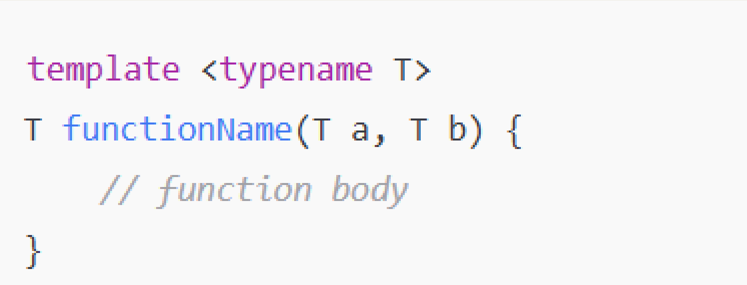
CHAPTER-7

Function Template and Exception handling

What is function template?

- A function template in C++ is a blueprint for creating functions that work with any data type.
- Instead of writing multiple versions of the same function for different data types (like int, float, double, etc.), you write one generic function using a template, and the compiler generates the specific version when needed.
- Template is simple and yet very powerful tools in c++. The simple idea is to pass data type as parameter so that we don’t need to write the same code for different data types.
- Syntax



```
template <typename T>
T functionName(T a, T b) {
    // function body
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Importance of template

Eliminates Code Duplication

– You don’t need to write separate functions or classes for each data type.

Improves Code Reusability

– The same template works for int, float, char, string, etc.

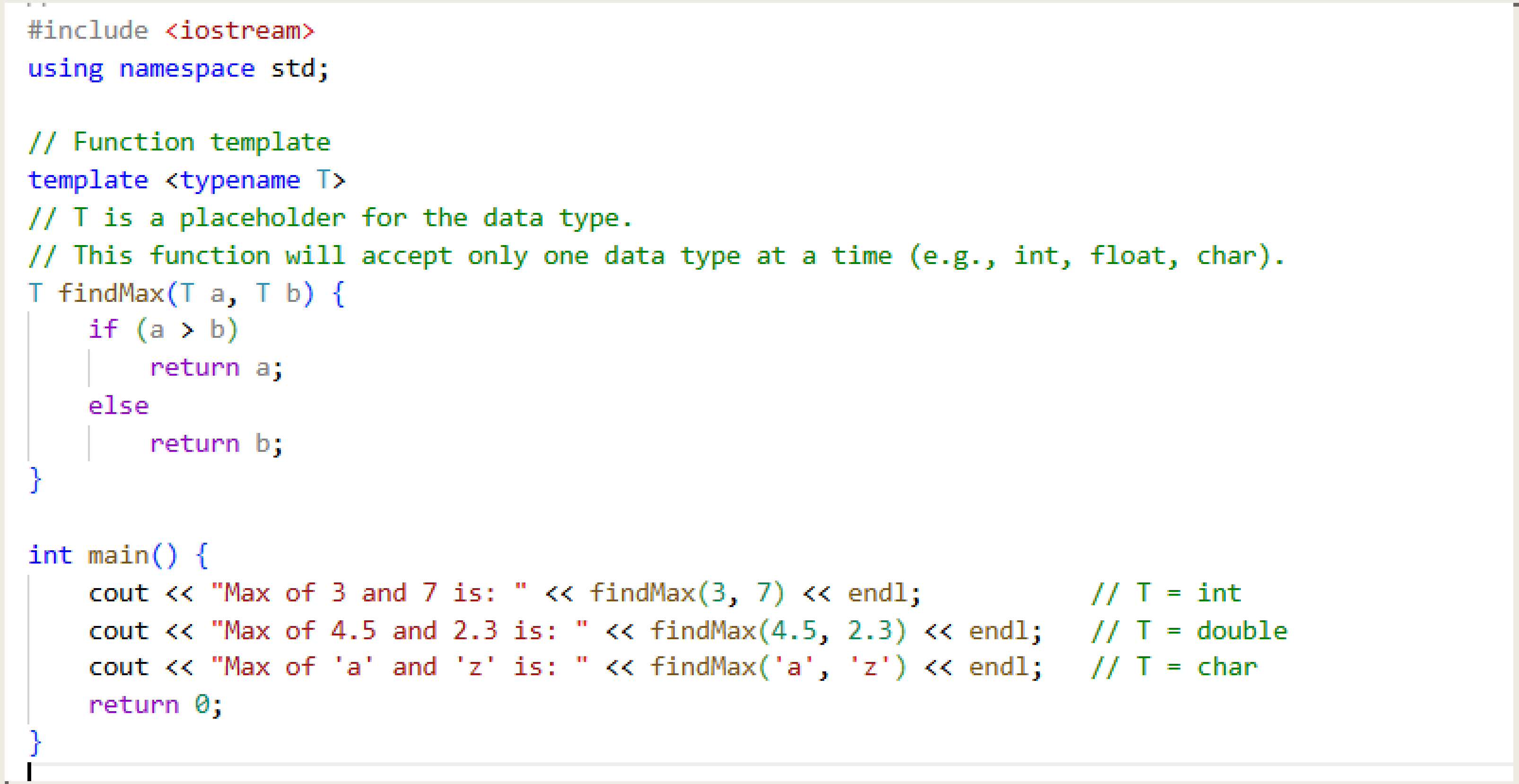
Makes Development Easier

– Less code to write and maintain. Logic is written once.

Enhances Maintainability

– If logic changes, you only need to update the template, not multiple versions.

Example of template



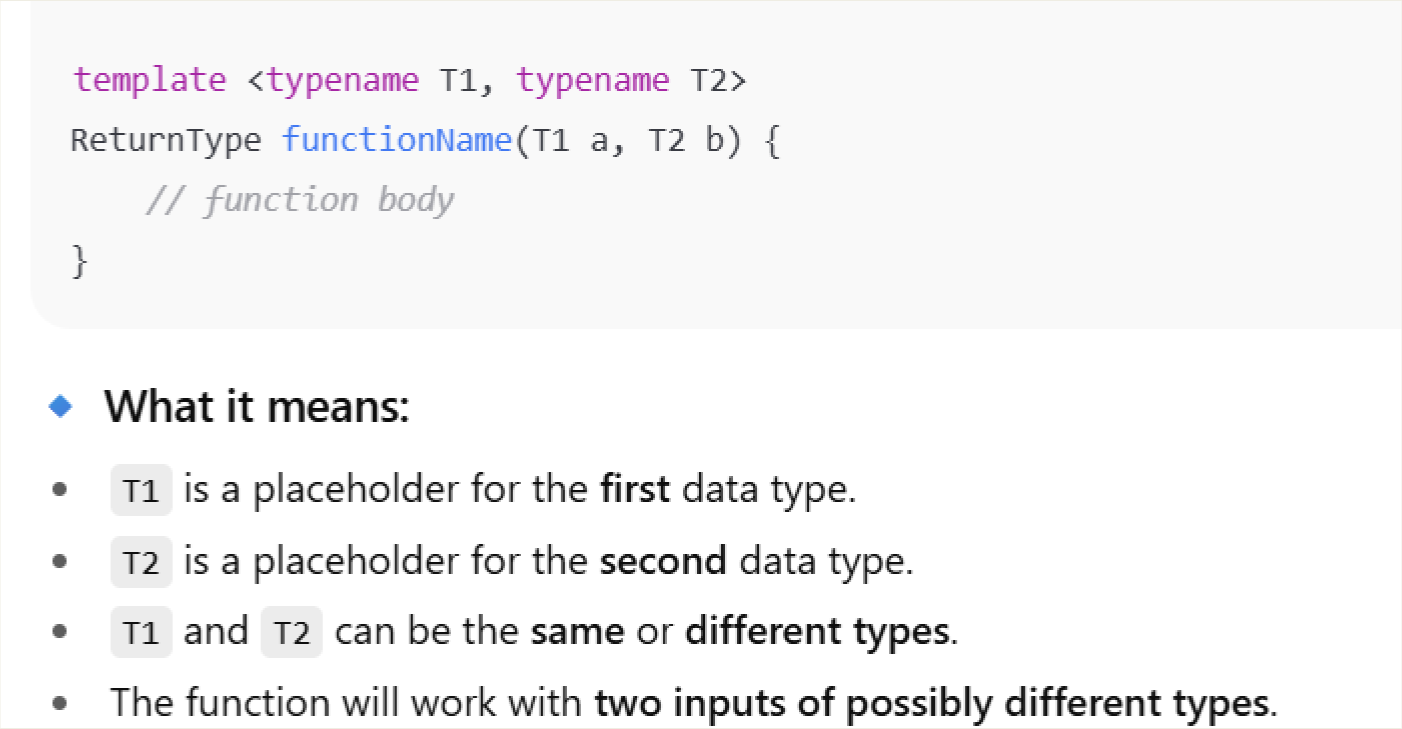
```cpp
#include <iostream>
using namespace std;

// Function template
template <typename T>
// T is a placeholder for the data type.
// This function will accept only one data type at a time (e.g., int, float, char).
T findMax(T a, T b) {
    if (a > b)
        return a;
    else
        return b;
}

int main() {
    cout << "Max of 3 and 7 is: " << findMax(3, 7) << endl;           // T = int
    cout << "Max of 4.5 and 2.3 is: " << findMax(4.5, 2.3) << endl;   // T = double
    cout << "Max of 'a' and 'z' is: " << findMax('a', 'z') << endl;   // T = char
    return 0;
}
```

Assignment

Try example of function template by taking input form user

Function Template with multiple parameters

Syntax:



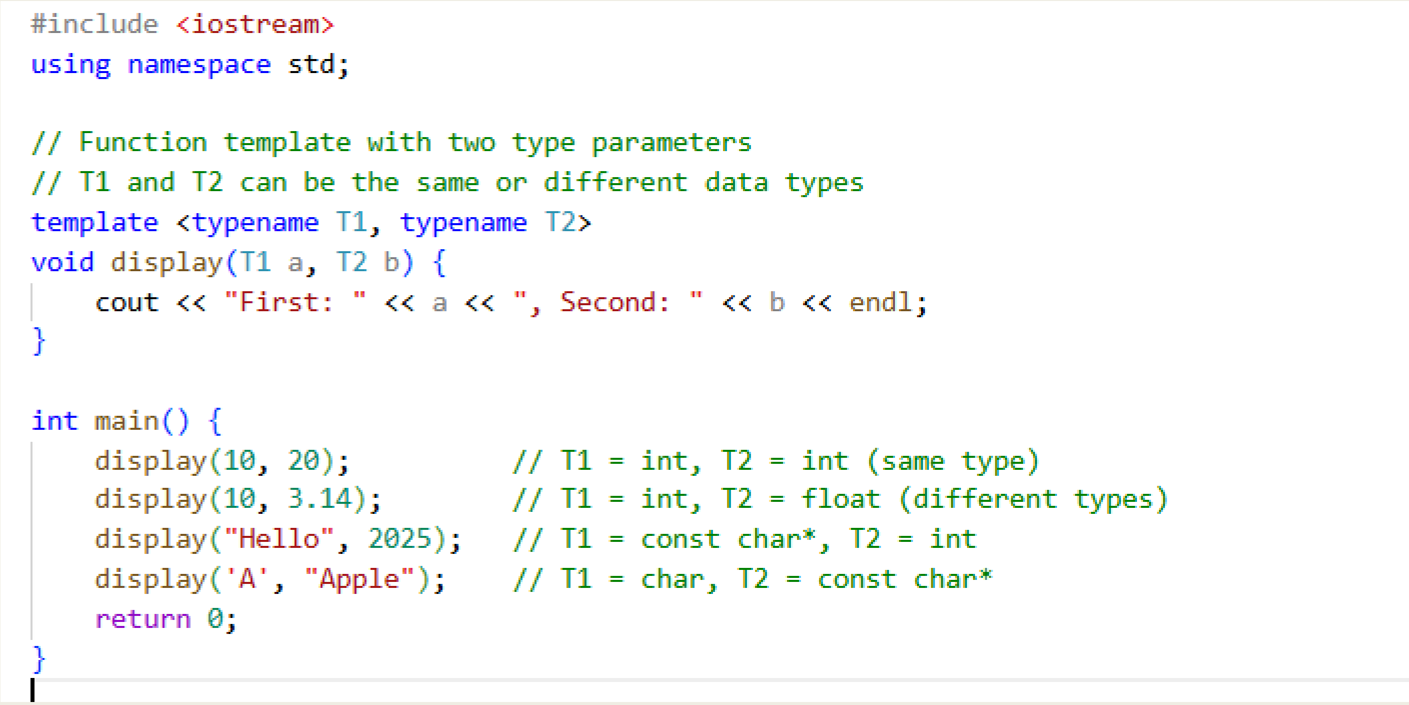
```
template <typename T1, typename T2>
ReturnType functionName(T1 a, T2 b) {
    // function body
}
```

◆ **What it means:**

- `T1` is a placeholder for the **first** data type.
- `T2` is a placeholder for the **second** data type.
- `T1` and `T2` can be the **same** or **different types**.
- The function will work with **two inputs of possibly different types**.

Function Template with multiple parameters

Syntax:

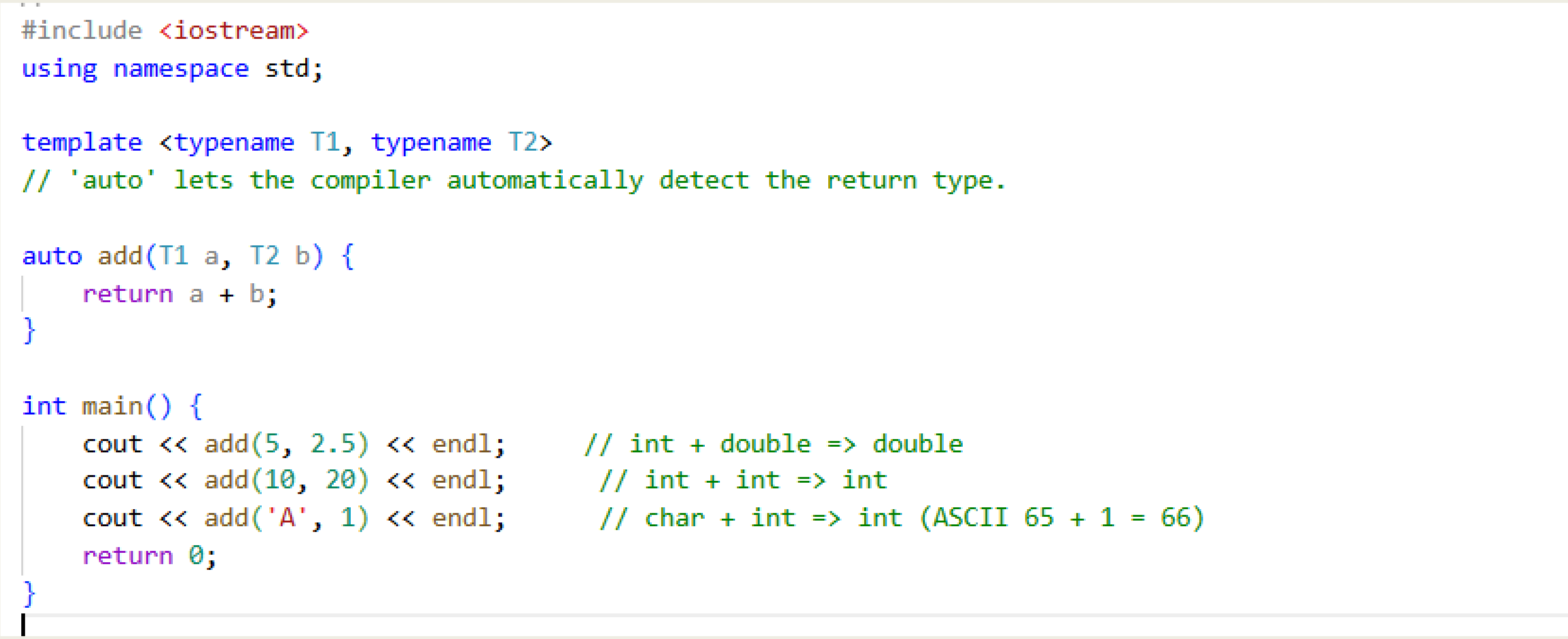


```
#include <iostream>
using namespace std;

// Function template with two type parameters
// T1 and T2 can be the same or different data types
template <typename T1, typename T2>
void display(T1 a, T2 b) {
    cout << "First: " << a << ", Second: " << b << endl;
}

int main() {
    display(10, 20);          // T1 = int, T2 = int (same type)
    display(10, 3.14);        // T1 = int, T2 = float (different types)
    display("Hello", 2025);   // T1 = const char*, T2 = int
    display('A', "Apple");    // T1 = char, T2 = const char*
    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Function Template with multiple parameters and return types (Another Example)

Note: In exam you should not have to specify the return type not retrun type just write one example as per you connivence



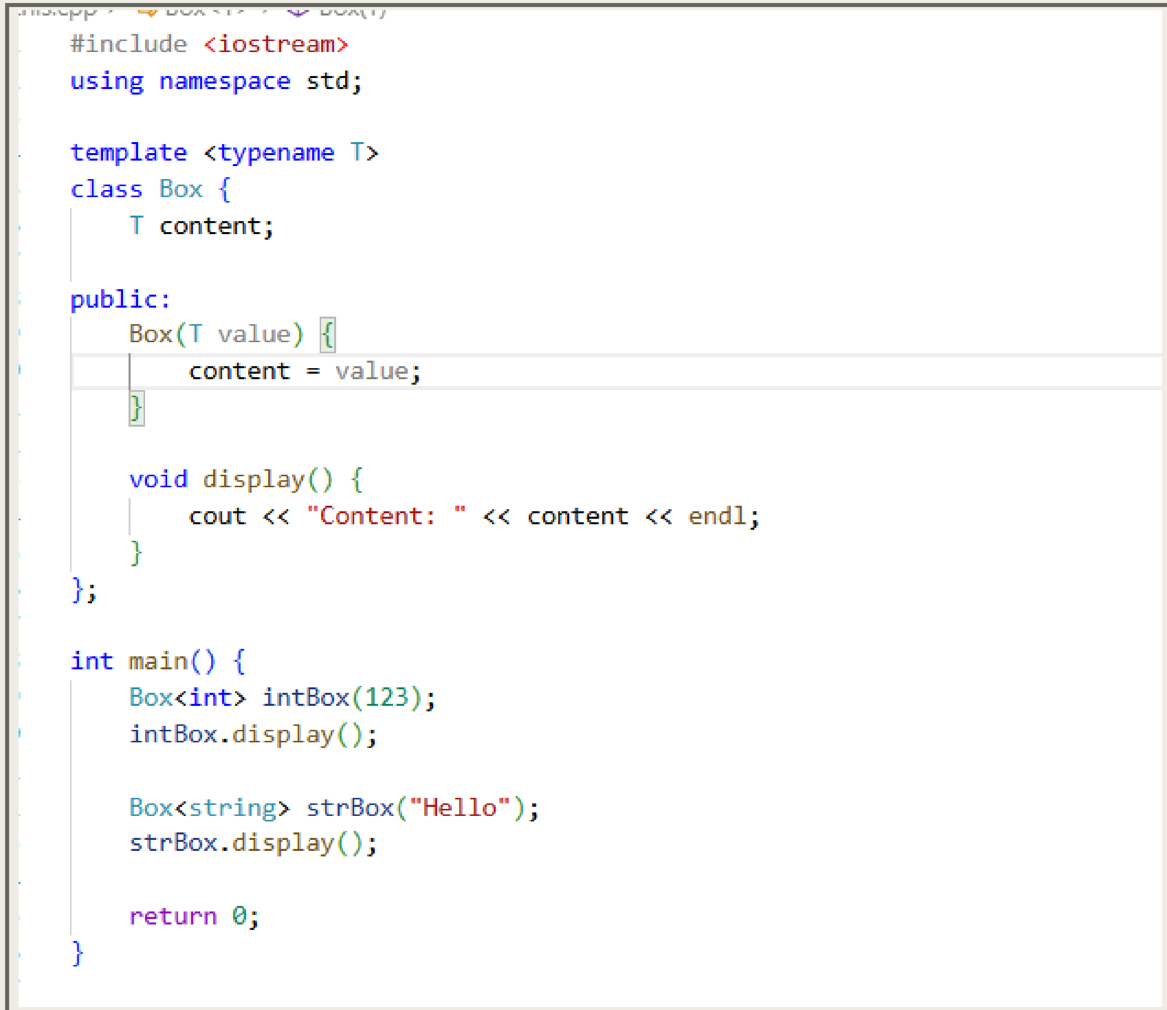
```cpp
#include <iostream>
using namespace std;

template <typename T1, typename T2>
// 'auto' lets the compiler automatically detect the return type.

auto add(T1 a, T2 b) {
    return a + b;
}

int main() {
    cout << add(5, 2.5) << endl;     // int + double => double
    cout << add(10, 20) << endl;      // int + int => int
    cout << add('A', 1) << endl;      // char + int => int (ASCII 65 + 1 = 66)
    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Class Template

- A class template is like a blueprint for creating classes that can work with any data type. Instead of writing the same class multiple times for different data types (like int, float, char), you write a single class template that can be used for all types.

Example

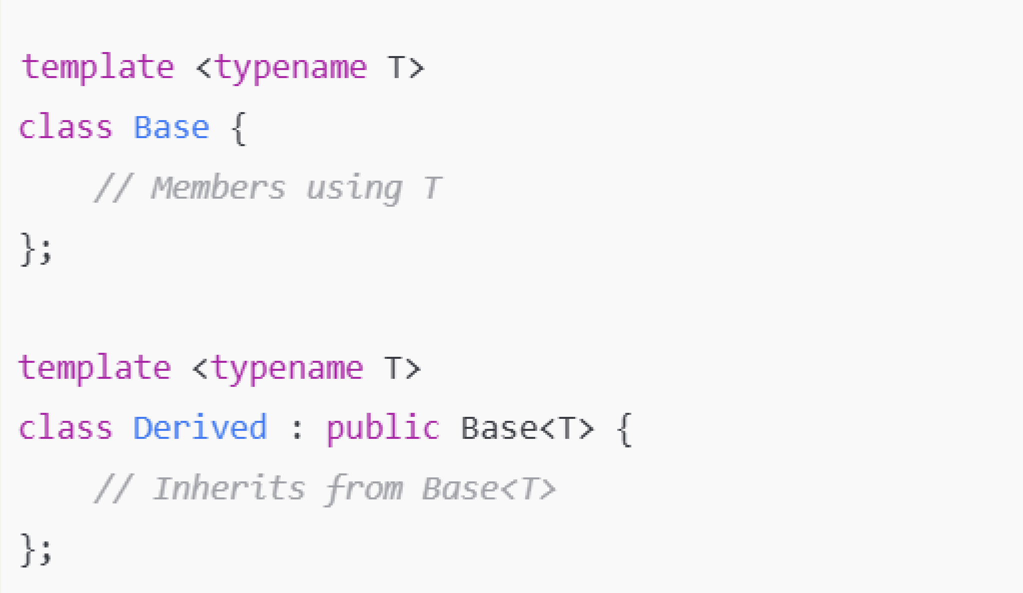


```
#include <iostream>
using namespace std;

template <typename T>
class Box {
    T content;

public:
    Box(T value) {
        content = value;
    }

    void display() {
        cout << "Content: " << content << endl;
    }
};

int main() {
    Box<int> intBox(123);
    intBox.display();

    Box<string> strBox("Hello");
    strBox.display();

    return 0;
}
```

Inheriting from a template class:

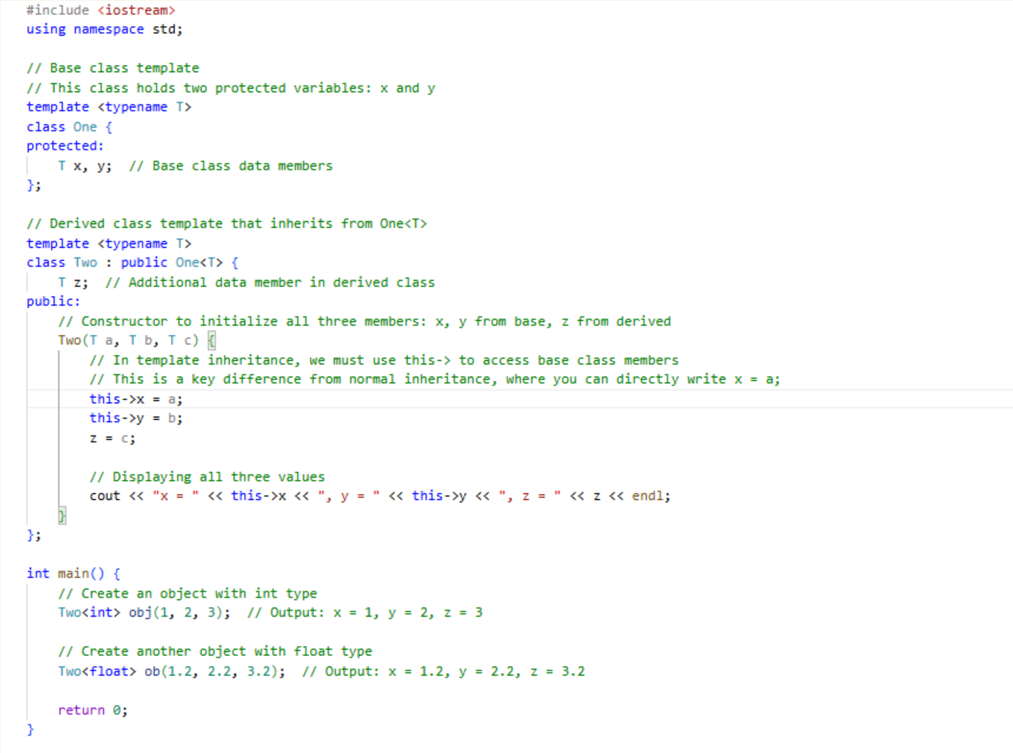
- **Template inheritance** in C++ means a **template class (child)** inherits from another **template class (parent).**
- This allows you to write generic, reusable code where both base and derived classes work with **generic (template) types.**
- The main purpose of template inheritance is to **extend the functionality** of a template base class in a type-independent way, by adding new members or modifying behavior in the derived class.

Syntax of Template inheritance



```
template <typename T>
class Base {
    // Members using T
};

template <typename T>
class Derived : public Base<T> {
    // Inherits from Base<T>
};
```

Inheriting from a template class:



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

// Base class template
// This class holds two protected variables: x and y
template <typename T>
class One {
protected:
    T x, y;  // Base class data members
};

// Derived class template that inherits from One<T>
template <typename T>
class Two : public One<T> {
    T z;  // Additional data member in derived class
public:
    // Constructor to initialize all three members: x, y from base, z from derived
    Two(T a, T b, T c) {
        // In template inheritance, we must use this-> to access base class members
        // This is a key difference from normal inheritance, where you can directly write x = a;
        this->x = a;
        this->y = b;
        z = c;

        // Displaying all three values
        cout << "x = " << this->x << ", y = " << this->y << ", z = " << z << endl;
    }
};

int main() {
    // Create an object with int type
    Two<int> obj(1, 2, 3);  // Output: x = 1, y = 2, z = 3

    // Create another object with float type
    Two<float> ob(1.2, 2.2, 3.2);  // Output: x = 1.2, y = 2.2, z = 3.2

    return 0;
}
```

Exception Handling0

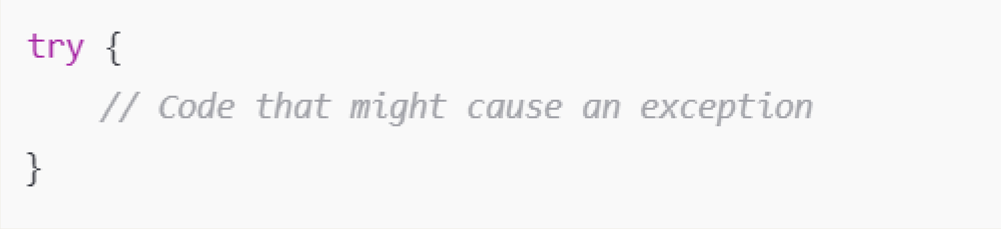
- **Exception handling** is a mechanism in C++ that handles runtime errors (also called **exceptions**) so that the program doesn't crash abruptly. It allows the program to **detect**, **throw**, and **catch** errors gracefully.
- **Why Use It?**
- To handle unexpected errors (e.g., divide by zero, file not found, memory allocation failure).
- To prevent the program from terminating abnormally.
- To separate **error-handling code** from **regular logic.**

Exception Handling0

C++ uses **three main keywords** to implement exception handling:

try

- Used to define a block of code that may throw an exception.
- It "tries" to execute the code inside it.



```
try {
    // Code that might cause an exception
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Exception Handling (contd...

Throw keyword

In C++, the throw keyword is used to raise an exception when an error occurs. This exception is then caught and handled by a matching catch block, which prevents the program from crashing.

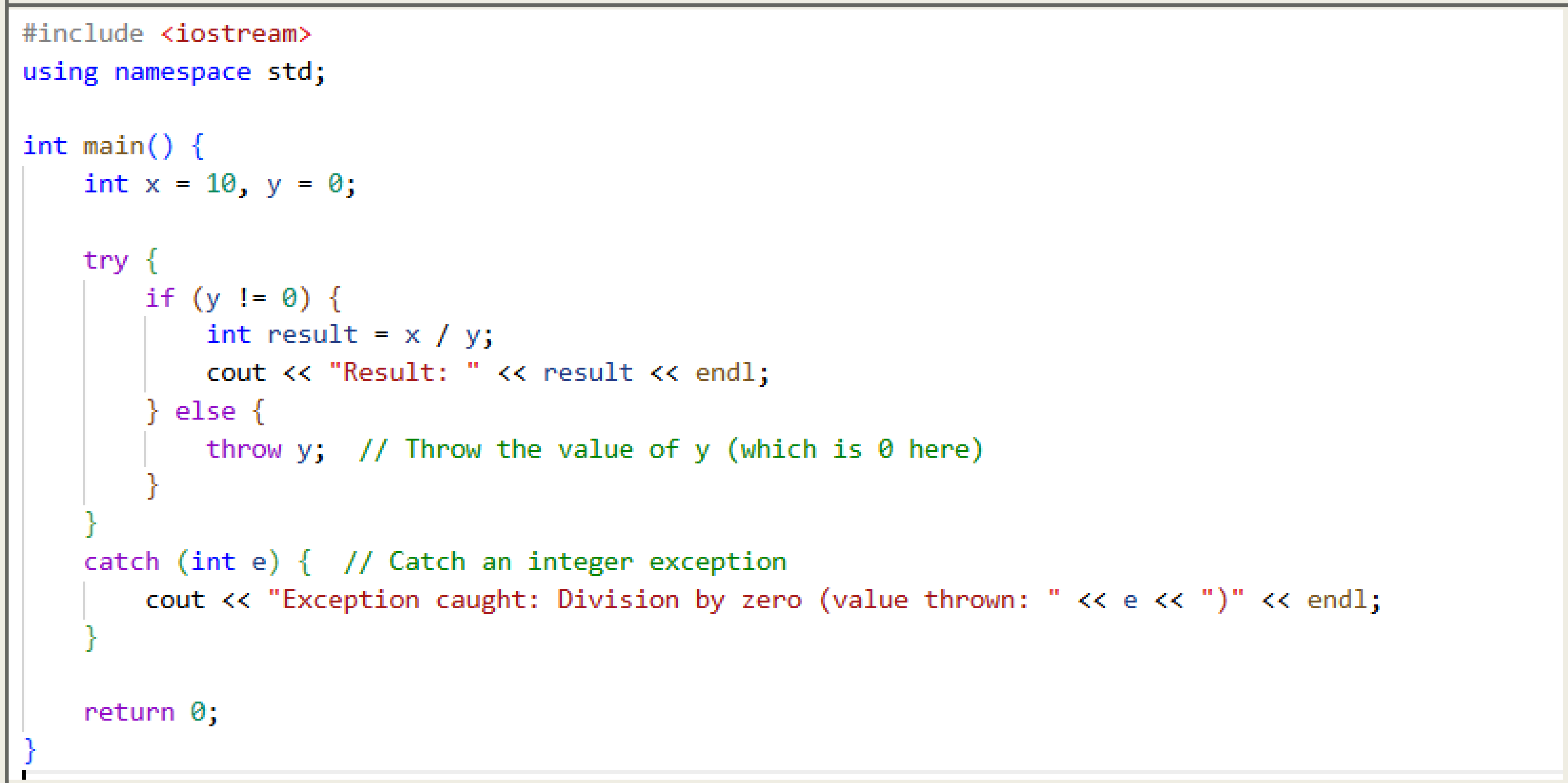
Prepared by Ayush Lamsal | |
layush788@gmail.com

Exception Handling0

Catch Keyword

- The catch block catches the exception thrown from the try block and contains the code to handle the error.
- The catch keyword is used to handle exceptions thrown by the throw statement inside a try block.
- It receives the exception and executes code to manage the error, preventing the program from crashing.
- A catch block is always written immediately after a try block to specify how to respond when an exception occurs.
- Multiple catch blocks can be used to handle different types of exceptions

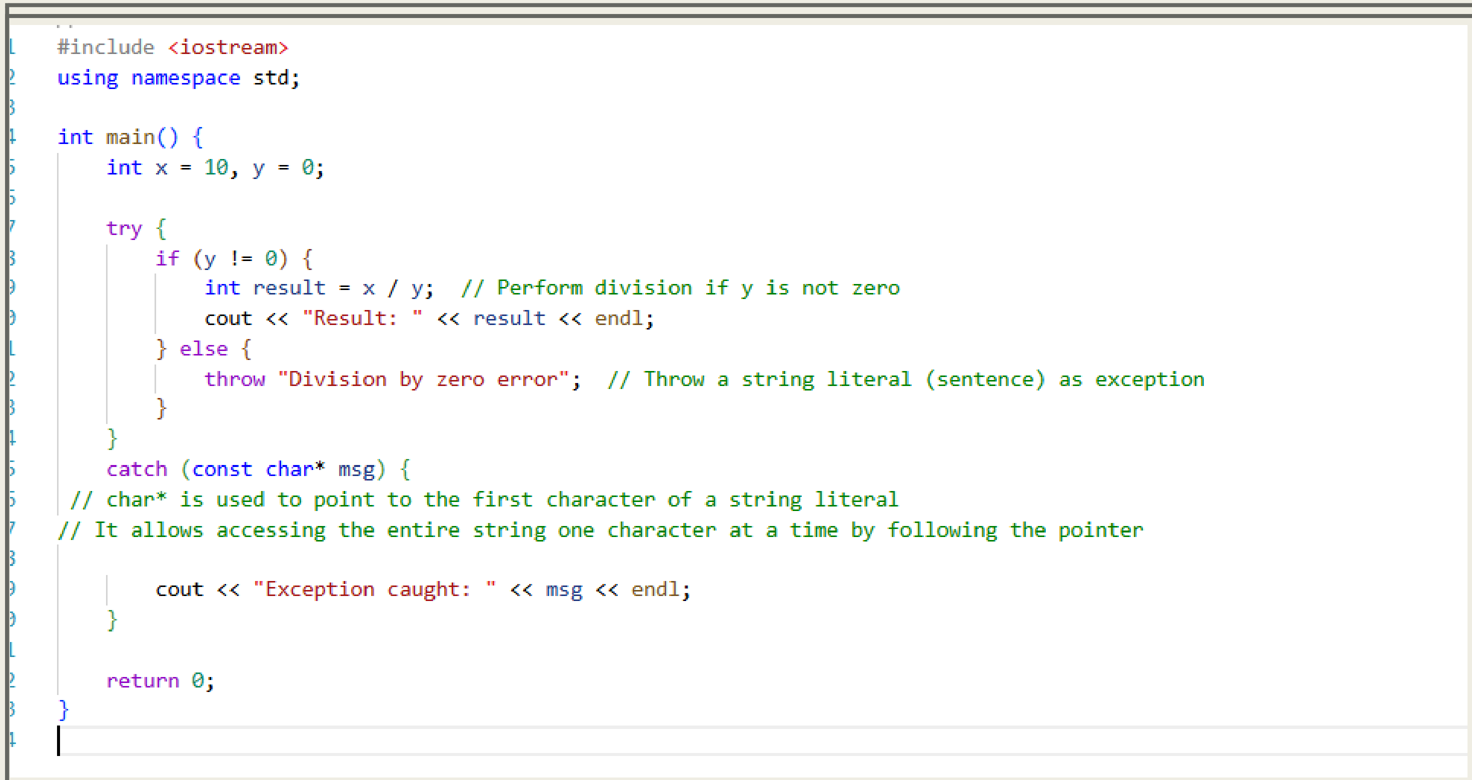
Exception Handling (Simple example)
Divide by zero exception



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

int main() {
    int x = 10, y = 0;

    try {
        if (y != 0) {
            int result = x / y;
            cout << "Result: " << result << endl;
        } else {
            throw y;  // Throw the value of y (which is 0 here)
        }
    }
    catch (int e) {  // Catch an integer exception
        cout << "Exception caught: Division by zero (value thrown: " << e << ")" << endl;
    }

    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

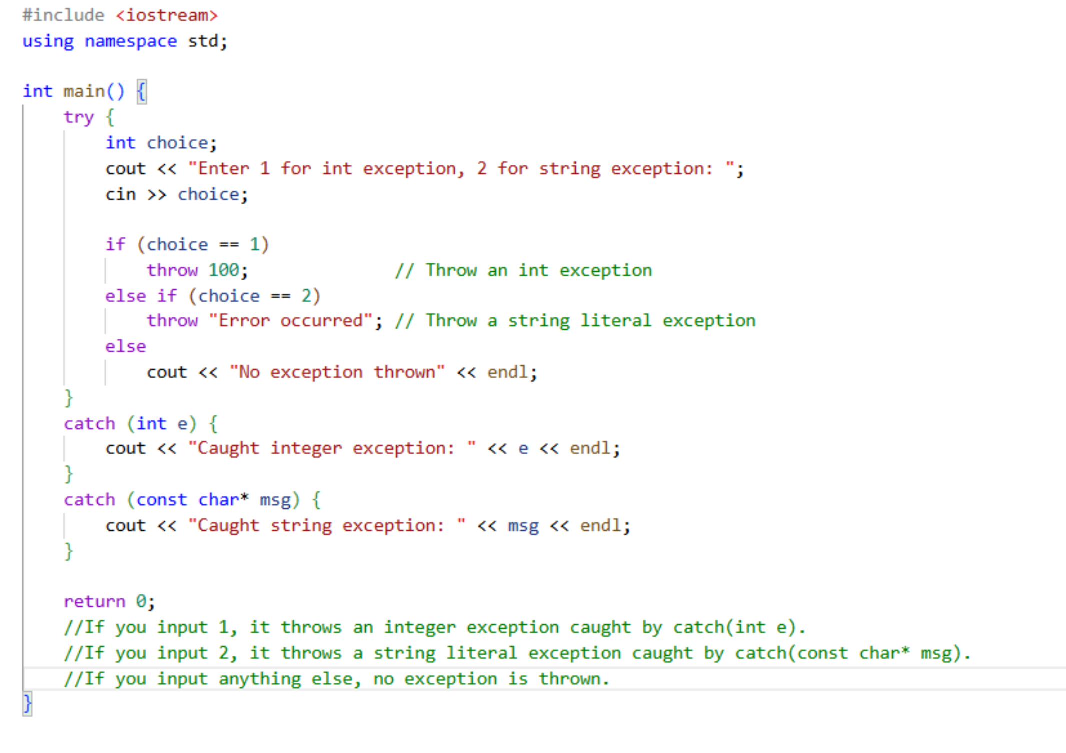
Exception Handling (Simple example)
Divide by zero exception



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

int main() {
    int x = 10, y = 0;

    try {
        if (y != 0) {
            int result = x / y;  // Perform division if y is not zero
            cout << "Result: " << result << endl;
        } else {
            throw "Division by zero error";  // Throw a string literal (sentence) as exception
        }
    }
    catch (const char* msg) {
 // char* is used to point to the first character of a string literal
// It allows accessing the entire string one character at a time by following the pointer

        cout << "Exception caught: " << msg << endl;
    }

    return 0;
}
```

Multiple exception (Handling)
Multiple catch block

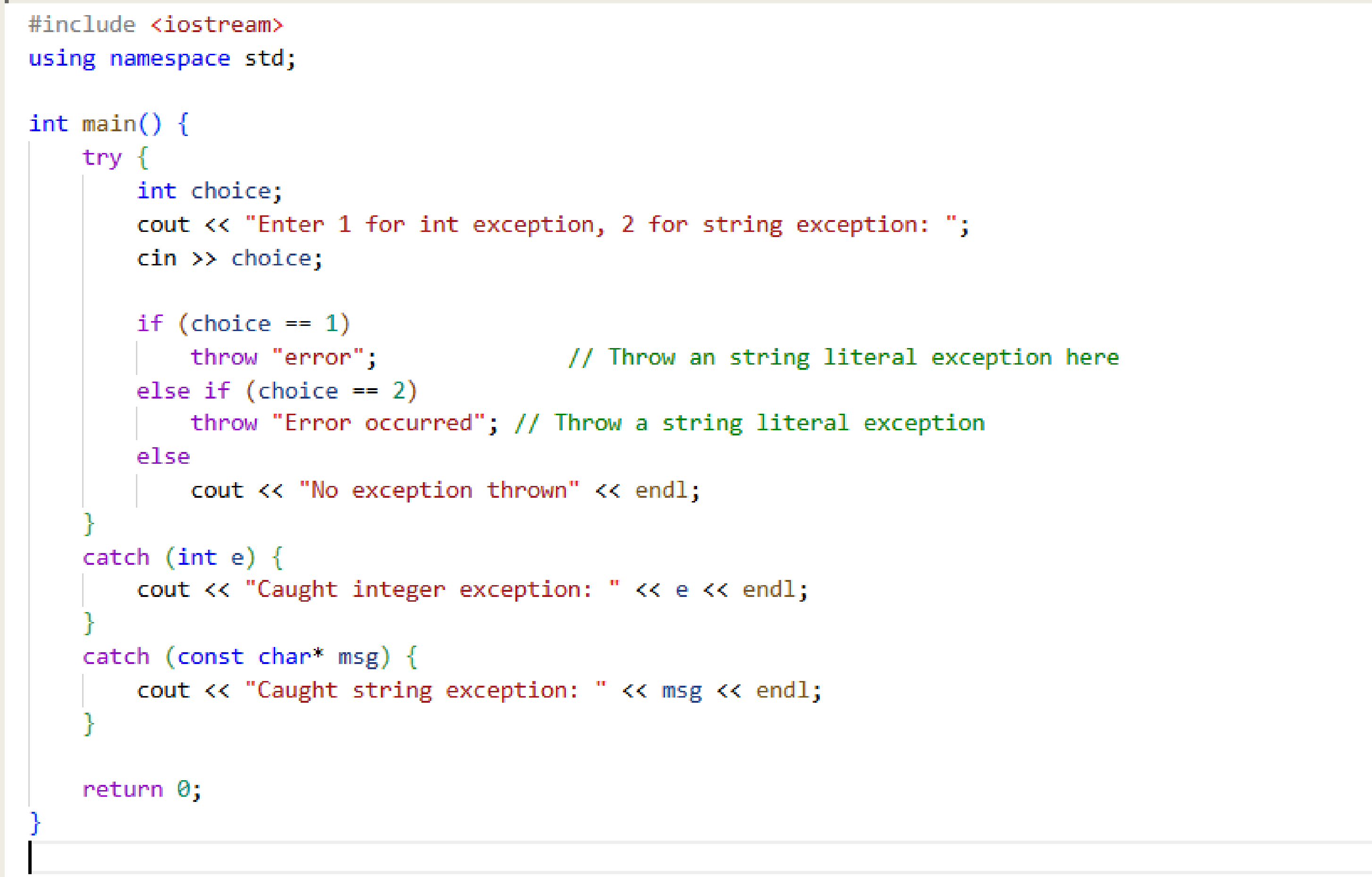
- A single try statement can have a multiple catch statement which is called multiple exception



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

int main() {
    try {
        int choice;
        cout << "Enter 1 for int exception, 2 for string exception: ";
        cin >> choice;

        if (choice == 1)
            throw 100;              // Throw an int exception
        else if (choice == 2)
            throw "Error occurred"; // Throw a string literal exception
        else
            cout << "No exception thrown" << endl;
    }
    catch (int e) {
        cout << "Caught integer exception: " << e << endl;
    }
    catch (const char* msg) {
        cout << "Caught string exception: " << msg << endl;
    }

    return 0;
    //If you input 1, it throws an integer exception caught by catch(int e).
    //If you input 2, it throws a string literal exception caught by catch(const char* msg).
    //If you input anything else, no exception is thrown.
}
```

Prepared by Ayush Lamsal ||
layush788@gmail.com

What will be output?



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

int main() {
    try {
        int choice;
        cout << "Enter 1 for int exception, 2 for string exception: ";
        cin >> choice;

        if (choice == 1)
            throw "error";              // Throw an string literal exception here
        else if (choice == 2)
            throw "Error occurred"; // Throw a string literal exception
        else
            cout << "No exception thrown" << endl;
    }
    catch (int e) {
        cout << "Caught integer exception: " << e << endl;
    }
    catch (const char* msg) {
        cout << "Caught string exception: " << msg << endl;
    }

    return 0;
}
```

Use of Exception Handling

- Exception handling helps build reliable and strong systems.
- It manages unexpected or undesired system behaviors.
- Separates error-handling code from normal program logic.
- Prevents showing technical error messages to users.Displays friendly and clear error messages instead.
- Allows the program to handle exceptions smoothly.
- Keeps the system stable and running well.

The End

Prepared by Ayush Lamsal ||
layush788@gmail.com