
| Date & Time | date\_time |
| Filesystem | filesystem |
| Threading & Concurrency | thread, mutex, condition\_variable |
| Serialization | serialization |
| Graphs | graph |
| Regex | regex |

**4. Key Advantages of Modularity**

* **Lightweight**: Only include what you need.
* **Maintainable**: Smaller code footprint.
* **Flexible**: Works with multiple C++ standards (C++98 to C++20+).
* **Testable**: Each library can be tested independently.

**5. Real-Life Analogy**

Think of Boost as a **toolbox**:

* You don’t have to carry all tools; pick only what you need.
* Each tool is **peer-reviewed, high-quality, and cross-platform**.
* Some tools are ready to use immediately (header-only), some require assembly (compiled libraries).

✅ **Key Takeaways**

* Boost is a **high-quality, peer-reviewed C++ library collection**.
* **Philosophy:** portability, reusability, efficiency, modern C++ practices.
* **Modularity:** use only what you need, header-only vs compiled libraries.
* Many Boost libraries later **became part of the C++ standard**.

**1. What Are Smart Pointers?**

* Smart pointers are **objects that act like pointers** but **manage memory automatically**.
* They **automatically delete the memory** they point to when it’s no longer needed.
* Prevent **memory leaks** and **dangling pointers**.

Boost provides several types of smart pointers:

**2. Types of Boost Smart Pointers**

| **Smart Pointer** | **Description** |
| --- | --- |
| boost::shared\_ptr | Reference-counted pointer; memory freed when **last owner goes out of scope**. |
| boost::scoped\_ptr | Simple RAII pointer; **cannot be copied**, automatically deleted at scope exit. |
| boost::weak\_ptr | Non-owning pointer; used with shared\_ptr to break **circular references**. |
| boost::intrusive\_ptr | Pointer that uses **internal reference counting** inside the object itself. |

**3. boost::shared\_ptr**

* Most commonly used smart pointer.
* Maintains a **reference count** internally.
* Memory is deleted automatically when the count reaches zero.

#include <iostream>

#include <boost/shared\_ptr.hpp>

using namespace std;

using namespace boost;

class Test {

public:

void show() { cout << "Hello from Test\n"; }

};

int main() {

shared\_ptr<Test> ptr1(new Test()); // allocate object

shared\_ptr<Test> ptr2 = ptr1; // reference count increases

cout << "Use count: " << ptr1.use\_count() << endl; // 2

ptr1->show();

ptr2->show();

} // object deleted automatically when both ptr1 and ptr2 go out of scope

**Output:**

Use count: 2

Hello from Test

Hello from Test

**4. boost::scoped\_ptr**

* Simple smart pointer with **scope-based lifetime**.
* **Cannot be copied or assigned**, safer than raw pointers in local scope.
* Automatically deleted at scope exit.

#include <boost/scoped\_ptr.hpp>

#include <iostream>

using namespace std;

using namespace boost;

class Demo {

public:

Demo() { cout << "Constructor called\n"; }

~Demo() { cout << "Destructor called\n"; }

};

int main() {

scoped\_ptr<Demo> p(new Demo()); // memory auto-freed at end of scope

}

**Output:**

Constructor called

Destructor called

**5. boost::weak\_ptr**

* **Non-owning** smart pointer.
* Points to an object managed by shared\_ptr without increasing reference count.
* Used to **break circular references** in complex object graphs.

shared\_ptr<int> sp(new int(42));

weak\_ptr<int> wp = sp;

cout << wp.expired() << endl; // false

sp.reset(); // release shared\_ptr

cout << wp.expired() << endl; // true

**6. Advantages of Boost Smart Pointers**

* Automatic memory management → **no explicit delete**.
* Prevents **memory leaks** and **dangling pointers**.
* Safer than raw pointers in **complex programs**.
* Works well with **dynamic objects and containers**.

**7. Key Notes**

* Use **shared\_ptr** for shared ownership.
* Use **scoped\_ptr** for single ownership within a scope.
* Use **weak\_ptr** to avoid circular references.
* Modern C++ (C++11+) has **std::shared\_ptr and std::unique\_ptr**, which are similar to Boost smart pointers.

**1. What is Boost.Filesystem?**

* Provides **portable and high-level access** to the filesystem (files, directories, paths).
* Works on **Windows, Linux, macOS**.
* Replaces the need for platform-specific APIs like CreateFile or stat.
* Many features of Boost.Filesystem are now part of **C++17 <filesystem>**.

**2. Key Features**

* Work with **paths** (boost::filesystem::path).
* Check **existence** of files or directories.
* Create, remove, and iterate **directories and files**.
* Query **file size, permissions, and type**.
* Combine **cross-platform path operations** (like concatenating directories).

**3. Setup**

1. Include header:

#include <boost/filesystem.hpp>

1. Use namespace:

namespace fs = boost::filesystem;

1. Link **Boost.Filesystem** and **Boost.System** libraries in your project (compiled libraries required).

**4. Basic Examples**

**A. Working with Paths**

#include <iostream>

#include <boost/filesystem.hpp>

namespace fs = boost::filesystem;

using namespace std;

int main() {

fs::path p1("C:/Users/Example/Documents");

fs::path p2 = p1 / "file.txt"; // append path safely

cout << "Full path: " << p2 << endl;

cout << "Filename: " << p2.filename() << endl;

cout << "Parent path: " << p2.parent\_path() << endl;

}

**Output (example):**

Full path: C:/Users/Example/Documents/file.txt

Filename: file.txt

Parent path: C:/Users/Example/Documents

**B. Check Existence and Type**

fs::path p("example.txt");

if (fs::exists(p)) {

cout << "File exists\n";

if (fs::is\_regular\_file(p)) cout << "It is a regular file\n";

if (fs::is\_directory(p)) cout << "It is a directory\n";

} else {

cout << "File does not exist\n";

}

**C. Create and Remove Directories**

fs::path dir("MyFolder");

if (!fs::exists(dir)) {

fs::create\_directory(dir);

cout << "Directory created\n";

}

fs::remove(dir); // remove directory

cout << "Directory removed\n";

**D. Iterate Over Directory Contents**

fs::path dir("C:/Users/Example/Documents");

if (fs::exists(dir) && fs::is\_directory(dir)) {

for (auto& entry : fs::directory\_iterator(dir)) {

cout << entry.path() << endl;

}

}

**E. File Size**

fs::path file("example.txt");

if (fs::exists(file) && fs::is\_regular\_file(file)) {

cout << "File size: " << fs::file\_size(file) << " bytes\n";

}

**5. Advantages of Boost.Filesystem**

* **Cross-platform** → same code works on Windows/Linux/macOS.
* **Safe path handling** → avoids manual string concatenation.
* **High-level API** → easier than raw OS calls.
* Works well with **Boost.Threads, Boost.SmartPointers, and other Boost libraries**.

**6. Notes**

* Many Boost.Filesystem functions **throw exceptions**, so wrap calls in try/catch for robust code.

try {

fs::remove("nonexistent.txt");

} catch (fs::filesystem\_error& e) {

cout << e.what() << endl;

}

* Boost.Filesystem uses **UTF-8 and wide strings** for cross-platform path support.

✅ **Key Takeaways**

* Boost.Filesystem is a **cross-platform C++ library** for file and directory operations.
* Handles **paths, files, directories, permissions, sizes, and iteration**.
* Safer and more portable than platform-specific APIs.
* C++17 <filesystem> is inspired by Boost.Filesystem, but Boost still offers **broader backward compatibility**.

**1. What is Boost.Asio?**

* **Boost.Asio** is a **cross-platform C++ library** for:
  + Network programming (TCP/UDP sockets)
  + Low-level I/O
  + Timers
  + Serial ports
* Supports **synchronous** and **asynchronous** operations.
* Works on **Windows, Linux, macOS**.

**2. Key Features**

* **Asynchronous I/O** → efficient, non-blocking operations.
* **Cross-platform networking** → same code works across OSs.
* Supports **TCP, UDP, ICMP, serial ports**.
* Integrates with **Boost.SmartPointers** for safe memory management.
* Can work **standalone or with Boost libraries** like thread.

**3. Basic Concepts**

* **io\_context**: core I/O service that manages all asynchronous operations.
* **Sockets**: objects for network communication (tcp::socket, udp::socket).
* **Endpoints**: represent network addresses (tcp::endpoint).
* **Handlers**: callback functions called when an asynchronous operation completes.

**4. Example: Simple TCP Client (Synchronous)**

#include <boost/asio.hpp>

#include <iostream>

using namespace boost::asio;

using ip::tcp;

using namespace std;

int main() {

io\_context io;

// Connect to localhost:1234

tcp::socket socket(io);

tcp::endpoint endpoint(ip::make\_address("127.0.0.1"), 1234);

socket.connect(endpoint);

string msg = "Hello, server!";

boost::system::error\_code error;

write(socket, buffer(msg), error);

if (!error) {

cout << "Message sent\n";

} else {

cout << "Error: " << error.message() << endl;

}

return 0;

}

**5. Key Advantages of Boost.Asio**

* **Cross-platform**: Works the same on Windows/Linux/macOS.
* **Flexible**: Supports **both synchronous and asynchronous I/O**.
* **Integration**: Works well with **threads, smart pointers, and other Boost libraries**.
* **High performance**: Ideal for **network servers, clients, and I/O-bound applications**.

**6. Notes**

* Boost.Asio can be used **without Boost** in modern C++20+ via **standalone Asio**.
* It’s the foundation for **high-performance network servers**, including HTTP, TCP, and UDP applications.

✅ **Key Takeaways**

* Boost.Asio = **networking and asynchronous I/O library**.
* Core concepts: **io\_context, sockets, endpoints, handlers**.
* Supports **synchronous and asynchronous operations**, **cross-platform**, integrates with other Boost libraries.

**1. What is Boost.Regex?**

* Provides **regular expression support** for C++ programs.
* Allows **pattern matching, searching, replacing, and splitting** strings.
* More flexible and robust than older C-style regex functions (regex.h).
* Many of its features inspired **std::regex in C++11**, but Boost.Regex works in **older C++ standards** as well.

**2. Key Features**

* **Pattern matching** using standard regex syntax.
* **Searching strings** for matches.
* **Replacing text** using regex patterns.
* **Splitting strings** based on patterns.
* **Unicode support** for wide strings.
* Exception-safe operations.

**3. Basic Syntax**

#include <boost/regex.hpp>

#include <iostream>

#include <string>

using namespace std;

using namespace boost;

int main() {

string text = "My email is example@test.com";

regex expr("\\w+@\\w+\\.\\w+"); // regex pattern for email

if (regex\_search(text, expr)) {

cout << "Email found!" << endl;

}

}

**Output:**

Email found!

**4. Common Boost.Regex Functions**

| **Function** | **Description** |
| --- | --- |
| regex\_match(str, reg) | Returns true if **entire string** matches regex |
| regex\_search(str, reg) | Returns true if **any part of string** matches regex |
| regex\_replace(str, reg, repl) | Replaces all matches of regex with replacement string |
| sregex\_iterator | Iterate over all matches in a string |

**5. Example: Regex Replace**

string text = "Boost is fun!";

regex expr("fun");

string result = regex\_replace(text, expr, "awesome");

cout << result << endl;

**Output:**

Boost is awesome!

**6. Example: Iterating All Matches**

string text = "Numbers: 10, 20, 30";

regex expr("\\d+"); // matches digits

sregex\_iterator begin(text.begin(), text.end(), expr);

sregex\_iterator end;

for (auto it = begin; it != end; ++it) {

cout << it->str() << endl; // prints 10 20 30

}

**7. Advantages of Boost.Regex**

* Works in **older and modern C++ standards**.
* Supports **complex patterns, Unicode, and wide strings**.
* Exception-safe and reliable.
* More powerful than simple string parsing.

✅ **Key Takeaways**

* Boost.Regex = **powerful regex library for C++**.
* Supports **matching, searching, replacing, splitting, and iteration**.
* Compatible with **older C++ versions** and influenced std::regex.
* Works with both **narrow and wide strings**.

**1. Boost.Bind**

**Purpose**

* boost::bind allows you to **bind arguments to a function** and create a **callable object** (functor).
* Useful for **callbacks**, **delayed execution**, or passing functions to algorithms.

**Syntax**

boost::bind(function, arg1, arg2, ...);

* function → function pointer or member function
* argN → arguments to bind; \_1, \_2, … are placeholders for unbound arguments

**Example 1: Simple Function Binding**

#include <boost/bind.hpp>

#include <iostream>

using namespace std;

using namespace boost;

void print\_sum(int a, int b) {

cout << "Sum: " << a + b << endl;

}

int main() {

auto f = bind(print\_sum, 10, 20); // bind arguments

f(); // calls print\_sum(10, 20)

}

**Output:**

Sum: 30

**Example 2: Using Placeholders**

void multiply(int a, int b) {

cout << "Product: " << a \* b << endl;

}

int main() {

auto f = bind(multiply, \_1, 5); // bind 2nd argument as 5, 1st is placeholder

f(10); // calls multiply(10, 5)

}

**Output:**

Product: 50

**Example 3: Member Function Binding**

class Test {

public:

void greet(string name) { cout << "Hello, " << name << endl; }

};

int main() {

Test t;

auto f = bind(&Test::greet, &t, \_1);

f("Alice"); // calls t.greet("Alice")

}

**2. Boost.Lambda**

**Purpose**

* Provides **inline anonymous functions (lambdas)** in C++03, before C++11 lambdas were available.
* Often used with **STL algorithms** like for\_each, transform, sort, etc.

**Example 1: Simple Lambda**

#include <boost/lambda/lambda.hpp>

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

using namespace boost::lambda;

int main() {

vector<int> v = {1, 2, 3, 4, 5};

for\_each(v.begin(), v.end(), cout << \_1 << " ");

}

**Output:**

1 2 3 4 5

**Example 2: Modify Elements**

vector<int> v = {1, 2, 3, 4, 5};

for\_each(v.begin(), v.end(), \_1 = \_1 \* 2); // double each element

for\_each(v.begin(), v.end(), cout << \_1 << " ");

**Output:**

2 4 6 8 10

**Example 3: Combining Bind and Lambda**

#include <boost/bind.hpp>

#include <boost/lambda/lambda.hpp>

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

using namespace boost;

using namespace boost::lambda;

void print\_num(int n) { cout << n << " "; }

int main() {

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

for\_each(v.begin(), v.end(), bind(print\_num, \_1)); // using bind

cout << endl;

for\_each(v.begin(), v.end(), cout << \_1 << " "); // using lambda

}

**Output:**

1 2 3

1 2 3

**3. Key Points**

* boost::bind = **bind function arguments**, create functors for callbacks.
* boost::lambda = **inline anonymous functions** (pre-C++11).
* Both are **useful with STL algorithms** like for\_each, transform, sort, etc.
* Modern C++ (C++11+) has **std::bind** and **lambda expressions**, but Boost is great for **C++03 compatibility**.

**1. Templates in C++**

**Definition**

* **Templates** allow writing **generic classes or functions** that work with **any data type**.
* Eliminates the need to **rewrite the same code** for different data types.

**A. Function Templates**

#include <iostream>

using namespace std;

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << add(5, 10) << endl; // int

cout << add(3.5, 2.5) << endl; // double

}

**Output:**

15

6

**B. Class Templates**

#include <iostream>

using namespace std;

template <class T>

class Box {

T content;

public:

void setContent(T c) { content = c; }

T getContent() { return content; }

};

int main() {

Box<int> intBox;

intBox.setContent(123);

cout << intBox.getContent() << endl;

Box<string> strBox;

strBox.setContent("Hello");

cout << strBox.getContent() << endl;

}

**Output:**

123

Hello

**Advantages of Templates**

* Code **reusability**.
* Type-safe: works with multiple data types.
* Reduces code duplication.

**2. Standard Template Library (STL)**

**Definition**

* STL is a **library of generic classes and functions** in C++.
* Built using **templates** for maximum flexibility.
* Provides **common data structures, algorithms, and iterators**.

**3. Components of STL**

| **Component** | **Description** |
| --- | --- |
| **Containers** | Store data (e.g., vector, list, map, set, stack, queue) |
| **Algorithms** | Functions for searching, sorting, manipulating data (e.g., sort, find, reverse) |
| **Iterators** | Objects to **traverse containers** (like pointers) |
| **Functors** | Objects acting like **functions**, used in algorithms |

**A. Containers Example**

**Vector**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> v = {1, 2, 3, 4, 5};

v.push\_back(6); // add element

v.pop\_back(); // remove last element

for (int i : v) cout << i << " "; // iterate

}

**Output:**

1 2 3 4 5

**Map**

#include <iostream>

#include <map>

using namespace std;

int main() {

map<int, string> m;

m[1] = "Alice";

m[2] = "Bob";

for (auto &pair : m)

cout << pair.first << " -> " << pair.second << endl;

}

**Output:**

1 -> Alice

2 -> Bob

**B. Algorithms Example**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

vector<int> v = {5, 2, 9, 1, 5};

sort(v.begin(), v.end()); // sort ascending

reverse(v.begin(), v.end()); // reverse

for (int i : v) cout << i << " ";

}

**Output:**

9 5 5 2 1

**C. Iterators Example**

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

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)

cout << \*it << " ";

* Iterators allow **traversing containers** uniformly.
* STL algorithms work with **any container that provides iterators**.

**4. Key Advantages of STL**

* **Reusable, generic code** → works with any data type.
* **High performance** → implemented efficiently.
* **Reduces coding effort** → common algorithms already implemented.
* **Consistency** → same interface for all containers.

✅ **Key Takeaways**

* Templates = **generic programming** in C++.
* STL = **library of generic containers, algorithms, iterators, and functors**.
* Containers store data, algorithms manipulate data, iterators traverse data.
* Works together to write **efficient, reusable, and type-safe code**.

**1. Function Templates**

**Definition**

* A **function template** is a blueprint for creating **functions** that can work with **any data type**.
* You write a **single function template**, and the compiler generates type-specific functions as needed.

**Syntax**

template <typename T>

T add(T a, T b) {

return a + b;

}

**Example**

#include <iostream>

using namespace std;

template <typename T>

T multiply(T a, T b) {

return a \* b;

}

int main() {

cout << multiply(5, 3) << endl; // int

cout << multiply(2.5, 1.5) << endl; // double

}

**Output:**

15

3.75

**Key Points**

* Works for **single operation or algorithm**.
* Can be **overloaded** or specialized.
* Generated by compiler **per type used**.

**2. Class Templates**

**Definition**

* A **class template** is a blueprint for creating **classes** that can work with **any data type**.
* Allows creation of **generic data structures** like stacks, boxes, arrays, etc.

**Syntax**

template <typename T>

class Box {

T content;

public:

void setContent(T c) { content = c; }

T getContent() { return content; }

};

**Example**

#include <iostream>

using namespace std;

template <class T>

class Pair {

T first, second;

public:

Pair(T a, T b) : first(a), second(b) {}

void show() { cout << first << ", " << second << endl; }

};

int main() {

Pair<int> p1(10, 20);

p1.show();

Pair<string> p2("Hello", "World");

p2.show();

}

**Output:**

10, 20

Hello, World

**Key Points**

* Works for **data structures or classes** with member functions.
* Can have **multiple member functions and variables**.
* Provides **type-safe containers and objects**.

**3. Differences: Function Templates vs Class Templates**

| **Feature** | **Function Templates** | **Class Templates** |
| --- | --- | --- |
| Purpose | Create **generic functions** | Create **generic classes** |
| Syntax | template <typename T> T func(...) | template <typename T> class ClassName |
| Use Case | Single operations (e.g., add, swap) | Containers or objects (e.g., stack, pair) |
| Instantiation | Compiler generates **function per type** | Compiler generates **class per type** |
| Members | Single function | Multiple member variables and functions |
| Example | add<T>(a,b) | Box<T> with setContent() and getContent() |

**4. Summary**

* **Function templates** → for **generic operations**.
* **Class templates** → for **generic data structures or objects**.
* Both are part of **generic programming** and **work together with STL**.

**1. What is a Variable Template?**

* A **variable template** allows you to create a **single variable definition that works with multiple types**.
* Similar to **function templates**, but for variables.
* Useful for **constants or global variables** that depend on a type.

**2. Syntax**

template <typename T>

T variable\_name = initial\_value;

* T → type parameter
* variable\_name → template variable name
* initial\_value → optional initialization

**3. Example 1: Simple Variable Template**

#include <iostream>

using namespace std;

template <typename T>

T pi = T(3.1415926535897932385);

int main() {

cout << pi<float> << endl; // float

cout << pi<double> << endl; // double

cout << pi<long double> << endl; // long double

}

**Output:**

3.14159

3.141592653589793

3.1415926535897932385

**4. Example 2: Template with Constexpr**

* Can define **type-safe constants** using constexpr:

template <typename T>

constexpr T e = T(2.718281828459045);

int main() {

cout << e<float> << endl;

cout << e<double> << endl;

}

**5. Advantages of Variable Templates**

* **Reduces code duplication** for constants or variables for multiple types.
* **Type-safe**: compiler ensures correct type usage.
* Works well with **function templates and class templates**.

**6. Notes**

* Introduced in **C++14**, not available in C++11 or earlier.
* Often used for **constants**, e.g., pi, e, epsilon, or **type-specific default values**.
* Can also be **specialized for specific types** if needed:

template <>

double pi<double> = 3.14159265358979323846; // specialization

✅ **Key Takeaways**

* **Variable templates** = templated variables, just like function or class templates.
* Useful for **type-generic constants** or **global variables**.
* Simplifies **generic programming** and reduces code duplication.
* Fully supported in **C++14 and later**.

**1. What is Template Specialization?**

* Template specialization allows you to **customize a template** for a **specific data type**.
* Useful when a **generic template** does not work efficiently or correctly for a certain type.

**2. Full (Explicit) Specialization**

**Syntax**

template <typename T>

class MyClass {

public:

void show() { std::cout << "Generic template\n"; }

};

// Specialization for int

template <>

class MyClass<int> {

public:

void show() { std::cout << "Specialized template for int\n"; }

};

**Example**

#include <iostream>

using namespace std;

template <typename T>

class Box {

public:

void info() { cout << "Generic Box\n"; }

};

// Specialization for double

template <>

class Box<double> {

public:

void info() { cout << "Double Box\n"; }

};

int main() {

Box<int> b1;

Box<double> b2;

b1.info(); // Generic Box

b2.info(); // Double Box

}

**Output:**

Generic Box

Double Box

**Key Point:** Full specialization **replaces the template entirely** for the specified type.

**3. Partial Specialization**

* Partial specialization allows customizing a template for **a subset of types or patterns**, not just a single type.
* Used mainly with **class templates**, not function templates.

**Syntax**

template <typename T1, typename T2>

class MyClass {

public:

void show() { std::cout << "Generic template\n"; }

};

// Partial specialization when both types are same

template <typename T>

class MyClass<T, T> {

public:

void show() { std::cout << "Partial specialization: both types same\n"; }

};

**Example**

#include <iostream>

using namespace std;

template <typename T1, typename T2>

class Pair {

public:

void display() { cout << "Generic Pair\n"; }

};

// Partial specialization when both types are same

template <typename T>

class Pair<T, T> {

public:

void display() { cout << "Partial Pair: types same\n"; }

};

int main() {

Pair<int, double> p1;

Pair<int, int> p2;

p1.display(); // Generic Pair

p2.display(); // Partial Pair: types same

}

**Output:**

Generic Pair

Partial Pair: types same

**Key Point:** Partial specialization **customizes behavior for a subset of template parameters**.

**4. Differences: Full vs Partial Specialization**

| **Feature** | **Full Specialization** | **Partial Specialization** |
| --- | --- | --- |
| Applies to | A **specific type only** | A **pattern or subset of types** |
| Syntax | template<> class MyClass<int> {} | template<typename T> class MyClass<T,T> {} |
| Replaces | Entire template definition | Partially modifies behavior |
| Example | Box<double> specialized | Pair<T,T> partially specialized |
| Function Templates | Allowed | Not allowed (C++ doesn’t support partial specialization for functions) |

**5. Notes**

* Partial specialization is **class-only** in C++.
* Function templates can use **overloading** instead of partial specialization.
* Specialization helps in **optimizing or customizing behavior** for specific types.

✅ **Key Takeaways**

* **Full specialization** → specific type, completely overrides template.
* **Partial specialization** → subset of types or patterns, modifies template behavior partially.
* Both are used in **generic programming** to handle **type-specific behavior** while keeping generic code reusable.

**1. What are Variadic Templates?**

* Variadic templates allow a **template to accept zero or more template parameters**.
* Useful for writing **generic functions or classes** that can handle **any number of arguments**.

**Example Use Cases:**

* Printing multiple values of different types.
* Implementing generic containers or wrappers.
* Forwarding arguments to other functions (like std::make\_tuple).

**2. Syntax**

template <typename... Args>

void func(Args... args) {

// args is a parameter pack

}

* typename... Args → **template parameter pack**
* args... → **function parameter pack**

**3. Example 1: Simple Variadic Function**

#include <iostream>

using namespace std;

template <typename T>

void print(T t) {

cout << t << endl; // base case

}

template <typename T, typename... Args>

void print(T t, Args... args) {

cout << t << " ";

print(args...); // recursive call

}

int main() {

print(1, 2.5, "Hello", 'A');

}

**Output:**

1 2.5 Hello A

**Explanation:**

* Variadic function expands the parameter pack recursively.
* The **base case** handles the last argument.

**4. Example 2: Variadic Class Template**

#include <iostream>

using namespace std;

template <typename... Types>

class Tuple {

public:

void info() {

cout << sizeof...(Types) << " elements in tuple" << endl;

}

};

int main() {

Tuple<int, double, char> t;

t.info();

}

**Output:**

3 elements in tuple

* sizeof...(Types) gives the **number of template arguments**.

**5. Advantages of Variadic Templates**

* **Flexible** → can accept **any number of arguments** of **any type**.
* **Type-safe** → compiler checks types.
* **Efficient** → expands at **compile time**, no runtime overhead.
* Useful for **generic programming**, logging, forwarding, tuple implementation, and more.

**6. Notes**

* Recursive expansion is common for **processing parameter packs**.
* C++17 introduced **fold expressions** to simplify variadic templates:

**Fold Expression Example**

template <typename... Args>

void sum(Args... args) {

cout << (args + ...) << endl; // sums all arguments

}

int main() {

sum(1, 2, 3, 4); // 10

}

* (args + ...) automatically expands the pack without recursion.

✅ **Key Takeaways**

* Variadic templates = templates with **variable number of parameters**.
* Can be used in **functions or classes**.
* **Recursive expansion** or **fold expressions** are used to process arguments.
* Introduced in **C++11**, enhanced with **fold expressions in C++17**.

**1. What is SFINAE?**

**SFINAE** stands for:  
**Substitution Failure Is Not An Error**

**Definition**

* In C++ templates, if **substituting a template parameter causes an invalid type or expression**, the compiler **does not produce an error**.
* Instead, that **template is simply ignored** in overload resolution.
* Enables **conditional compilation of templates** based on types or properties.

**Example 1: SFINAE Concept**

#include <iostream>

using namespace std;

template <typename T>

auto func(T t) -> decltype(t + 1) { // only valid if T supports +

return t + 1;

}

int main() {

cout << func(5) << endl; // OK, int + 1

// cout << func("abc") << endl; // Error if SFINAE not used properly

}

* Here, decltype(t+1) will only be valid if T supports +.
* If T doesn’t, this version is **discarded**, and other overloads may be chosen.

**2. std::enable\_if**

**Definition**

* std::enable\_if is a **utility in <type\_traits>** that works with SFINAE.
* Enables or disables **function/class templates** based on a **compile-time condition**.

**Syntax**

template <bool B, class T = void>

struct enable\_if { };

template <class T>

struct enable\_if<true, T> { typedef T type; };

* If condition B is **true**, enable\_if defines a type.
* If **false**, substitution fails → function/class **ignored** in overload resolution.

**Example 1: Function Enabled Only for Integral Types**

#include <iostream>

#include <type\_traits>

using namespace std;

template <typename T>

typename enable\_if<is\_integral<T>::value, T>::type

increment(T n) {

return n + 1;

}

int main() {

cout << increment(5) << endl; // OK: int

// cout << increment(3.5) << endl; // Error: double not integral

}

**Explanation:**

* is\_integral<T>::value → true for int, long, char
* If false, SFINAE discards the template
* Ensures **type-safe template selection**.

**Example 2: Using enable\_if in Return Type (C++11 style)**

template <typename T>

typename enable\_if<std::is\_floating\_point<T>::value, T>::type

half(T x) {

return x / 2;

}

int main() {

cout << half(3.5) << endl; // OK

// cout << half(10); // Error: int is not floating point

}

**Example 3: Using enable\_if as Parameter**

template <typename T>

void show(T t, typename enable\_if<std::is\_integral<T>::value>::type\* = 0) {

cout << "Integral: " << t << endl;

}

int main() {

show(10); // OK

// show(3.5); // Error: double not integral

}

**3. Advantages**

* Allows **compile-time selection of templates**.
* Prevents **invalid template instantiations**.
* Enables **type-safe and flexible generic programming**.
* Widely used in **STL, Boost, and modern C++ libraries**.

**4. Notes**

* SFINAE is **core to template metaprogramming**.
* enable\_if is a **common tool to implement SFINAE**.
* Modern C++17 introduces **if constexpr** and C++20 introduces **concepts**, which often simplify SFINAE.

✅ **Key Takeaways**

* **SFINAE** → substitution failure does not produce an error; template discarded.
* **enable\_if** → used with SFINAE to **enable/disable templates** based on type traits.
* Helps in **type-safe, conditional compilation** of functions and classes.

**1. Type Traits**

**Definition**

* **Type traits** are utilities in **<type\_traits>** that allow **compile-time type information**.
* They help **check or modify types** during template programming.
* Heavily used with **SFINAE** and **enable\_if**.

**Common Type Traits**

| **Trait** | **Description** |
| --- | --- |
| is\_integral<T> | Checks if T is an integral type (int, char, etc.) |
| is\_floating\_point<T> | Checks if T is float, double, long double |
| is\_pointer<T> | Checks if T is a pointer |
| remove\_const<T> | Returns type T without const qualifier |
| add\_pointer<T> | Converts type T to a pointer type |

**Example**

#include <iostream>

#include <type\_traits>

using namespace std;

int main() {

cout << boolalpha;

cout << is\_integral<int>::value << endl; // true

cout << is\_integral<double>::value << endl; // false

cout << is\_pointer<int\*>::value << endl; // true

}

**2. decltype**

**Definition**

* decltype inspects **the type of an expression** at compile time.
* Useful for **type deduction**, especially with templates and auto.

**Syntax**

decltype(expression) variable\_name;

**Example 1: Basic Usage**

int x = 5;

decltype(x) y = 10; // y has type int

cout << y << endl;

**Example 2: With Expressions**

int a = 3, b = 4;

decltype(a + b) c = a + b; // type of a+b, i.e., int

cout << c << endl;

**Example 3: With Functions**

int add(int x, int y) { return x + y; }

decltype(add(1,2)) sum; // sum is int

**3. auto**

**Definition**

* auto tells the compiler to **deduce the variable type automatically** from the initializer.
* Introduced in C++11.

**Syntax**

auto variable\_name = initializer;

**Example 1: Basic Usage**

auto x = 10; // int

auto y = 3.14; // double

auto z = "Hello"; // const char\*

**Example 2: With Containers**

#include <vector>

#include <iostream>

using namespace std;

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

for (auto it = v.begin(); it != v.end(); ++it) {

cout << \*it << " ";

}

**Example 3: With Functions**

auto add(int a, int b) -> int {

return a + b;

}

**4. Comparison: decltype vs auto**

| **Feature** | **decltype** | **auto** |
| --- | --- | --- |
| Purpose | Deduces **type of expression** | Deduces type **from initializer** |
| Usage | decltype(x+y) z; | auto z = x+y; |
| Compile-time | Yes | Yes |
| Modification | Does not initialize | Initializes with value |

**5. Advantages**

* **Type traits** → **type introspection** for templates.
* **decltype** → get **exact type of an expression**, even complex ones.
* **auto** → reduces boilerplate, easier with **STL iterators** and templates.
* Together, they make **modern C++ generic programming** simpler and safer.

✅ **Key Takeaways**

1. **Type traits** → check, modify, or inspect types at compile-time.
2. **decltype** → deduce the type of an expression without evaluating it.
3. **auto** → automatic type deduction from initializer.
4. Widely used in **templates, STL, SFINAE, and generic programming**.

**1. vector**

**Definition**

* Dynamic **array** that can **grow or shrink** at runtime.
* Provides **random access**, supports iterators.

**Key Functions**

* push\_back(), pop\_back(), size(), begin(), end(), insert(), erase()

**Example**

#include <iostream>

#include <vector>

using namespace std;

int main() {

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

v.push\_back(4);

v.pop\_back();

for (int i : v) cout << i << " "; // 1 2 3

}

**Properties:**

* Dynamic size
* Fast random access
* Slower insert/delete in middle

**2. list**

**Definition**

* **Doubly linked list**.
* Efficient **insert/delete at any position**, no random access.

**Key Functions**

* push\_back(), push\_front(), pop\_back(), pop\_front(), insert(), erase()

**Example**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> l = {1, 2, 3};

l.push\_back(4);

l.push\_front(0);

for (int i : l) cout << i << " "; // 0 1 2 3 4

}

**Properties:**

* No random access (use iterators)
* Efficient insert/delete anywhere

**3. map**

**Definition**

* **Associative container** storing **key-value pairs**.
* **Ordered** by key (internally uses Red-Black tree).

**Key Functions**

* insert(), erase(), find(), operator[]

**Example**

#include <iostream>

#include <map>

using namespace std;

int main() {

map<int, string> m;

m[1] = "Alice";

m[2] = "Bob";

for (auto &p : m) cout << p.first << " -> " << p.second << endl;

}

**Output:**

1 -> Alice

2 -> Bob

**Properties:**

* Sorted keys
* O(log n) insert/find/erase

**4. unordered\_map**

**Definition**

* **Associative container** storing **key-value pairs**.
* **Unordered**, implemented using **hash tables**.

**Key Functions**

* Same as map: insert(), erase(), find(), operator[]

**Example**

#include <iostream>

#include <unordered\_map>

using namespace std;

int main() {

unordered\_map<string, int> um;

um["apple"] = 5;

um["banana"] = 3;

for (auto &p : um) cout << p.first << " -> " << p.second << endl;

}

**Properties:**

* Not sorted
* Average O(1) insert/find/erase
* Faster than map for large data

**5. set**

**Definition**

* **Associative container** storing **unique keys** in **sorted order**.

**Key Functions**

* insert(), erase(), find(), count(), begin(), end()

**Example**

#include <iostream>

#include <set>

using namespace std;

int main() {

set<int> s;

s.insert(3);

s.insert(1);

s.insert(2);

s.insert(2); // duplicate ignored

for (int x : s) cout << x << " "; // 1 2 3

}

**Properties:**

* Unique keys only
* Sorted order
* O(log n) operations

**6. Quick Comparison Table**

| **Container** | **Type** | **Order** | **Duplicate** | **Access** | **Complexity (insert/find)** |
| --- | --- | --- | --- | --- | --- |
| vector | Sequence | No | Yes | Random | O(n) middle insert |
| list | Sequence | No | Yes | No random | O(1) insert/delete |
| map | Associative | Yes (key) | No | By key | O(log n) |
| unordered\_map | Associative | No | No | By key | O(1) average |
| set | Associative | Yes | No | By key | O(log n) |

✅ **Key Takeaways**

* **vector** → dynamic array, random access
* **list** → linked list, efficient insert/delete
* **map** → ordered key-value store
* **unordered\_map** → hash-based key-value store
* **set** → ordered unique elements

**1. Iterators**

**Definition**

* Iterators are **objects that point to elements in containers** and allow **traversing, accessing, or modifying** elements.
* Think of them as **generalized pointers** for STL containers.

**Iterator Categories**

| **Type** | **Features** | **Example Containers** |
| --- | --- | --- |
| **Input Iterator** | Read elements sequentially | istream\_iterator |
| **Output Iterator** | Write elements sequentially | ostream\_iterator |
| **Forward Iterator** | Read/write, can move forward | forward\_list |
| **Bidirectional** | Move forward & backward | list, set, map |
| **Random Access** | Supports +, -, indexing | vector, deque, array |

**Iterator Syntax**

container<T>::iterator it = container.begin();

* begin() → points to **first element**
* end() → points **one past the last element**

**Example: Using Iterators**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> v = {1,2,3,4,5};

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {

cout << \*it << " ";

}

}

**Output:**

1 2 3 4 5

**Modern C++: Auto Iterator**

for (auto it = v.begin(); it != v.end(); ++it) {

cout << \*it << " ";

}

**2. Algorithms**

**Definition**

* Algorithms are **template-based functions** that operate on **containers via iterators**.
* Includes **searching, sorting, transforming, counting, and modifying elements**.

**Common STL Algorithms**

| **Algorithm** | **Purpose** |
| --- | --- |
| sort | Sort elements |
| reverse | Reverse container |
| for\_each | Apply a function to all elements |
| find | Find an element |
| count | Count occurrences |
| accumulate | Sum of elements (from <numeric>) |
| transform | Apply operation and store results |
| min\_element/max\_element | Find min/max element |

**Example 1: Sorting and Reversing**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

vector<int> v = {5,2,9,1,5};

sort(v.begin(), v.end()); // ascending

reverse(v.begin(), v.end()); // reverse

for (int x : v) cout << x << " ";

}

**Output:**

9 5 5 2 1

**Example 2: Searching and Counting**

vector<int> v = {1,2,3,2,4};

auto it = find(v.begin(), v.end(), 2); // first occurrence of 2

cout << "Found: " << \*it << endl;

int cnt = count(v.begin(), v.end(), 2); // count of 2

cout << "Count: " << cnt << endl;

**Output:**

Found: 2

Count: 2

**Example 3: Transform / Lambda**

vector<int> v = {1,2,3,4,5};

vector<int> v2(v.size());

transform(v.begin(), v.end(), v2.begin(), [](int x){ return x\*2; });

for (int x : v2) cout << x << " "; // 2 4 6 8 10

**Key Points**

* Algorithms work **independently of container type** → they operate via iterators.
* Iterators **connect containers to algorithms**.
* Together, they provide **powerful, reusable, and generic operations** in STL.

✅ **Key Takeaways**

1. **Iterators** → traverse containers (like generalized pointers).
2. **Algorithms** → operate on containers via iterators.
3. **STL design** → separation of **data structures (containers)** and **operations (algorithms)**.
4. Modern C++ often uses **auto, lambdas, and range-based loops** to simplify iterators and algorithms.

**1. Functors (Function Objects)**

**Definition**

* A **functor** is a **class or struct with operator() overloaded**, so instances behave like functions.
* Allows **stateful behavior** unlike regular functions.

**Syntax**

struct Adder {

int operator()(int a, int b) {

return a + b;

}

};

**Example**

#include <iostream>

using namespace std;

struct Multiply {

int factor;

Multiply(int f) : factor(f) {}

int operator()(int x) { return x \* factor; }

};

int main() {

Multiply mul2(2);

cout << mul2(5) << endl; // 10

}

**Advantages**

* Can **store state** (like factor).
* Can be passed to **STL algorithms**.

**2. Lambdas**

**Definition**

* **Anonymous functions** defined **inline** using [capture](parameters) -> return\_type { body }.
* Introduced in **C++11**, widely used in STL algorithms.

**Syntax**

[ captures ] ( parameters ) -> return\_type { body }

**Example 1: Basic Lambda**

#include <iostream>

using namespace std;

int main() {

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

cout << add(3,4) << endl; // 7

}

**Example 2: Lambda with Capture**

int factor = 3;

auto multiply = [factor](int x) { return x \* factor; };

cout << multiply(5) << endl; // 15

**Example 3: Lambda with STL**

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

int main() {

vector<int> v = {1,2,3,4,5};

for\_each(v.begin(), v.end(), [](int x){ cout << x\*2 << " "; });

}

**Output:**

2 4 6 8 10

**3. std::function**

**Definition**

* A **general-purpose polymorphic function wrapper** in <functional>.
* Can **store any callable**: function pointer, lambda, functor, or bind expression.

**Syntax**

std::function<return\_type(parameter\_types)> func;

**Example 1: Store Lambda**

#include <iostream>

#include <functional>

using namespace std;

int main() {

function<int(int,int)> add = [](int a, int b){ return a + b; };

cout << add(5,7) << endl; // 12

}

**Example 2: Store Functor**

struct Multiply {

int operator()(int a, int b) { return a\*b; }

};

int main() {

function<int(int,int)> mul = Multiply();

cout << mul(3,4) << endl; // 12

}

**Advantages**

* **Type-erased wrapper** → can hold **any callable of matching signature**.
* Useful in **callbacks, event handling, and STL algorithms**.

**4. Comparison Table**

| **Feature** | **Functor** | **Lambda** | **std::function** |
| --- | --- | --- | --- |
| Definition | Class with operator() | Anonymous inline function | Type-erased function wrapper |
| Stateful | Yes | Can capture state | Yes (via callable stored) |
| Syntax | Struct/class | [captures](args)->type{} | function<type> |
| STL Usage | Yes | Yes | Yes |
| Flexibility | Limited (fixed type) | Very flexible | Most flexible, type-erased |

**5. Key Points**

* **Functors** → reusable, stateful, works in STL.
* **Lambdas** → concise, inline, modern C++.
* **std::function** → general wrapper, can store **any callable with compatible signature**.
* Together, they **enable functional programming patterns in C++**.

Smart pointers **automate memory management** and help prevent **memory leaks and dangling pointers**.

**1. unique\_ptr**

**Definition**

* unique\_ptr is a **smart pointer that owns a resource exclusively**.
* Only **one unique\_ptr can own the object** at a time.
* Automatically deletes the object when the unique\_ptr goes out of scope.

**Key Points**

* Cannot be **copied** (copy constructor is deleted).
* Can be **moved** to transfer ownership.
* Lightweight and no reference counting.

**Syntax**

#include <memory>

std::unique\_ptr<Type> ptr(new Type(args));

**Example**

#include <iostream>

#include <memory>

using namespace std;

int main() {

unique\_ptr<int> p1 = make\_unique<int>(10); // preferred over new

cout << \*p1 << endl;

// Transfer ownership

unique\_ptr<int> p2 = move(p1);

if (!p1) cout << "p1 is null\n";

cout << \*p2 << endl; // 10

}

**Output:**

10

p1 is null

10

**Advantages:**

* Prevents memory leaks
* Lightweight
* Ownership semantics clear

**2. shared\_ptr**

**Definition**

* shared\_ptr is a **reference-counted smart pointer**.
* Multiple shared\_ptrs can share ownership of the same object.
* Object is deleted automatically when **last shared\_ptr goes out of scope**.

**Key Points**

* Reference count is maintained internally.
* Can be **copied freely**, unlike unique\_ptr.
* Slightly heavier due to reference counting.

**Syntax**

#include <memory>

std::shared\_ptr<Type> ptr = std::make\_shared<Type>(args);

**Example**

#include <iostream>

#include <memory>

using namespace std;

int main() {

shared\_ptr<int> p1 = make\_shared<int>(20);

cout << \*p1 << endl;

cout << "Use count: " << p1.use\_count() << endl;

shared\_ptr<int> p2 = p1; // shared ownership

cout << "Use count after copy: " << p1.use\_count() << endl;

p1.reset(); // release one reference

cout << "Use count after reset: " << p2.use\_count() << endl;

}

**Output:**

20

Use count: 1

Use count after copy: 2

Use count after reset: 1

**Advantages:**

* Safe sharing of resources
* Automatic cleanup when last owner is gone
* Can be used with **STL containers**

**3. Quick Comparison: unique\_ptr vs shared\_ptr**

| **Feature** | **unique\_ptr** | **shared\_ptr** |
| --- | --- | --- |
| Ownership | Single owner | Multiple owners (shared) |
| Copyable | No | Yes |
| Movable | Yes | Yes |
| Reference Count | No | Yes |
| Overhead | Low | Slightly higher (due to ref count) |
| Use Case | Exclusive ownership, lightweight | Shared ownership, flexible |

**4. Notes**

* Prefer **unique\_ptr** when exclusive ownership is sufficient.
* Use **shared\_ptr** only when multiple owners are necessary.
* Avoid **raw new/delete** in modern C++ unless interfacing with legacy code.
* weak\_ptr can be used with shared\_ptr to **break circular references**.

✅ **Key Takeaways**

1. Smart pointers automate **memory management**.
2. unique\_ptr → exclusive ownership, lightweight.
3. shared\_ptr → shared ownership, reference-counted.
4. Modern C++ encourages **make\_unique and make\_shared** over raw new.

The STL is **exception-safe**, meaning it provides mechanisms to handle errors safely without leaking resources.

**1. Basics of Exception Handling in C++**

* Use try, catch, and throw keywords.
* Exceptions are thrown when **runtime errors** occur.

try {

// code that may throw

} catch (const std::exception& e) {

// handle exception

}

**Example:**

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

throw runtime\_error("Something went wrong!");

} catch (const exception& e) {

cout << "Exception: " << e.what() << endl;

}

}

**Output:**

Exception: Something went wrong!

**2. Exceptions in STL Containers**

STL containers **throw exceptions** in certain scenarios:

| **STL Component** | **Exception Scenario** | **Exception Type** |
| --- | --- | --- |
| vector::at() | Access out of bounds | std::out\_of\_range |
| map::at() | Key not found | std::out\_of\_range |
| std::string::at() | Index out of bounds | std::out\_of\_range |
| Memory allocation | new or make\_shared fails | std::bad\_alloc |

**Example 1: vector::at()**

#include <iostream>

#include <vector>

using namespace std;

int main() {

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

try {

cout << v.at(5) << endl; // out of bounds

} catch (const out\_of\_range& e) {

cout << "Caught exception: " << e.what() << endl;

}

}

**Output:**

Caught exception: vector::\_M\_range\_check: \_\_n (which is 5) >= this->size() (which is 3)

**Example 2: map::at()**

#include <iostream>

#include <map>

using namespace std;

int main() {

map<int, string> m;

m[1] = "Alice";

try {

cout << m.at(2) << endl; // key 2 not present

} catch (const out\_of\_range& e) {

cout << "Caught exception: " << e.what() << endl;

}

}

**Output:**

Caught exception: map::at: key not found

**Example 3: Memory Allocation Failure**

#include <iostream>

#include <memory>

using namespace std;

int main() {

try {

auto p = make\_shared<int[]>(1e10); // very large allocation

} catch (const bad\_alloc& e) {

cout << "Memory allocation failed: " << e.what() << endl;

}

}

**3. STL Algorithms and Exception Safety**

* Most STL algorithms provide **strong exception guarantees**:
  + Either **complete success** or **no changes to data**.
* Examples: sort, copy, transform → if an exception occurs, container is not left in an invalid state.
* Functors/lambdas used in algorithms may throw exceptions; STL propagates them to the caller.

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

int main() {

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

try {

for\_each(v.begin(), v.end(), [](int x) {

if (x == 2) throw runtime\_error("Error in functor");

cout << x << " ";

});

} catch (const runtime\_error& e) {

cout << "\nCaught: " << e.what() << endl;

}

}

**Output:**

1

Caught: Error in functor

**4. Key Points**

1. STL containers **throw exceptions** for invalid operations (at, memory allocation).
2. STL algorithms are **exception-safe** — they **don’t corrupt data** if an exception occurs.
3. Use **try-catch blocks** to handle STL exceptions.
4. Prefer **at() over []** if you want exception safety; [] does not throw.
5. Exception handling in STL **integrates with RAII** — objects clean up automatically.

✅ **Key Takeaways**

* Exception handling is **essential for robust STL code**.
* Most STL components follow **RAII and strong exception guarantees**.
* Use **std::exception hierarchy** to catch and handle errors.

Day 5

Windows Programming with MFC & ATL

MFC architecture and message maps

Creating a simple MFC dialog-based application

Event handling and UI controls

Document/View architecture (overview)

COM basics and ATL’s role

Creating a simple COM component using ATL

ATL Smart pointers (CComPtr, CComBSTR)

ATL vs MFC: When to use what

Using STL/Boost in MFC/ATL applications

Exception safety and RAII in GUI apps

Debugging and memory management tools (basics)

Performance tips and best practices (basics)

Project/Practice Ideas (mini project : Hands On)

**1. Overview of Windows Programming in C++**

* Native Windows applications require **Win32 API**, which is **complex and verbose**.
* To simplify, Microsoft provided **MFC** and **ATL**:

| **Framework** | **Purpose** | **Typical Use** |
| --- | --- | --- |
| **MFC** | Object-oriented wrapper over Win32 API | GUI applications, dialogs, document/view architecture |
| **ATL** | Template-based C++ library | Lightweight COM components, ActiveX controls |

**2. Microsoft Foundation Classes (MFC)**

**2.1 What is MFC?**

* **C++ library** that wraps Windows API into **classes**.
* Simplifies **GUI, file handling, messaging, and document-view architecture**.
* Supports **dialogs, controls, menus, toolbars, and printing**.

**2.2 Key Concepts**

1. **Document/View Architecture**
   * CWinApp → Application class
   * CDocument → Stores data
   * CView → Displays data
   * CFrameWnd → Main window frame
2. **Message Maps**
   * Handle Windows messages (like button clicks).
   * Declared using BEGIN\_MESSAGE\_MAP and END\_MESSAGE\_MAP.
3. **Common Classes**  
   | Class | Purpose |  
   |------------|---------|  
   | CWnd | Base class for all windows/controls |  
   | CDialog | Dialog box support |  
   | CButton | Button control |  
   | CEdit | Text input control |  
   | CListBox | List box control |  
   | CMenu | Menus |  
   | CFrameWnd | Main application window frame |

**2.3 MFC Example: Simple Dialog App**

class CMyDialog : public CDialogEx {

public:

CMyDialog() : CDialogEx(IDD\_MYDIALOG) {}

protected:

afx\_msg void OnBnClickedOk() {

AfxMessageBox(\_T("OK Button Clicked!"));

CDialogEx::OnOK();

}

DECLARE\_MESSAGE\_MAP()

};

BEGIN\_MESSAGE\_MAP(CMyDialog, CDialogEx)

ON\_BN\_CLICKED(IDOK, &CMyDialog::OnBnClickedOk)

END\_MESSAGE\_MAP()

* Here, ON\_BN\_CLICKED maps **button click event** to the function.

**3. Active Template Library (ATL)**

**3.1 What is ATL?**

* Template-based **lightweight C++ library** for **COM (Component Object Model)** objects.
* Provides **ActiveX controls, COM interfaces, and automation support**.
* Smaller and faster than MFC (no GUI framework included by default).

**3.2 Key Concepts**

1. **COM Objects**
   * ATL simplifies creation of COM components.
   * Classes derive from CComObjectRootEx and CComCoClass.
2. **Smart Pointers**
   * CComPtr → manages COM interface pointers automatically.
3. **Macros**  
   | Macro | Purpose |  
   |----------------|---------|  
   | DECLARE\_REGISTRY\_RESOURCEID | Declare registry info for COM |  
   | BEGIN\_COM\_MAP / END\_COM\_MAP | Map interfaces supported by class |  
   | OBJECT\_ENTRY\_AUTO | Register COM object |

**3.3 ATL Example: Simple COM Object**

class ATL\_NO\_VTABLE CSimpleObject :

public CComObjectRootEx<CComSingleThreadModel>,

public CComCoClass<CSimpleObject, &CLSID\_SimpleObject>,

public ISimpleInterface

{

public:

STDMETHOD(Add)(int a, int b, int\* result) {

\*result = a + b;

return S\_OK;

}

DECLARE\_REGISTRY\_RESOURCEID(IDR\_SIMPLEOBJECT)

BEGIN\_COM\_MAP(CSimpleObject)

COM\_INTERFACE\_ENTRY(ISimpleInterface)

END\_COM\_MAP()

};

* Here, ATL handles **COM registration, interface maps, and memory management**.

**4. Key Differences: MFC vs ATL**

| **Feature** | **MFC** | **ATL** |
| --- | --- | --- |
| Primary Use | GUI applications | COM components / ActiveX |
| Size / Overhead | Larger | Lightweight |
| GUI Support | Yes | No (minimal support) |
| Architecture | Document/View model | COM interface-based |
| Template Use | Minimal | Heavy use of C++ templates |

**5. Summary / Key Points**

* **MFC** → Use for **traditional Windows GUI applications** with message maps and document/view architecture.
* **ATL** → Use for **lightweight COM components or ActiveX controls**, template-heavy and efficient.
* Both **simplify native Win32 programming**, but **MFC is GUI-focused**, ATL is **COM-focused**.
* Modern alternatives: **C++/WinRT** and **.NET/CLR** for newer Windows development.

**1. MFC Architecture Overview**

MFC (Microsoft Foundation Classes) is **an object-oriented wrapper over the Win32 API**, designed to simplify Windows GUI programming. Its architecture is heavily based on the **Document/View model**.

**1.1 Core Components**

| **Component** | **Purpose** |
| --- | --- |
| **CWinApp** | Application class; manages app initialization, main message loop, and termination. |
| **CFrameWnd** | Main application window (frame); hosts menus, toolbars, and views. |
| **CDocument** | Represents the **data/model** of the application. |
| **CView** | Responsible for **displaying the data** from the document; handles drawing and user interaction. |
| **CWnd** | Base class for **all windows and controls** (buttons, edit boxes, etc.). |

**1.2 Document/View Architecture**

* **Separation of data and UI:**
  + CDocument → stores the **data**.
  + CView → **visualizes** the data.
* **Advantages:**
  + Multiple views can share the same document.
  + Promotes **modular and maintainable code**.

**Diagram (Textual View):**

+----------------+

| CWinApp |

+----------------+

|

v

+----------------+

| CFrameWnd |

+----------------+

/ \

v v

+----------+ +----------+

| CView1 | | CView2 |

+----------+ +----------+

^

|

+------------+

| CDocument |

+------------+

**1.3 Other Important MFC Classes**

| **Class** | **Description** |
| --- | --- |
| CDialog | Dialog boxes (modal or modeless) |
| CButton | Button control |
| CEdit | Text input control |
| CListBox | List box control |
| CMenu | Menus |
| CToolBar | Toolbar support |

**2. Message Maps**

**2.1 What is a Message Map?**

* Windows OS communicates with applications via **messages** (mouse clicks, key presses, menu selections).
* **Message map** connects **Windows messages** to **member functions** in MFC classes.
* Declared using macros:
  + BEGIN\_MESSAGE\_MAP
  + END\_MESSAGE\_MAP
  + Message handlers like ON\_COMMAND, ON\_BN\_CLICKED, ON\_WM\_PAINT

**2.2 Syntax**

class CMyWnd : public CFrameWnd

{

public:

afx\_msg void OnMyButtonClicked();

DECLARE\_MESSAGE\_MAP()

};

BEGIN\_MESSAGE\_MAP(CMyWnd, CFrameWnd)

ON\_BN\_CLICKED(ID\_MYBUTTON, &CMyWnd::OnMyButtonClicked)

END\_MESSAGE\_MAP()

void CMyWnd::OnMyButtonClicked()

{

AfxMessageBox(\_T("Button Clicked!"));

}

* ON\_BN\_CLICKED(ID\_MYBUTTON, &CMyWnd::OnMyButtonClicked) → Maps **button click** to the function.

**2.3 Common Message Map Macros**

| **Macro** | **Purpose** |
| --- | --- |
| ON\_COMMAND | Handle menu or command events |
| ON\_BN\_CLICKED | Handle button click events |
| ON\_WM\_PAINT | Handle paint/draw messages |
| ON\_WM\_KEYDOWN | Handle key press events |
| ON\_WM\_LBUTTONDOWN | Handle left mouse button click |
| ON\_UPDATE\_COMMAND\_UI | Enable/disable UI elements dynamically |

**2.4 Example: Handling a Menu Command**

BEGIN\_MESSAGE\_MAP(CMyAppView, CView)

ON\_COMMAND(ID\_FILE\_NEW, &CMyAppView::OnFileNew)

ON\_COMMAND(ID\_FILE\_OPEN, &CMyAppView::OnFileOpen)

END\_MESSAGE\_MAP()

void CMyAppView::OnFileNew()

{

AfxMessageBox(\_T("New File Selected!"));

}

**3. Key Points**

1. **MFC Architecture**
   * CWinApp → App lifecycle
   * CFrameWnd → Main window
   * CDocument → Data
   * CView → Presentation of data
2. **Message Maps**
   * Map **Windows messages** to **class member functions**.
   * Central to **event-driven programming** in MFC.
3. **Advantages**
   * Simplifies **Windows programming** using OO concepts.
   * Supports **modular, maintainable GUI apps**.
   * Integrates **dialogs, menus, and controls** seamlessly.

Let’s go **step by step** to create a **simple MFC dialog-based application** in **Visual Studio 2022**. This is the most common way to get started with MFC GUI programming.

**Step 1: Open Visual Studio and Create Project**

1. Open **Visual Studio 2022**.
2. Go to **File → New → Project**.
3. Search for **MFC App** or **MFC Application**.
4. Select **MFC Application** and click **Next**.
5. Give the project a name, e.g., MyDialogApp, and choose a location. Click **Create**.

**Step 2: MFC Application Wizard**

1. In the wizard, choose **Dialog based** (not SDI/MDI).
2. Leave the other settings as default (Windows XP style, Use Unicode).
3. Click **Finish**.

Visual Studio generates a **skeleton MFC dialog project** with a main dialog class, e.g., CMyDialogAppDlg.

**Step 3: Explore Generated Files**

* **CMyDialogApp.cpp** → Main application code.
* **CMyDialogAppDlg.h / .cpp** → Dialog class with **OnInitDialog()**, **DoDataExchange()**, and message maps.
* **Resource.h / MyDialogApp.rc** → Dialog resources (buttons, text boxes, static text).

**Step 4: Add Controls to Dialog**

1. Open **Resource View → Dialogs → IDD\_MYDIALOGAPP\_DIALOG**.
2. Drag controls from the **Toolbox**:
   * Button → e.g., **IDOK**, **IDC\_MYBUTTON**
   * Static text → **IDC\_STATIC**
   * Edit box → **IDC\_EDIT1**
3. Arrange controls as needed.

**Step 5: Add Message Handlers**

1. Right-click a **button** → **Add Event Handler**.
2. Select **BN\_CLICKED** → **Add Handler** → e.g., OnBnClickedMyButton().
3. Visual Studio automatically adds the **message map entry** in CMyDialogAppDlg.cpp:

BEGIN\_MESSAGE\_MAP(CMyDialogAppDlg, CDialogEx)

ON\_BN\_CLICKED(IDC\_MYBUTTON, &CMyDialogAppDlg::OnBnClickedMyButton)

END\_MESSAGE\_MAP()

**Step 6: Implement Button Click**

void CMyDialogAppDlg::OnBnClickedMyButton()

{

CString text;

GetDlgItemText(IDC\_EDIT1, text); // Get text from edit box

text += \_T(" clicked!");

SetDlgItemText(IDC\_STATIC, text); // Show text in static label

}

* GetDlgItemText() → Read value from edit control
* SetDlgItemText() → Display value in static text

**Step 7: Build and Run**

1. Press **F5** or **Ctrl+Shift+B** to build the project.
2. Run the app:
   * Enter text in the edit box.
   * Click the button → see the text updated in the static label.

**Step 8: Key Points**

1. **Dialog-based MFC** is simple for small GUI apps.
2. **Controls** are linked to code via **message maps**.
3. **Data exchange** can also use **DDX/DDV** for automatic binding.
4. Visual Studio **automates a lot** of the Win32 setup via MFC skeleton.

**Optional Enhancements**

* Add **more buttons** with separate handlers.
* Use **MessageBox** for alerts:

AfxMessageBox(\_T("Button Clicked!"));

* Use **ComboBox/ListBox** to select items.
* Implement **OnInitDialog()** to initialize controls at startup.

**1. Event Handling in MFC**

**1.1 How MFC Handles Events**

* Windows OS sends **messages** for events (mouse clicks, key presses, menu selections).
* MFC uses **message maps** to connect messages to **member functions**.

**Message Map Syntax:**

BEGIN\_MESSAGE\_MAP(CMyDialog, CDialogEx)

ON\_BN\_CLICKED(IDC\_MYBUTTON, &CMyDialog::OnMyButtonClicked)

END\_MESSAGE\_MAP()

* ON\_BN\_CLICKED → maps **button click** to OnMyButtonClicked().
* Other message macros:
  + ON\_COMMAND → menu/toolbar commands
  + ON\_WM\_PAINT → paint/draw
  + ON\_WM\_KEYDOWN → key press
  + ON\_LBN\_SELCHANGE → listbox selection

**1.2 Example: Button Click**

void CMyDialog::OnMyButtonClicked()

{

AfxMessageBox(\_T("Button Clicked!"));

}

* When the user clicks the button, this function is called automatically.

**1.3 Event Flow**

User Action (click, key, etc.)

↓

Windows OS generates message (WM\_COMMAND, WM\_KEYDOWN, etc.)

↓

MFC dispatches via message map

↓

Corresponding handler function executes

**2. Common UI Controls in MFC**

| **Control** | **Class** | **Common Usage** |
| --- | --- | --- |
| Button | CButton | Trigger actions |
| Edit box | CEdit | User input text |
| Static text | CStatic | Display text or images |
| List box | CListBox | Display selectable list |
| Combo box | CComboBox | Drop-down selection |
| Check box | CButton (BS\_CHECKBOX style) | Toggle options |
| Radio button | CButton (BS\_RADIOBUTTON style) | Select one option from group |
| Slider / TrackBar | CSliderCtrl | Adjust numeric value |
| Progress bar | CProgressCtrl | Show progress of tasks |

**2.1 Interacting with Controls**

* **Get/Set values**:

CString text;

GetDlgItemText(IDC\_EDIT1, text); // Read edit box

SetDlgItemText(IDC\_STATIC, \_T("Hello")); // Set static text

* **Check state** (checkbox/radio):

BOOL checked = ((CButton\*)GetDlgItem(IDC\_CHECK1))->GetCheck();

* **List/Combo Box**:

CComboBox\* pCombo = (CComboBox\*)GetDlgItem(IDC\_COMBO1);

pCombo->AddString(\_T("Option 1"));

int sel = pCombo->GetCurSel(); // get selected index

**2.2 Example: Button Updates Edit and Static**

void CMyDialog::OnBnClickedUpdate()

{

CString input;

GetDlgItemText(IDC\_EDIT1, input); // Read edit box

input += \_T(" clicked!");

SetDlgItemText(IDC\_STATIC, input); // Update static text

}

* When the button is clicked, the **text from edit box is displayed in static label**.

**3. Dialog Initialization**

* Use **OnInitDialog()** to initialize controls before showing the dialog:

BOOL CMyDialog::OnInitDialog()

{

CDialogEx::OnInitDialog();

SetDlgItemText(IDC\_STATIC, \_T("Welcome!"));

((CButton\*)GetDlgItem(IDC\_CHECK1))->SetCheck(BST\_CHECKED);

return TRUE; // return TRUE unless focus is set manually

}

**4. Key Points**

1. **Message Maps** are central to **event handling**.
2. **Controls** are accessed via **CWnd-derived classes** (CButton, CEdit, CComboBox).
3. Use **GetDlgItemText / SetDlgItemText** for text.
4. **OnInitDialog()** is used for control initialization.
5. Most user interactions (clicks, selections) generate **WM\_COMMAND messages**.

**1. What is Document/View Architecture?**

* **Purpose:** Separates **data (model)** from **presentation (view/UI)**.
* **Document** → stores the application data.
* **View** → displays and interacts with that data.
* **Frame** → main window that hosts the view(s).

**Advantages:**

1. Multiple views can share the same data.
2. Promotes **modular and maintainable code**.
3. Simplifies **printing, serialization, and undo/redo** functionality.

**2. Core Components**

| **Component** | **Class in MFC** | **Role** |
| --- | --- | --- |
| **Application** | CWinApp | Manages app lifecycle, initialization, message loop |
| **Frame Window** | CFrameWnd | Hosts menus, toolbars, status bar, and views |
| **Document** | CDocument | Stores data, handles serialization (save/load) |
| **View** | CView | Displays the document, handles drawing, user interaction |

**3. Relationship Between Components**

+----------------+

| CWinApp |

+----------------+

|

v

+----------------+

| CFrameWnd |

+----------------+

/ \

v v

+----------+ +----------+

| CView1 | | CView2 |

+----------+ +----------+

^

|

+------------+

| CDocument |

+------------+

* **Multiple views** (CView1, CView2) can display the same **document**.
* The **frame window** (CFrameWnd) manages the views and UI elements.

**4. Key Features**

**4.1 Serialization**

* CDocument supports saving/loading data to/from files using Serialize().

void CMyDoc::Serialize(CArchive& ar)

{

if (ar.IsStoring())

ar << m\_data; // save data

else

ar >> m\_data; // load data

}

**4.2 Printing**

* Views handle printing via OnPreparePrinting, OnBeginPrinting, OnDraw.

**4.3 Command Routing**

* Commands (menu, toolbar) are routed automatically from **frame → view → document**.

**4.4 Update/View Notification**

* When document changes, it can notify all views using:

UpdateAllViews(nullptr);

* Views then **redraw themselves** by overriding OnDraw().

**5. Example Workflow**

1. User opens a file → CDocument reads data (serialization).
2. CView displays the data.
3. User edits data → CDocument is updated.
4. UpdateAllViews() → redraws all views.
5. User saves → CDocument::Serialize() writes data to file.

**6. Key Points**

1. Document/View = **Model/View** pattern in MFC.
2. **Document** = Data storage, **View** = Display/interaction, **Frame** = UI container.
3. Supports **multiple views per document**, **serialization**, **printing**, **undo/redo**, and **automatic command routing**.
4. Promotes **separation of concerns** and **maintainable code**.

**1. What is COM (Component Object Model)?**

* **COM** is a **Microsoft technology** for creating **reusable software components** that can communicate across processes and languages.
* **Key idea:** Separate **interface** from **implementation**, allowing objects to interact without knowing each other’s internal details.

**1.1 Key Concepts of COM**

| **Concept** | **Description** |
| --- | --- |
| **COM Object** | Encapsulates functionality; can be reused across apps/languages. |
| **Interface** | Contract exposing methods; **IUnknown** is the base interface for all COM interfaces. |
| **CLSIDs** | Globally unique identifier for a COM class. |
| **IIDs** | Globally unique identifier for a COM interface. |
| **Reference Counting** | COM objects are managed with AddRef() and Release() to control lifetime. |

**1.2 COM Principles**

1. **Language-independent** → can be used from C++, C#, VB, etc.
2. **Binary standard** → allows objects in different processes/machines to interact.
3. **Interface-based programming** → clients interact only via interfaces, not implementations.
4. **Object lifetime** → managed through **reference counting**.

**1.3 Example: IUnknown Interface**

struct IUnknown {

virtual HRESULT QueryInterface(REFIID riid, void\*\* ppv) = 0;

virtual ULONG AddRef() = 0;

virtual ULONG Release() = 0;

};

* QueryInterface → ask object for a specific interface.
* AddRef / Release → manage lifetime.

**2. ATL (Active Template Library) and COM**

**2.1 What is ATL?**

* **ATL** is a **C++ template library** by Microsoft for creating **lightweight COM objects**.
* Provides:
  + Smart pointers (CComPtr) for automatic memory management
  + Macros to simplify **COM registration** and **interface mapping**
  + Support for **ActiveX controls, COM servers, and automation**

**2.2 Why ATL?**

| **Feature** | **Benefit** |
| --- | --- |
| Template-based | Lightweight, fast, compile-time checks |
| COM Smart Pointers | Automatic AddRef/Release (CComPtr) |
| Macros (COM\_MAP, REG) | Simplifies registration & interface implementation |
| Minimal GUI Overhead | Focused on COM, not GUI (unlike MFC) |

**2.3 ATL COM Object Example**

class ATL\_NO\_VTABLE CSimpleObject :

public CComObjectRootEx<CComSingleThreadModel>,

public CComCoClass<CSimpleObject, &CLSID\_SimpleObject>,

public ISimpleInterface

{

public:

STDMETHOD(Add)(int a, int b, int\* result) {

\*result = a + b;

return S\_OK;

}

DECLARE\_REGISTRY\_RESOURCEID(IDR\_SIMPLEOBJECT)

BEGIN\_COM\_MAP(CSimpleObject)

COM\_INTERFACE\_ENTRY(ISimpleInterface)

END\_COM\_MAP()

};

* CComObjectRootEx → provides **AddRef/Release**
* CComCoClass → handles **COM class registration**
* COM\_INTERFACE\_ENTRY → maps interface to class
* STDMETHOD → standard method signature for COM

**2.4 How ATL Simplifies COM**

1. **Memory Management:** Smart pointers (CComPtr) eliminate manual AddRef/Release calls.
2. **Interface Mapping:** Macros (BEGIN\_COM\_MAP) reduce boilerplate.
3. **Class Registration:** DECLARE\_REGISTRY\_RESOURCEID registers COM object in Windows registry.
4. **Threading Models:** Templates handle single-threaded vs multi-threaded models.

**3. Key Points**

1. **COM** → interface-based, language-independent, reference-counted objects.
2. **ATL** → lightweight C++ library to **create and manage COM objects easily**.
3. **ATL advantages** → smart pointers, interface mapping, registry macros, templates for threading and object lifetime.
4. Use ATL when you need **efficient COM components or ActiveX controls** without full MFC overhead.

Let’s go **step by step** to create a **simple COM component using ATL** in **Visual Studio 2022**.

**Step 1: Create an ATL Project**

1. Open **Visual Studio 2022** → **File → New → Project**.
2. Search for **ATL Project** → Select → Click **Next**.
3. Name the project, e.g., SimpleATLComponent.
4. Click **Create**.
5. ATL Project Wizard:
   * Check **Dynamic-Link Library (DLL)**.
   * Check **Allow merging of proxy/stub code** (optional).
   * Click **Finish**.

Visual Studio generates a skeleton ATL project with basic DLL support.

**Step 2: Add an ATL Simple Object**

1. Right-click the project → **Add → Class…**
2. Choose **ATL Simple Object** → Click **Add**.
3. Enter:
   * Short name: SimpleCalc
   * Interfaces: leave default (I simpleCalc)
   * Threading model: Apartment
   * Multi-instance: yes
4. Click **Finish**.

This generates files:

* SimpleCalc.h / SimpleCalc.cpp → class implementation
* ISimpleCalc.idl → interface definition

**Step 3: Implement a Method**

Open SimpleCalc.h and add a simple method in the interface:

// In ISimpleCalc

STDMETHOD(Add)(LONG a, LONG b, LONG\* result) = 0;

Implement it in SimpleCalc.cpp:

STDMETHODIMP CSimpleCalc::Add(LONG a, LONG b, LONG\* result)

{

if (!result) return E\_POINTER;

\*result = a + b;

return S\_OK;

}

**Step 4: Register the COM Component**

1. Build the project.
2. Right-click project → **Properties → Configuration Properties → General**
   * Set **Register Output** = Yes.
3. Build → Visual Studio registers the COM DLL automatically.

Alternatively, you can manually register using regsvr32 SimpleATLComponent.dll.

**Step 5: Use the COM Component in a Client**

Create a **C++ console application**:

#include <atlbase.h>

#include <iostream>

#include "SimpleATLComponent\_i.h" // generated header

int main() {

CoInitialize(NULL);

CComPtr<ISimpleCalc> spCalc;

HRESULT hr = spCalc.CoCreateInstance(CLSID\_SimpleCalc);

if (SUCCEEDED(hr)) {

LONG result;

spCalc->Add(10, 20, &result);

std::cout << "Result: " << result << std::endl;

}

CoUninitialize();

return 0;

}

**Output:**

Result: 30

**Step 6: Key Points**

1. **ATL simplifies COM creation**:
   * Smart pointers (CComPtr) → automatic AddRef/Release
   * Macros (COM\_INTERFACE\_ENTRY) → map interfaces
   * CComCoClass → handles registration
2. **COM basics used**:
   * Reference counting, interface-based programming, apartment threading model
3. Steps summary:
   * Create ATL DLL → Add ATL Simple Object → Implement interface → Build/Register → Use in client

**1. Why Use Smart Pointers in ATL?**

* COM objects use **reference counting** (AddRef / Release).
* Manual management is error-prone → **memory leaks or crashes**.
* **ATL smart pointers** automatically manage lifetime and resource cleanup.

**2. CComPtr**

**2.1 What is CComPtr?**

* Template class that wraps a **COM interface pointer**.
* Automatically calls **AddRef** when assigned and **Release** when going out of scope.
* Eliminates manual AddRef / Release calls.

**2.2 Syntax**

CComPtr<IMyInterface> spMyObject;

spMyObject.CoCreateInstance(CLSID\_MyObject);

spMyObject->SomeMethod();

**2.3 Example**

#include <atlbase.h>

#include <iostream>

#include "SimpleATLComponent\_i.h" // generated header

int main() {

CoInitialize(NULL);

CComPtr<ISimpleCalc> spCalc;

HRESULT hr = spCalc.CoCreateInstance(CLSID\_SimpleCalc);

if (SUCCEEDED(hr)) {

LONG result;

spCalc->Add(5, 7, &result);

std::cout << "Result: " << result << std::endl;

} // spCalc.Release() is automatic here

CoUninitialize();

return 0;

}

* spCalc automatically calls Release() when it goes out of scope.

**2.4 Common Operations**

| **Operation** | **Notes** |
| --- | --- |
| CoCreateInstance | Creates COM object |
| Attach(pInterface) | Attach raw pointer (takes ownership) |
| Detach() | Detach raw pointer (releases ownership) |
| -> | Access methods of COM interface |

**3. CComBSTR**

**3.1 What is CComBSTR?**

* ATL wrapper for **BSTR** (COM string type).
* Automatically manages allocation and deallocation.
* Can be used with **COM properties, automation, and VARIANTs**.

**3.2 Syntax**

CComBSTR bstrText(L"Hello COM");

BSTR rawBstr = bstrText; // implicit conversion

**3.3 Example**

#include <atlbase.h>

#include <atlcomcli.h>

#include <iostream>

int main() {

CComBSTR name(L"John Doe");

std::wcout << L"Name: " << static\_cast<BSTR>(name) << std::endl;

// Assignment

CComBSTR another;

another = L"ATL BSTR Example";

std::wcout << L"Another: " << static\_cast<BSTR>(another) << std::endl;

return 0;

}

**Key points:**

* CComBSTR automatically calls SysAllocString and SysFreeString.
* Supports assignment, concatenation, and conversion to BSTR.

**4. Advantages of ATL Smart Pointers**

| **Feature** | **Benefit** |
| --- | --- |
| CComPtr | Automatic AddRef/Release; safer COM pointer management |
| CComBSTR | Automatic allocation/deallocation of BSTR strings |
| Templates | Works for any COM interface type |
| Operator overloads | Easy to use like regular pointers/strings |
| Exception safety | Automatically cleans up if exceptions occur |

**5. Key Points**

1. COM programming requires careful **reference counting** and memory management.
2. **ATL smart pointers** (CComPtr, CComBSTR) automate this and reduce errors.
3. CComPtr → wraps COM interface pointers (IUnknown derived).
4. CComBSTR → wraps COM string (BSTR) safely.
5. Using them ensures **exception safety** and **resource leak prevention**.

Here’s a detailed comparison of **ATL vs MFC** to help decide **when to use each**:

**1. Overview**

| **Framework** | **Purpose** | **Typical Usage** |
| --- | --- | --- |
| **MFC (Microsoft Foundation Classes)** | Object-oriented wrapper over Win32 API | GUI applications with dialogs, menus, toolbars, document/view apps |
| **ATL (Active Template Library)** | Template-based C++ library for COM | Lightweight COM objects, ActiveX controls, automation |

**2. Key Differences**

| **Feature** | **MFC** | **ATL** |
| --- | --- | --- |
| **Primary Use** | Building **full-featured Windows GUI apps** | Creating **COM components** or **ActiveX controls** |
| **Size / Overhead** | Larger; includes GUI and framework features | Lightweight; template-based, minimal overhead |
| **GUI Support** | Extensive (dialogs, views, controls, document/view) | Minimal; mostly for COM objects (no full GUI framework) |
| **COM Support** | Can wrap COM, but verbose | Built-in support for COM, smart pointers, interface mapping |
| **Document/View Architecture** | Yes, natively supports it | No; ATL is component-focused |
| **Memory Management** | Manual or via MFC classes (CWnd, CString, etc.) | Automatic via smart pointers (CComPtr, CComBSTR) |
| **Threading Models** | Managed manually | Templates handle single/multi-threaded models (CComSingleThreadModel, etc.) |
| **Complexity** | Easier for GUI, more heavyweight | Lightweight, requires understanding COM and templates |

**3. When to Use MFC**

* You are building a **traditional Windows GUI application** (dialog-based, SDI, MDI).
* You need **document/view architecture** for multi-document apps.
* You want **easy integration of menus, toolbars, and dialogs**.
* You don’t want to deal directly with COM internals.

**Example:**

* A text editor, paint application, or database GUI tool.

**4. When to Use ATL**

* You are building a **COM component**, **ActiveX control**, or **automation server**.
* You want a **lightweight DLL** without GUI overhead.
* You need **automatic reference counting and smart pointer support**.
* You want **fine-grained control over interfaces and threading**.

**Example:**

* A COM library for calculations, a reusable ActiveX chart control, or a plugin for another application.

**5. Summary**

| **Criteria** | **Use MFC** | **Use ATL** |
| --- | --- | --- |
| GUI Application | ✅ | ❌ (requires extra work) |
| COM Object / ActiveX | ❌ (verbose) | ✅ (native support) |
| Lightweight / Fast | ❌ | ✅ |
| Document/View | ✅ | ❌ |
| Smart Pointer & String Management | Limited (manual) | ✅ (CComPtr, CComBSTR) |
| Learning Curve | Moderate | Steeper (templates + COM) |

**Rule of Thumb:**

* **MFC** → Full Windows applications with GUI.
* **ATL** → Component development (COM/ActiveX) without full GUI.

**1. Why Use STL or Boost with MFC/ATL?**

* **MFC**: Provides its own classes like CString, CArray, CList.
* **ATL**: Minimal framework, primarily for COM; relies on C++ features.
* **STL (Standard Template Library)** and **Boost** offer:
  + **Modern C++ containers** (vector, map, set, unordered\_map)
  + **Algorithms** (sort, find\_if, accumulate)
  + **Smart pointers** (unique\_ptr, shared\_ptr)
  + **Utility libraries** (Boost.Regex, Boost.Asio, Boost.Filesystem)

**Advantages:**

1. Type-safe and standardized.
2. Easier memory management and resource handling.
3. Cross-platform compatible code (especially Boost).

**2. Using STL in MFC/ATL Applications**

**2.1 Containers**

#include <vector>

#include <map>

#include <string>

void ExampleSTL()

{

std::vector<CString> names;

names.push\_back(\_T("Alice"));

names.push\_back(\_T("Bob"));

std::map<int, CString> idNameMap;

idNameMap[1] = \_T("Alice");

idNameMap[2] = \_T("Bob");

for (auto& name : names)

AfxMessageBox(name);

}

* STL containers work **well with ATL and MFC classes** like CString.
* Use \_T() macro to ensure **Unicode/ANSI compatibility**.

**2.2 Algorithms**

#include <algorithm>

#include <vector>

std::vector<int> numbers = {1, 2, 3, 4, 5};

auto it = std::find\_if(numbers.begin(), numbers.end(), [](int n){ return n > 3; });

if (it != numbers.end()) {

AfxMessageBox(std::to\_wstring(\*it).c\_str());

}

* Standard algorithms (sort, find\_if, accumulate) simplify common operations.

**3. Using Boost in MFC/ATL Applications**

**3.1 Common Boost Libraries**

| **Library** | **Use Case** |
| --- | --- |
| Boost.SmartPtr | Shared, weak, scoped pointers (alternative to STL smart pointers) |
| Boost.Regex | Powerful regular expressions |
| Boost.Filesystem | Platform-independent file/directory handling |
| Boost.Asio | Networking and asynchronous IO |
| Boost.Bind/Lambda | Callbacks, event handling in GUI or COM apps |

**3.2 Example: Boost.Regex with MFC**

#include <boost/regex.hpp>

#include <afxwin.h>

void ValidateEmail(const CString& email)

{

boost::wregex pattern(LR"((\w+)(\.\w+)\*@(\w+\.)+(\w+))");

if (boost::regex\_match(email.GetString(), pattern))

AfxMessageBox(\_T("Valid Email"));

else

AfxMessageBox(\_T("Invalid Email"));

}

* Works seamlessly in **MFC dialog applications**.

**4. Best Practices**

1. **String Compatibility**
   * MFC: CString
   * STL: std::wstring or convert using CString::GetString()
   * Boost: Usually supports std::wstring or CString conversion
2. **Memory Management**
   * Prefer **smart pointers** (CComPtr, unique\_ptr, shared\_ptr) to avoid leaks.
3. **Mixing MFC Containers with STL**
   * Avoid mixing MFC collection classes (CArray, CList) with STL containers unless necessary.
4. **Thread Safety**
   * Boost.Thread or STL threads can be used in ATL COM servers.
   * Be cautious with MFC GUI elements (must update GUI on main thread).

**5. Example: STL Vector with MFC Dialog**

void CMyDialog::ShowNames()

{

std::vector<CString> names = { \_T("Alice"), \_T("Bob"), \_T("Charlie") };

for (auto& name : names)

AfxMessageBox(name);

}

* Simple integration: **STL containers** for data, **MFC GUI** for display.

**6. Key Points**

1. **MFC** → Use STL for data handling and algorithms; GUI handled via MFC.
2. **ATL** → STL and Boost essential for building **lightweight COM components**.
3. **Boost** → Adds extra functionality (regex, filesystem, smart pointers, async I/O).
4. Always ensure **string type compatibility** and **memory management** when mixing MFC/ATL with STL/Boost.

Here’s a detailed explanation of **Exception Safety and RAII in GUI applications**, especially when using **C++ with MFC or ATL**:

**1. What is RAII?**

* **RAII (Resource Acquisition Is Initialization)** is a C++ idiom:
  + **Acquire resources in a constructor**
  + **Release resources in a destructor**
* Ensures **automatic cleanup**, even if exceptions occur.

**Resources include:**

* Memory (new/delete)
* File handles (FILE\*)
* GDI objects (CBrush, CPen)
* COM objects (CComPtr)

**2. Why Exception Safety is Important in GUI Apps**

* GUI apps often interact with:
  + **File I/O** (loading/saving settings)
  + **Network or database operations**
  + **COM objects**
  + **GDI resources** (pens, brushes, bitmaps)
* If an exception occurs and resources are not released:
  + Memory leaks
  + Resource exhaustion
  + Crashes or undefined behavior

**3. Exception Safety Levels**

1. **No-throw guarantee:** Function guarantees it will never throw.
2. **Strong guarantee:** Function either completes successfully or **state is unchanged**.
3. **Basic guarantee:** Function ensures **object remains in valid state**, even if modified.

In GUI apps, **strong guarantee** is desirable for operations like file save, network requests, or updating UI data.

**4. Using RAII in GUI Apps**

**4.1 Memory Management**

std::unique\_ptr<CMyData> pData(new CMyData());

* Automatically deletes pData if function throws.

**4.2 COM Objects**

CComPtr<ISimpleCalc> spCalc;

spCalc.CoCreateInstance(CLSID\_SimpleCalc);

spCalc->Add(5, 10, &result); // Automatic Release() when out of scope

**4.3 GDI Objects**

CBrush brush(RGB(255,0,0));

CClientDC dc(this);

CBrush\* pOldBrush = dc.SelectObject(&brush);

// RAII wrapper example

class CBrushGuard {

CBrush\* m\_pOld;

CDC\* m\_pDC;

public:

CBrushGuard(CDC\* dc, CBrush\* brush) : m\_pDC(dc) {

m\_pOld = dc->SelectObject(brush);

}

~CBrushGuard() { m\_pDC->SelectObject(m\_pOld); }

};

CBrushGuard guard(&dc, &brush); // Automatic restore even if exception

**5. Exception-Safe GUI Patterns**

**5.1 File I/O in Dialog**

try {

CFile file(\_T("data.txt"), CFile::modeRead);

CArchive ar(&file, CArchive::load);

MyData.Serialize(ar); // Exception safe

} catch (CFileException\* e) {

TCHAR msg[256];

e->GetErrorMessage(msg, 256);

AfxMessageBox(msg);

e->Delete();

}

**5.2 STL Containers for Data**

* Use **vector/map** instead of raw arrays:

std::vector<CString> items;

items.push\_back(\_T("Item1")); // Automatically cleaned if exception occurs

**5.3 Updating UI**

* Minimize direct state changes in UI until all operations succeed.
* Example: update model first, then call UpdateData(FALSE) in MFC.

**6. Key Principles**

1. **Wrap all resources in RAII objects**:
   * Memory → unique\_ptr / shared\_ptr
   * GDI → custom RAII wrappers
   * COM → CComPtr
2. **Use try/catch blocks for risky operations**:
   * File I/O, network, COM calls
3. **Separate model updates from UI updates**:
   * Modify data structures first
   * Update GUI only after successful completion
4. **Avoid raw pointers and manual cleanup** in GUI code.

**7. Summary Table**

| **Concept** | **Example in GUI App** |
| --- | --- |
| RAII for memory | std::unique\_ptr<CMyData> |
| RAII for COM | CComPtr<ISimpleCalc> |
| RAII for GDI | Custom CBrushGuard |
| Exception safety | try/catch around file/network operations |
| STL for containers | std::vector<CString> instead of raw arrays |

This ensures that your **MFC/ATL GUI applications remain robust**, even in the presence of exceptions, without leaking resources.

**1. Debugging Basics in Visual Studio**

**1.1 Setting Breakpoints**

* Click **left margin** next to a line or press F9.
* Program stops at the breakpoint during execution.
* Useful for checking **program flow** and variable values.

**1.2 Step Execution**

| **Command** | **Shortcut** | **Description** |
| --- | --- | --- |
| Step Into | F11 | Enter function calls |
| Step Over | F10 | Execute function without entering |
| Step Out | Shift+F11 | Exit current function |
| Continue | F5 | Resume execution until next breakpoint |

**1.3 Watch and Locals Window**

* **Watch Window:** Monitor specific variables or expressions.
* **Locals Window:** Shows all variables in current scope.
* **QuickWatch (Ctrl+Alt+Q):** Evaluate expressions on the fly.

**1.4 Call Stack**

* Displays the sequence of function calls leading to current point.
* Essential for tracing **crashes or unexpected behavior**.

**2. Memory Management Tools in Visual Studio**

**2.1 Built-in Debug Features**

* Enable **CRT debug heap** to track memory leaks:

#define \_CRTDBG\_MAP\_ALLOC

#include <crtdbg.h>

int main() {

\_CrtSetDbgFlag(\_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF);

int\* p = new int(5);

// Intentionally not deleting p to test leak detection

}

* On program exit, Visual Studio reports **memory leaks with allocation number**.

**2.2 Using Smart Pointers**

* **std::unique\_ptr / std::shared\_ptr** → automatic cleanup
* **CComPtr** → automatic COM object reference management
* **CComBSTR** → automatic BSTR cleanup

**2.3 GDI and GUI Resource Management**

* Use RAII wrappers for CBrush, CPen, CBitmap.
* Prevent **resource leaks** in MFC dialogs.

**3. ATL / COM-Specific Memory Management**

* COM objects require **reference counting**: AddRef / Release.
* Using CComPtr ensures proper **automatic Release()**.
* Avoid circular references; consider CComQIPtr or weak\_ptr patterns.

**4. Profiling Tools in Visual Studio**

* **Diagnostic Tools (Ctrl+Alt+F2):** Track memory usage, CPU usage, and GPU usage.
* **Memory Usage Window:** Snapshot memory allocations and analyze leaks.
* **Performance Profiler:** Identify slow functions and bottlenecks.

**5. Third-Party Tools**

* **Valgrind** (Linux/WSL) → Detect memory leaks and invalid accesses.
* **Deleaker / Dr. Memory / Visual Leak Detector** → Windows memory leak detection.
* **Application Verifier** → Detect heap corruption, handle leaks, and GDI resource leaks.

**6. Best Practices**

1. **Always use RAII / smart pointers** in MFC/ATL GUI apps.
2. **Check for memory leaks in debug mode** regularly.
3. **Release COM objects properly**; use CComPtr.
4. **Wrap GDI resources** to avoid handle leaks.
5. **Use Watch and Call Stack** to debug runtime crashes.

**7. Summary Table**

| **Tool/Feature** | **Use Case** |
| --- | --- |
| Breakpoints, Step Over/Into | Trace code execution |
| Watch / Locals / QuickWatch | Inspect variable values |
| Call Stack | Trace function calls |
| \_CrtDbgFlag / CRT Debug Heap | Detect memory leaks |
| Smart pointers (unique\_ptr, CComPtr) | Automatic memory management |
| RAII wrappers (GDI, COM) | Resource safety |
| Diagnostic Tools / Profiler | CPU/memory profiling |
| Third-party leak detectors | Advanced memory tracking |

**1. General C++ Performance Tips**

| **Tip** | **Description** |
| --- | --- |
| **Use references/pointers instead of copies** | Pass large objects by const & to avoid unnecessary copying. |
| **Prefer emplace over insert** | std::vector::emplace\_back() constructs in-place, reducing copy overhead. |
| **Avoid unnecessary dynamic memory allocations** | Reuse objects or use stack allocation where possible. |
| **Minimize expensive operations in loops** | Move calculations or memory allocations outside loops. |
| **Use reserve() for STL containers** | Pre-allocate memory to reduce vector resizing overhead. |
| **Prefer std::array or std::vector over raw arrays** | Safer, cache-friendly, and easier to maintain. |

**2. STL/Boost Specific Tips**

| **Tip** | **Description** |
| --- | --- |
| **Use appropriate containers** | std::vector for sequential access, std::unordered\_map for fast lookup, std::list only when frequent insertions/deletions needed. |
| **Use std::move / move semantics** | Avoid unnecessary copies when returning or inserting objects. |
| **Boost smart pointers** | shared\_ptr for shared ownership, unique\_ptr for exclusive ownership → safer and often faster than manual memory management. |
| **Boost algorithms** | Use boost::algorithm functions for efficient string and container operations. |

**3. MFC-Specific Tips**

| **Tip** | **Description** |
| --- | --- |
| **Use CString efficiently** | Avoid repeated concatenations; use CString::Format or CString::Append carefully. |
| **Minimize frequent redraws** | Use InvalidateRect judiciously; avoid unnecessary OnPaint calls. |
| **Optimize GDI usage** | Select only necessary pens/brushes; release GDI objects promptly. |
| **Use double-buffering** | Reduce flicker in complex GUI rendering. |

**4. ATL / COM Specific Tips**

| **Tip** | **Description** |
| --- | --- |
| **Use CComPtr** | Automatic reference counting avoids unnecessary AddRef/Release calls. |
| **Avoid circular references** | Prevent memory leaks in COM by breaking cycles or using weak references. |
| **Apartment vs. free-threaded models** | Use the correct threading model to avoid locking overhead. |
| **Keep COM methods lightweight** | Avoid heavy computations in COM method calls; offload to worker threads if needed. |

**5. Multi-threading Tips**

* Use **worker threads** for long-running tasks in GUI apps to prevent freezing.
* Use **thread-safe STL containers or synchronization primitives** when sharing data.
* Avoid frequent locking; minimize contention.
* Use **thread pools** for repeated short tasks (Boost.Asio, std::async).

**6. File I/O and Resource Management**

* Read/write in **large blocks** rather than single bytes.
* Use memory-mapped files for large datasets if applicable.
* Always release handles and resources promptly using RAII.

**7. Profiling and Measuring Performance**

* Use **Visual Studio Profiler** to identify bottlenecks.
* Use **Diagnostic Tools** for memory and CPU usage.
* Use **timing utilities** (std::chrono) for measuring execution time.

#include <chrono>

auto start = std::chrono::high\_resolution\_clock::now();

// Code to measure

auto end = std::chrono::high\_resolution\_clock::now();

std::cout << "Duration: "

<< std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count()

<< " microseconds\n";

**8. Summary Table**

| **Area** | **Tip** |
| --- | --- |
| General C++ | Use references, move semantics, minimize copies |
| STL/Boost | Use efficient containers, emplace, smart pointers |
| MFC GUI | Minimize redraws, optimize GDI, use double-buffering |
| ATL/COM | CComPtr, avoid circular refs, lightweight methods |
| Multi-threading | Worker threads, thread-safe containers, thread pools |
| I/O | Read/write in blocks, RAII for handles |
| Profiling | Measure before optimizing, use Visual Studio tools |

These practices ensure **efficient, maintainable, and safe C++ applications**, whether GUI-based (MFC), COM-based (ATL), or STL/Boost heavy.

**1. MFC-Based GUI Mini Projects**

| **Project Idea** | **Description** | **Key Concepts** |
| --- | --- | --- |
| **Simple Calculator** | Dialog-based app with basic arithmetic | MFC dialogs, controls, event handling |
| **To-Do List App** | Add, edit, delete, and save tasks | STL vectors, file I/O, CString |
| **Text File Viewer/Editor** | Open, edit, and save text files | MFC file dialogs, CString, RAII for file handling |
| **Image Viewer** | Load and display images | GDI+, RAII for resources, exception handling |
| **Mini Paint App** | Draw shapes, select colors, save canvas | MFC mouse events, GDI objects, double-buffering |

**2. ATL / COM Mini Projects**

| **Project Idea** | **Description** | **Key Concepts** |
| --- | --- | --- |
| **COM Calculator Component** | Implement addition, subtraction, multiplication, division | ATL, CComPtr, interface-based programming |
| **COM String Utilities** | Reverse string, count words, validate email | ATL, CComBSTR, exception safety |
| **ActiveX Chart Control** | Display a bar chart using COM | ATL, GDI, smart pointers |
| **COM Logger** | Log messages to file or console via COM interface | RAII, file I/O, ATL smart pointers |

**3. STL & Boost Practice Projects**

| **Project Idea** | **Description** | **Key Concepts** |
| --- | --- | --- |
| **Student Management System** | Store student info, search, sort, display | STL containers (vector, map), algorithms, RAII |
| **File Search Tool** | Search files by pattern in directory | Boost.Filesystem, Boost.Regex |
| **Simple HTTP Client** | Fetch webpage content | Boost.Asio (networking), RAII |
| **Text Analyzer** | Count word frequency from a file | STL map/unordered\_map, algorithms, Boost.Regex |

**4. Combined Projects (MFC + STL/Boost/ATL)**

| **Project Idea** | **Description** | **Key Concepts** |
| --- | --- | --- |
| **Email Validator GUI** | Enter emails in dialog, validate using Boost.Regex | MFC GUI, Boost.Regex, CString/std::wstring |
| **File Organizer** | Move files into folders based on extension | MFC GUI, Boost.Filesystem, STL containers |
| **Mini Banking App** | Accounts with deposit/withdraw, save/load from file | MFC dialogs, STL vectors/maps, RAII, exception safety |
| **COM Calculator with GUI** | GUI calls ATL COM component for calculations | MFC dialogs, ATL COM, CComPtr, smart pointers |

**5. Hands-On Concepts Covered**

* **MFC:** Dialogs, controls, message handling, GDI objects.
* **ATL:** COM objects, interfaces, CComPtr, CComBSTR, reference counting.
* **STL:** Containers (vector, map, set), algorithms (sort, find\_if), smart pointers.
* **Boost:** Filesystem, Regex, Asio, smart pointers.
* **RAII & Exception Safety:** Smart pointers, GDI/COM wrappers, try/catch.
* **Performance:** Minimize copies, memory management, STL efficiency.

**Tip for Hands-On Practice**

1. Pick one **GUI project** (MFC) and one **COM/STL/Boost project**.
2. Implement **incrementally**: start with core logic, then GUI or COM wrapper.
3. Use **smart pointers and RAII** to handle resources safely.
4. Add **file I/O or networking** to make it more practical.

**Mini Project: Student Management GUI Application**

**Goal:**  
Create a dialog-based MFC application to **add, view, search, and delete students**. Students are stored in memory using **STL vector**. GUI uses **MFC controls**, and memory is safely managed with **smart pointers**.

**Step 1: Create MFC Dialog-Based Project**

1. Open **Visual Studio 2022** → **File → New → Project**.
2. Choose **MFC App** → Next.
3. Set **Project Name:** StudentManager.
4. Choose **Dialog-based** → Finish.

**Step 2: Design the Dialog**

**Add controls to the dialog:**

* **CListBox** → IDC\_LIST\_STUDENTS (Display student names)
* **CEdit** → IDC\_EDIT\_NAME (Input student name)
* **CButton** → IDC\_BUTTON\_ADD (Add student)
* **CButton** → IDC\_BUTTON\_DELETE (Delete selected student)
* **CButton** → IDC\_BUTTON\_SEARCH (Search student)

**Step 3: Define Student Structure**

struct Student {

int id;

CString name;

};

**Step 4: Define STL Container with Smart Pointers**

#include <vector>

#include <memory>

std::vector<std::unique\_ptr<Student>> students;

int nextId = 1;

* Using unique\_ptr ensures **automatic memory cleanup**.

**Step 5: Implement Add Student Function**

**Dialog header (StudentManagerDlg.h):**

afx\_msg void OnBnClickedButtonAdd();

**Dialog cpp (StudentManagerDlg.cpp):**

void CStudentManagerDlg::OnBnClickedButtonAdd()

{

CString name;

GetDlgItemText(IDC\_EDIT\_NAME, name);

if (name.IsEmpty()) {

AfxMessageBox(\_T("Enter student name"));

return;

}

auto student = std::make\_unique<Student>();

student->id = nextId++;

student->name = name;

students.push\_back(std::move(student));

// Update list box

CListBox\* pList = (CListBox\*)GetDlgItem(IDC\_LIST\_STUDENTS);

pList->AddString(name);

SetDlgItemText(IDC\_EDIT\_NAME, \_T(""));

}

**Step 6: Implement Delete Student Function**

void CStudentManagerDlg::OnBnClickedButtonDelete()

{

CListBox\* pList = (CListBox\*)GetDlgItem(IDC\_LIST\_STUDENTS);

int sel = pList->GetCurSel();

if (sel == LB\_ERR) {

AfxMessageBox(\_T("Select a student to delete"));

return;

}

students.erase(students.begin() + sel); // Remove from vector

pList->DeleteString(sel); // Remove from listbox

}

**Step 7: Implement Search Student Function**

void CStudentManagerDlg::OnBnClickedButtonSearch()

{

CString searchName;

GetDlgItemText(IDC\_EDIT\_NAME, searchName);

if (searchName.IsEmpty()) {

AfxMessageBox(\_T("Enter student name to search"));

return;

}

bool found = false;

for (auto& student : students) {

if (student->name.CompareNoCase(searchName) == 0) {

found = true;

break;

}

}

if (found)

AfxMessageBox(\_T("Student found!"));

else

AfxMessageBox(\_T("Student not found."));

}

**Step 8: Connect Buttons to Handlers**

1. Open **Dialog Editor** → Right-click **Add Button** → **Add Event Handler**.
2. Assign BN\_CLICKED to each function:
   * Add → OnBnClickedButtonAdd
   * Delete → OnBnClickedButtonDelete
   * Search → OnBnClickedButtonSearch

**Step 9: Build and Run**

* Press **F5** to run.
* Test adding, deleting, and searching students.

**Step 10: Key Concepts Covered**

| **Concept** | **Implementation** |
| --- | --- |
| MFC Dialog | Used for GUI input/output |
| STL Vector | Stores students in memory |
| Smart Pointers | unique\_ptr avoids memory leaks |
| CString | Handles text in MFC |
| Event Handling | Buttons linked to handler functions |
| Exception Safety | RAII via unique\_ptr |

✅ **Result:**  
You have a **fully functional mini Student Management system** in MFC using **modern C++ practices**, **STL containers**, and **smart pointers**.