Introduction to C++

What is C++?

- C++ is a **general-purpose, object-oriented programming language**.
- It is an extension of the C programming language with added features like **classes, inheritance, polymorphism**, and **encapsulation**.
- C++ is **compiled**, **statically typed**, and known for its **performance and control** over system resources.
- It is widely used in **game development, embedded systems, high-performance applications**, and **system programming**.

Features of C++

1. **Object-Oriented**:
    - Supports classes, objects, inheritance, polymorphism, and encapsulation.

    ```
    class Car {
    public:
        void drive() {
            cout << "Driving...";
        }
    };

    int main() {
        Car c;
        c.drive();
        return 0;
    }
    // Output: Driving...
    ```

2. **Compiled Language**:
    - C++ code is compiled into machine code, which makes it faster than interpreted languages.
3. **Low-level Manipulation**:
    - Direct memory access via pointers.

    ```
    int a = 10;
    int* p = &a;
    cout << *p; // Output: 10
    ```

4. **Platform Independent**:
    - The source code is platform-independent but needs recompilation for each OS.

5. **Rich Library Support**:
    - Includes the Standard Template Library (STL) with built-in data structures and algorithms.
6. **Strong Type Checking**:
    - Variables must be declared with a specific type.

Applications of C++

- **Game Development**: Unreal Engine uses C++.
- **Operating Systems**: Windows and parts of macOS.
- **Browsers**: Chrome and Firefox use C++ for rendering engines.
- **Finance and Trading Platforms**: High-frequency trading apps.
- **Embedded Systems**: IoT and microcontroller programming.

Installing C++

On Windows:

1. Download and install Code::Blocks or Dev C++ or install **MinGW** and use **G++**.
2. Set the environment variable for G++.
3. Use any text editor (VS Code, Sublime) and compile using terminal:
4. g++ filename.cpp -o output
5. ./output

On Linux (Ubuntu):

```
sudo apt update
sudo apt install g++
g++ version.cpp -o version
./version
```

First C++ Program

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Output:

```
Hello, World!
```

Explanation:

- #include<iostream>: Includes the input-output stream.
- using namespace std;: Avoids prefixing with std::.
- int main(): Entry point of the program.
- cout: Used to print output.
- return 0;: Indicates successful program termination.

Output:

```
Hello, World!
```

C++ Modules, Comments, and Package Management

□ Modules in C++

What is a Module in C++?

- Modules in C++ allow code to be organized into logical units.
- You can separate your program into **multiple files** for easier maintenance and reusability.
- These files can include **functions, classes, variables**, and **headers**.

How to Use Modules in C++ (Before C++20)

Traditionally, C++ used **header files (.h)** and **source files (.cpp)** to implement modular code.

Example:

mathutils.h

```
#ifndef MATHUTILS_H
#define MATHUTILS_H

int add(int a, int b);

#endif
```

mathutils.cpp

```
#include "mathutils.h"
```

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

main.cpp

```
            #include <iostream>
            #include "mathutils.h"
            using namespace std;

            int main() {
                cout << add(5, 3); // Output: 8
                return 0;
            }
```

C++20 Modules (Modern Approach)

C++20 introduces a **module keyword** to officially support modular programming.

```
// math.ixx
export module math;
export int square(int x) {
    return x * x;
}
// main.cpp
import math;
#include <iostream>
using namespace std;

int main() {
    cout << square(5); // Output: 25
    return 0;
}
```

Note: You need a compiler that supports C++20 modules (like MSVC, GCC 10+).

Comments in C++

Purpose of Comments:

- Improve code readability.
- Leave notes or reminders.
- Help other developers understand your code.

Single-line Comment:

- Starts with `//`

```
// This is a single-line comment
cout << "Hello"; // Prints Hello
```

Multi-line Comment:

- Starts with `/*` and ends with `*/`

```
/*
This is a multi-line comment.
Used for longer explanations.
*/
```

Comments are ignored by the compiler and have no effect on execution.

C++ Package Management

C++ does **not** have a universal package manager like Python's pip. However, several tools exist:

Common Tools:

1. **vcpkg** (by Microsoft)
2. **Conan** (open-source, popular in large C++ projects)
3. **CMake FetchContent** (built-in to CMake)

Using vcpkg:

```
./vcpkg install boost
./vcpkg install fmt
```

Using Conan:

- Create `conanfile.txt`

```
[requires]
fmt/8.0.1

[generators]
cmake
```

- Then run:

```
conan install .
```

These tools help you manage libraries like Boost, fmt, OpenCV, etc.

Summary Table

Concept	Python Equivalent	C++ Equivalent
Module Import	`import math`	`#include "mathutils.h"`
User Module	Custom .py file	Header + Source files
Single Comment	`# comment`	`// comment`
Multi Comment	`''' text '''` (not real comment)	`/* text */`
Package Manager	pip	vcpkg / Conan / CMake

C++: Variables, Data Types, and Keywords

1. Variables in C++

Definition:

Variables are containers for storing data values. In C++, each variable must be declared with a **data type**.

Syntax:

```
datatype variable_name = value;
```

Example:

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

int main() {
    int age = 21;
    float pi = 3.14;
    char grade = 'A';

    cout << "Age: " << age << endl;
    cout << "Pi: " << pi << endl;
    cout << "Grade: " << grade << endl;
    return 0;
}
```

Output:

```
Age: 21
Pi: 3.14
Grade: A
```

Notes:

- Variable names are **case-sensitive**.
- Must start with a **letter or underscore**, not a number.

---

2. Data Types in C++

Categories:

1. **Primary (Built-in)**
2. **Derived**
3. **User-defined**

Basic Data Types:

Type	Description	Example
`int`	Integer numbers	`int a = 10;`
`float`	Floating point numbers	`float pi = 3.14;`
`double`	Double precision float	`double d = 3.14159;`
`char`	Single character	`char ch = 'A';`
`bool`	Boolean (true/false)	`bool flag = true;`
`void`	No value (used in functions)	`void print();`

Example with all:

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

int main() {
    int age = 21;
    float height = 5.9;
    char grade = 'A';
    bool pass = true;

    cout << "Age: " << age << endl;
    cout << "Height: " << height << endl;
    cout << "Grade: " << grade << endl;
    cout << "Pass: " << pass << endl;
    return 0;
}
```

Output:

```
Age: 21
Height: 5.9
Grade: A
Pass: 1
```

---

3. Keywords in C++

What are Keywords?

Keywords are **reserved words** used by the compiler for specific purposes.

List of Common Keywords:

```
auto       break      case       char       const
continue   default    do         double     else
enum       extern     float      for        goto
if         int        long       register   return
short      signed     sizeof     static     struct
switch     typedef    union      unsigned   void
volatile   while      bool       class      namespace
new        delete     public     private    protected
this       throw      try        catch      virtual
```

Example:

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

int main() {
    int number = 100;  // 'int' is a keyword
    return 0;          // 'return' is a keyword
}
```

Note:

- You **cannot use keywords as variable names**.

C++ Operators – Explained with Examples

---

1. Arithmetic Operators

These are used to perform mathematical operations.

Operator	**Description**	**Example**
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`

Operator	**Description**	**Example**
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (Remainder)	`a % b`

Example:

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

int main() {
    int a = 10, b = 3;
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl;
    cout << "a % b = " << a % b << endl;
    return 0;
}
```

Output:

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
```

---

2. Comparison (Relational) Operators

Used to compare two values. Result is `true (1)` or `false (0)`.

Operator	**Description**	**Example**
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater or equal	`a >= b`
`<=`	Less or equal	`a <= b`

Example:

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

int main() {
    int a = 5, b = 10;
    cout << (a == b) << endl; // 0
    cout << (a != b) << endl; // 1
    cout << (a < b) << endl;  // 1
    cout << (a > b) << endl;  // 0
    return 0;
}
```

Output:

```
0
1
1
0
```

---

3. Assignment Operators

Used to assign values to variables.

Operator	Example	Same As
`=`	`a = 10;`	—
`+=`	`a += b;`	`a = a + b`
`-=`	`a -= b;`	`a = a - b`
`*=`	`a *= b;`	`a = a * b`
`/=`	`a /= b;`	`a = a / b`
`%=`	`a %= b;`	`a = a % b`

Example:

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

int main() {
    int a = 5;
    a += 3; // a = 8
    cout << "a = " << a << endl;
    a *= 2; // a = 16
    cout << "a = " << a << endl;
    return 0;
}
```

Output:

```
a = 8
a = 16
```

---

4. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example
`&&`	Logical AND	`a > 5 && b < 10`
`		`
`!`	Logical NOT	`!(a > 5)`

Example:

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

int main() {
    int a = 6, b = 8;
    cout << (a > 5 && b < 10) << endl; // true => 1
    cout << (a > 7 || b < 10) << endl; // true => 1
    cout << !(a > 7) << endl;          // true => 1
    return 0;
}
```

Output:

```
1
1
1
```

---

5. Bitwise Operators

Works on bits and performs bit-by-bit operations.

Operator	Description	Example
`&`	Bitwise AND	`a & b`
`	`	Bitwise OR
`^`	Bitwise XOR	`a ^ b`
`~`	Bitwise NOT	`~a`

Operator	Description	Example
`<<`	Left Shift	`a << 1`
`>>`	Right Shift	`a >> 1`

Example:

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

int main() {
    int a = 5;   // 0101
    int b = 3;   // 0011

    cout << (a & b) << endl; // 1
    cout << (a | b) << endl; // 7
    cout << (a ^ b) << endl; // 6
    cout << (~a) << endl;    // -6 (2's complement)
    cout << (a << 1) << endl; // 10
    cout << (a >> 1) << endl; // 2
    return 0;
}
```

Output:

```
1
7
6
-6
10
2
```

C++: Control Flow Statements

If Statement

➤ Used to execute a block of code if a specified condition is true.

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

int main() {
    int x = 10;
    if (x > 5) {
        cout << "x is greater than 5" << endl;
    }
    return 0;
}
```

```
// Output: x is greater than 5
```

Else and Else If Statements

➢ `else` is executed when the condition in `if` is false.

➢ `else if` is used to check multiple conditions.

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

int main() {
    int num = 0;
    cin >> num;

    if (num > 0) {
        cout << "Positive number";
    } else if (num < 0) {
        cout << "Negative number";
    } else {
        cout << "Zero";
    }
    return 0;
}
```

```
// Output: Zero
```

Example:2

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

int main() {
    int marks;
    cout << "Enter marks: ";
    cin >> marks;

    if (marks >= 90) cout << "Grade A";
    else if (marks >= 75) cout << "Grade B";
    else cout << "Grade C";

    return 0;
}
```

Output (Example):

```
Enter marks: 80
Grade B
```

Nested If Statement

➢ Using if inside another if block.

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

int main() {
    int age = 20;

    if (age >= 18) {
        if (age < 60) {
            cout << "You are an adult";
        }
    }
    return 0;
}
```
```
// Output: You are an adult
```

While Loop

➢ Repeats a block of code as long as the condition is true.

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

int main() {
    int i = 1;
    while (i <= 5) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```
```
// Output: 1 2 3 4 5
```

For Loop

➢ Used to iterate a block of code a known number of times.

Sample Program

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

int main() {
    for (int i = 1; i <= 5; i++) {
        cout << "Count: " << i << endl;
    }
    return 0;
}
```

Output:

```
Count: 1
Count: 2
```

```
Count: 3
Count: 4
Count: 5
```

Loop Control Statements

➤ Used to control the execution flow of loops.

1. `break`: Exit the loop.

2. `continue`: Skip current iteration.

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

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3)
            continue;
        if (i == 5)
            break;
        cout << i << " ";
    }
    return 0;
}
// Output: 1 2 4
```

Input and Output

C++ uses `cin` for input and `cout` for output.

Sample Program

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

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    return 0;
}
```

Output (Example):

```
Enter a number: 5
You entered: 5
```

C++: Vectors and Tuples (STL Containers)

The Standard Template Library (STL) in C++ provides `vector` for dynamic arrays and `tuple` for grouping multiple values of different types.

Vectors

Vectors are dynamic arrays in C++ that can grow or shrink during execution.

Basic Declaration and Initialization

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

int main() {
    vector<int> nums = {10, 20, 30};
    cout << nums[0]; // Output: 10
    return 0;
}
```

Common Vector Methods

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

int main() {
    vector<int> v = {1, 2, 3};
    v.push_back(4);      // Add element to end
    v.pop_back();        // Remove last element
    v.insert(v.begin(), 0); // Insert at beginning
    v.erase(v.begin());     // Remove from beginning
    cout << v.size();    // Get size of vector
    return 0;
}
```

Indexing and Manual Slicing

C++ does not support slicing natively. But you can mimic it manually:

Example: Access and Manual Slice

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

int main() {
    vector<int> v = {10, 20, 30, 40, 50};
    cout << v[1]; // Output: 20 (Indexing)

    // Slicing manually (index 1 to 3)
    for (int i = 1; i < 4; i++) {
        cout << v[i] << " "; // Output: 20 30 40
    }
    return 0;
}
```

Loop-based Vector Population

Use loops to populate vectors dynamically:

Example: Square Each Element

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

int main() {
    vector<int> result;
    for (int i = 0; i < 5; i++) {
        result.push_back(i * i);
    }

    for (int num : result) {
        cout << num << " "; // Output: 0 1 4 9 16
    }
    return 0;
}
```

Tuples in C++

Tuples can store elements of different data types.

Tuple Declaration and Access

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

int main() {
    tuple<int, string, float> t1(1, "Hello", 3.14);
    cout << get<0>(t1) << endl;  // Output: 1
    cout << get<1>(t1) << endl;  // Output: Hello
    cout << get<2>(t1) << endl;  // Output: 3.14
    return 0;
}
```

Tuple Packing and Unpacking

```
#include <iostream>
#include <tuple>
```

```
using namespace std;

int main() {
    // Packing
    auto t = make_tuple(42, "C++", 2.5);

    // Unpacking
    int a;
    string b;
    float c;
    tie(a, b, c) = t;

    cout << a << " " << b << " " << c << endl; // Output: 42
C++ 2.5
    return 0;
}
```

Arrays

What is an Array?

An **array** is a **collection of elements** of the **same data type** stored in **contiguous memory locations**. It allows you to store multiple values under a single variable name.

Array Declaration

```
data_type array_name[array_size];
```

Example:

```
int marks[5]; // Declares an array of 5 integers
```

Array Initialization

You can assign values to arrays when you declare them.

Example:

```
int marks[5] = {80, 85, 90, 75, 95};
```

Or leave the size and let the compiler count:

```
int marks[] = {80, 85, 90}; // Automatically sets size to 3
```

Accessing Array Elements

Use **indexing** (starts from 0):

```
cout << marks[0];  // prints 80
```

Updating Array Values

```
cpp
CopyEdit
marks[1] = 88; // Updates second element
```

Input/Output with Arrays

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

int main() {
    int arr[5];

    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; i++) {
        cin >> arr[i];
    }

    cout << "You entered: ";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Array Size

Use `sizeof()` to find the number of elements:

```
int size = sizeof(arr) / sizeof(arr[0]);
```

Array Memory Layout

All array elements are stored **next to each other in memory**:

```
Index:    0     1     2     3     4
Value:   10    20    30    40    50
```

Multi-dimensional Arrays

Used for matrices or tables (e.g., 2D arrays).

Declaration:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Access:

```
cout << matrix[1][2];  // prints 6
```

Nested Loop:

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}
```

Character Arrays (Strings)

```
char name[6] = {'M', 'a', 'z', 'i', 'd', '\0'};
cout << name; // prints Mazid
```

Or directly:

```
char name[] = "Mazid";
```

Limitations of Arrays

- Fixed size
- Cannot change size at runtime
- No built-in bounds checking

Modern Alternative: `vector` from STL

```
#include <vector>
vector<int> v = {1, 2, 3, 4, 5};
v.push_back(6);
```

You can use vectors when you need dynamic sizing.

Sample Program

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

int main() {
    int numbers[3] = {10, 20, 30};
    for (int i = 0; i < 3; i++) {
        cout << "Element " << i << ": " << numbers[i] <<
endl;
    }
    return 0;
}
```

Output:

```
Element 0: 10
Element 1: 20
Element 2: 30
```

Functions

Functions are reusable blocks of code.

Sample Program

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

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

int main() {
    cout << "Sum: " << add(3, 4) << endl;
    return 0;
}
```

Output:

```
Sum: 7
```

Pointers

Pointers store memory addresses.

Sample Program

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

int main() {
    int x = 10;
    int* p = &x;
    cout << "Value: " << *p << endl;
    cout << "Address: " << p << endl;
    return 0;
}
```

Output:

```
Value: 10
Address: 0x7ffeece97b5c  // (may vary)
```

Object-Oriented Programming

OOP is a programming paradigm based on **objects and classes**. It helps write **modular, reusable, and maintainable code**.

---

4 Pillars of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

---

1. Class & Object

Class: A blueprint for creating objects.

Object: An instance of a class.

Example:

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

class Car {
public:
    string brand;
    int speed;

    void drive() {
        cout << brand << " is driving at " << speed << "
km/h." << endl;
    }
};

int main() {
    Car myCar;
    myCar.brand = "BMW";
    myCar.speed = 120;
    myCar.drive();

    return 0;
}
```

---

2. Encapsulation – Binding data and methods in one unit (class)

Protects data from direct access

```
class Student {
private:
    int age;

public:
    void setAge(int a) {
        if (a > 0)
            age = a;
    }

    int getAge() {
        return age;
    }
};
```

`private` data is accessed using `public` methods only.

---

3. Abstraction – Showing essential features, hiding internal details

User sees what, not how

```
class ATM {
public:
    void withdraw() {
        cout << "Amount withdrawn" << endl;
    }

    void checkBalance() {
        cout << "Balance is ₹10,000" << endl;
    }
};
```

We don't show internal processing like PIN verification, database fetch, etc.

---

4. Inheritance – Acquiring properties of another class

`base class ⟶ derived class`

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

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

int main() {
    Dog d;
    d.sound();  // from base class
    d.bark();   // from derived class
}
```

Types of Inheritance:

- Single
- Multilevel
- Multiple
- Hierarchical
- Hybrid

Single Inheritance

```
class Animal {
public:
    void sound() {
        cout << "Animal Sound\n";
    }
};

class Dog : public Animal {
};

int main() {
    Dog d;
    d.sound();  // Inherited method
    return 0;
}
```

Multilevel Inheritance

```
class LivingThing {
public:
    void alive() {
        cout << "I am alive\n";
    }
};

class Animal : public LivingThing {
};

class Dog : public Animal {
};

int main() {
    Dog d;
    d.alive();  // Accessing grandparent method
    return 0;
}
```

Multiple Inheritance

```
class A {
public:
    void methodA() {
        cout << "From A\n";
    }
};

class B {
public:
    void methodB() {
        cout << "From B\n";
    }
};

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

```
int main() {
    C obj;
    obj.methodA();
    obj.methodB();
    return 0;
}
```

Hierarchical Inheritance

```
class Parent {
public:
    void greet() {
        cout << "Hello from Parent\n";
    }
};

class Child1 : public Parent {
};

class Child2 : public Parent {
};

int main() {
    Child1 c1;
    Child2 c2;
    c1.greet();
    c2.greet();
    return 0;
}
```

Hybrid Inheritance (Combination)

```
class A {
public:
    void showA() {
        cout << "Class A\n";
    }
};

class B : public A {
public:
    void showB() {
        cout << "Class B\n";
    }
};

class C {
public:
    void showC() {
        cout << "Class C\n";
    }
};

class D : public B, public C {
public:
    void showD() {
        cout << "Class D\n";
    }
};
```

```
int main() {
    D obj;
    obj.showA();
    obj.showB();
    obj.showC();
    obj.showD();
    return 0;
}
```

5. Polymorphism – Many forms (same name, different behavior)

Two Types:

Compile-Time (Function Overloading)

```
class Math {
public:
    int add(int a, int b) {
        return a + b;
    }

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

Runtime (Function Overriding using Virtual Functions)

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

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

int main() {
    Animal* a;
    Cat c;
    a = &c;
    a->sound();  // Calls Cat's sound
}
```

Access Specifiers in C++

Specifier	Description
`public`	Accessible from anywhere
`private`	Accessible only within the class
`protected`	Accessible in class and derived classes

File Handling

Read/write files using `ifstream` and `ofstream`.

Sample Program

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

int main() {
    ofstream fout("sample.txt");
    fout << "Hello File";
    fout.close();

    string text;
    ifstream fin("sample.txt");
    getline(fin, text);
    cout << text << endl;
    fin.close();
    return 0;
}
```

Output:

```
Hello File
```

Exception Handling in C++

What is Exception Handling?

Exception handling in C++ is a mechanism used to **detect and manage runtime errors**, allowing a program to continue running or fail gracefully instead of crashing unexpectedly.

It separates **error-handling code** from the **normal logic**, making programs more robust and readable.

Keywords Used:

- `try`: Defines a block of code in which an exception might occur.
- `throw`: Used to **raise an exception** when a problem is detected.
- `catch`: Defines a block of code that handles the exception.

Basic Flow:

1. Code that might cause an exception is placed inside a `try` block.
2. If an error occurs, the `throw` statement sends the exception to the `catch` block.
3. The `catch` block then handles the error.

Sample Program

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

int main() {
    try {
        int x = 5;
        if (x < 10) throw "Value too small";
    } catch (const char* msg) {
        cout << "Error: " << msg << endl;
    }
    return 0;
}
```

Output:

```
Error: Value too small
```

Explanation:

- `try` block checks if `x < 10`. Since `x` is `5`, it is **less than 10**.
- So, `throw "Value too small"` is executed.
- This exception is **caught** in the `catch` block, and the error message is printed.

Other Exception Types:

You can throw different types of exceptions like:

- `int` – throw 404;
- `float` – throw 3.14;
- `string` – throw string("Custom error");
- Custom exception classes

Multiple Catch Blocks:

You can also have multiple `catch` blocks for handling different types:

```
try {
    throw 10;
} catch (int e) {
    cout << "Integer Exception: " << e << endl;
} catch (...) {
    cout << "Default handler for unknown exceptions" << endl;
}
```

STL – Standard Template Library

Includes vectors, lists, stacks, queues, maps.

Sample Program

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

int main() {
    vector<int> v = {1, 2, 3};
    v.push_back(4);
    for (int x : v)
        cout << x << " ";
    return 0;
}
```

Output:

```
1 2 3 4
```

Templates

Allows generic functions/classes.

Sample Program

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

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

int main() {
    cout << add(2, 3) << endl;
    cout << add(2.5, 3.1) << endl;
    return 0;
}
```

Output:

```
5
5.6
```

Lambda Functions

Anonymous inline functions.

Sample Program

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

int main() {
    auto square = [](int x) { return x * x; };
    cout << square(4) << endl;
    return 0;
}
```

Output:

```
16
```
