Object Oriented Programming
CL-1004

Lab 13
Generics & Exception Handling

National University of Computer & Emerging Sciences – NUCES – Karachi

National University of Computer & Emerging Sciences – NUCES – Karachi

Course Code: CL-1004	Course: Object Oriented Programming - OOP

Contents

Generics / Template

Generics, in the context of programming, denotes the capacity to craft code adaptable to various data types. In C++, this adaptability is facilitated through templates, which serve as customizable code structures accommodating one or multiple types.

In C++, the terms **generics** and **templates** are frequently used interchangeably due to templates enabling the creation of versatile code. By utilizing templates, developers can write code capable of processing diverse data types, eliminating the need for separate implementations tailored to each type.

1. Template Function

A template function is a function that is defined using a template parameter, which can represent any data type. The template parameter is specified using the `typename` keyword, followed by a name that is used within the function definition to represent the actual data type. Here's the basic syntax for defining a template function:

```
template <typename T>
void functionName(T parameter) {
// function body
}
```

In this example, **functionName** is the name of the function, and **T** is the template parameter. The **typename** keyword specifies that T represents a data type, and the function definition can use **T** to declare variables, parameters, and return values of that data type.

To call a template function, you specify the actual data type as the template argument when you call the function.

For example:

```
// Implicit template argument specification
int intValue = 42;
int doubleValue = 3.14;

functionName(intValue);
functionName(doubleValue);

// Explicit template argument specification
functionName<int>(42);   // calls functionName with an int parameter
functionName<double>(3.14); // calls functionName with a double parameter
```

In this example, **int** and **double** are the template arguments that specify the actual data types for the T template parameter.

Template functions are useful when you need to write a function that can work with different data types, without having to write separate implementations for each data type. Templates allow you to write generic code that can be reused with different data types, which can save time and reduce code duplication.

Example 01: Template functions to print values of any data type

```
#include "iostream"
using namespace std;

template <typename T, typename U>
void printValue(T value1, U value2) {
    cout << "Addition of integer and double: " << value1 + value2 << endl;
}

template <typename T>
void printValue(T value1) {
    cout << "Value: " << value1 << endl;
}


int main() {
    int intValue = 42;
    double doubleValue = 3.14;
    char charValue = 'A';

    printValue(intValue, doubleValue);
    printValue(doubleValue);
```

```
    printValue(charValue);

    return 0;
}
```

Output	Addition of integer and double: 45.14 Value: 3.14 Value: A

Explanation:

In the above code, two generic functions with the same name **printValue** are defined using templates.

For the first **printValue** function, the **typename T** and **typename U** declares two type parameters **T** and **U**, which can be any data type. The function takes two parameters of type **T** and **U** and prints the sum of an **integer** and a **double** value.

Similarly, for the second generic function **printValue**, **typename T** declares a type parameter **T** which is again can be any data type. This function only takes one parameter of type **T** and prints its value to the console.

In the **main** function, three variable **intValue**, **doubleValue** and **charValue** are initialized.

> The **printValue(T value1, U value2)** function with two arguments **value1** and **value2** is called first.
>
> Then the other **printValue(T value1)** function with single argument is called two times, one with a value of type **double** and second with a value of type **char**.

Since, **printValue** is a template function, it can be used with any data type.

Example 02: A template function in C++ that finds the maximum element in an array of any data type

```
#include "iostream"
using namespace std;

template <typename T>
T maxElement(T* array, int size) {
    T max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

```
int main() {
    int intArray[] = { 4, 2, 5, 1, 3 };
    double doubleArray[] = { 3.14, 1.5, 2.7, 4.2 };

    int maxInt = maxElement(intArray, 5);
    double maxDouble = maxElement(doubleArray, 4);

    cout << "Max element in intArray: " << maxInt << endl;
    cout << "Max element in doubleArray: " << maxDouble << endl;

    return 0;
}
```

Output	Max element in intArray: 5 Max element in doubleArray: 4.2

Explanation: In this example, the **maxElement** template function takes an array of any data type and its size as input, and returns the maximum element in the array. The function uses a simple loop algorithm to traverse the array and find the maximum element. The **main** program calls the **maxElement** function with an array of integers and an array of doubles, and prints the maximum element for each array. This function can be used with any data type that supports the greater-than operator (**>**).

2. Template Class

In C++, a template class is a class that is defined with generic type parameters. These parameters can be used to define member functions, member variables, and the overall structure of the class.

Template classes are useful because they allow you to write code that is independent of the specific data types it operates on. For example, you could write a generic container class that can hold any type of object. This container class could then be used with any data type without needing to be re-implemented for each specific type.

Suppose you are writing software for a weather station that can measure different types of weather data, such as temperature, humidity, and air pressure. You want to create a generic data structure to store this weather data that can work with any data type.

Example 01: Implementation of a generic weather data container that can store any data type for temperature, humidity, and air pressure

```
#include "iostream"
using namespace std;

template <typename T>
class WeatherData {
private:
    T temperature;
```

```
    T humidity;
    T air_pressure;
public:
    WeatherData(T temp, T hum, T press)
            : temperature(temp), humidity(hum), air_pressure(press) {}

    void print_data() {
        cout << "Temperature: " << temperature << " C" << endl;
        cout << "Humidity: " << humidity << " %" << endl;
        cout << "Air Pressure: " << air_pressure << " kPa" << endl;
    }
};

int main() {
    WeatherData<double> data1(25.6, 70.2, 101.3);
    data1.print_data();

    cout << "------------------------" << endl;

    WeatherData<int> data2(20, 65, 100);
    data2.print_data();

    return 0;
}
```

Output

```
Temperature: 25.6 C
Humidity: 70.2 %
Air Pressure: 101.3 kPa
------------------------
Temperature: 20 C
Humidity: 65 %
Air Pressure: 100 kPa
```

Explanation:

In this example, the `WeatherData` class is defined as a `template class` with a single type parameter `T`. This class has private member variables `temperature`, `humidity`, and `air_pressure`, all of type `T`.

The constructor of the `class` takes three parameters of type `T`, which correspond to the `temperature`, `humidity`, and `air_pressure` measurements, respectively. The `print_data` function is a member function of the `class` that prints the stored data to the console.

In the main function, we create two instances of the `WeatherData class`, one with `double` data types and the other with integer data types. The `print_data` function is then called on both instances to display the stored weather data.

Using a template `class` in this way allows us to create a single data structure that can store any type of weather data without needing to define a separate `class` for each data type.

Exception Handling

Exception handling enables the orderly management of run-time errors. Through exception handling, your program can trigger an error-handling routine automatically upon encountering an error. The key benefit of exception handling lies in its ability to automate a significant portion of the error-handling code that was previously manually implemented in large programs.

C++ exception handling relies on three fundamental keywords: **`try`**, **`catch`**, and **`throw`**. In essence, the code segments that require monitoring for exceptions are encapsulated within a **`try`** block. Should an exception, indicating an error, arise within this block, it is triggered using the **`throw`** keyword. Subsequently, the thrown exception is captured and handled through a **`catch`** block. Further elaboration on this basic framework will be provided in the subsequent discussion.

Any code intended for exception monitoring must be enclosed within a **`try`** block. Additionally, functions invoked within a **`try`** block may also generate exceptions. These exceptions, pertinent to the monitored code, are intercepted by a **`catch`** statement, situated immediately after the **`try`** block where the exception originated. The typical structure of **`try`** and **`catch`** blocks is outlined below:

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}
```

What prompts the necessity for Exception Handling in C++?

Here are the primary advantages it offers over conventional error handling methods:

Segregation of Error Handling Code from Normal Code: Traditional error handling involves using if-else conditions to manage errors, intertwining error-handling logic with the main flow of code. This integration diminishes code readability and maintainability. Through try/catch blocks, error handling code becomes distinct from the regular program flow.

Selective Exception Handling by Functions/Methods: A function may throw multiple exceptions but opt to handle only a subset of them. Any unhandled exceptions are propagated to the caller. In C++, functions can declare the exceptions they may throw using the throw keyword.

Consequently, callers are obligated to handle these exceptions either explicitly or by catching them.

Organization of Error Types: C++ permits the throwing of both basic types and objects as exceptions. This facilitates the creation of an exception object hierarchy, enabling the grouping of exceptions within namespaces or classes based on their types.

Example 01: A simple exception handling example

```
#include "iostream"
using namespace std;

int main(){
    cout << "Start " << endl;

    try {
        cout << "Inside try block" << endl;
        throw 20;
        cout << "This will not execute";
    }
    catch (int i) {
        cout << "Caught an exception -- value is: ";
        cout << i << endl;
    }
    cout << "End";
    return 0;
}
```

Output

```
Start
Inside try block
Caught an exception -- value is: 20
End
```

Explanation:

The program starts by printing **"Start"** to the console.

It then enters a `try` block, where it prints **"Inside try block"**.

The `throw 20` statement throws an integer exception with the value `20`.

The line `"This will not execute"` is skipped because an **exception has been thrown.**

The `catch (int i)` block catches the integer exception and prints **"Caught an exception -- value is: 20"**.

Finally, the program prints **"End"**.

Understanding that the code within a **catch** statement executes solely upon catching an exception is crucial. Otherwise, the flow of execution completely bypasses the **catch** block. In simpler terms, if no exception is thrown, the **catch** statement remains inactive, as demonstrated in the program below where no exception is thrown, hence the **catch** block remains unexecuted.

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

int main() {
    cout << "Start" << endl;
    try {
        cout << "Inside try block" << endl;
        cout << "Still inside try block" << endl;
    }
    catch (int i) {
        cout << "Caught an exception -- value is: ";
        cout << i << endl;
    }
    cout << "End";
    return 0;
}
```

Output	Start Inside try block Still inside try block End

As observed, the flow of execution bypasses the **catch** statement.

1. Catching Class Types

Exceptions can take various forms, including those you design as **class** types. In practical applications, **class** types are often preferred over built-in types for exceptions. One primary motivation for defining a **class** type for an exception is to generate an object that provides details about the encountered error. This data can assist the exception handler in effectively managing the error.

Example 01: A simple exception handling example

```
#include "iostream"
using namespace std;

class MyException {
public:
    string str;
    int num;

    MyException(string str, int n) {
```

```
        this->str = str;
        num = n;
    }
};

int main(){
    int i;
    try {
        cout << "Enter a positive number: ";
        cin >> i;
        if (i < 0)
            throw MyException("Not Positive", i);
    }
    catch (MyException e) {
        cout << e.str << ": ";
        cout << e.num << endl;
    }
    return 0;
}
```

Output	Enter a positive number: -10 Not Positive: -10

Explanation: The program requests the user to input a **positive number**. If a **negative number** is provided, an instance of the `MyException` `class` is generated to detail the error. Consequently, `MyException` contains specifics about the encountered issue, which the exception handler utilizes. Typically, it's advisable to develop exception classes that encapsulate error details, facilitating efficient responses by the exception handler.

2. Multiple `catch` Statements

As mentioned, it's possible to include multiple catch blocks within a try statement, a practice often adopted. However, each catch block must handle a distinct type of exception.

Example 01: A program that catches both integers and strings.

```
#include "iostream"
using namespace std;

void exceptionHandler(int test){

        try{
            if(test) throw test;
            else throw "Value is zero";
        }
        catch(int i){
            cout << "Caught Exception #: " << i << endl;
        }
        catch(const char *str){
            cout << "Caught a string: ";
```

```
            cout << str << endl;
        }
};

int main(){
    cout << "Start" << endl;
    exceptionHandler(1);
    exceptionHandler(2);
    exceptionHandler(0);
    exceptionHandler(3);
    cout << "End";
    return 0;
}
```

Output	Start Caught Exception #: 1 Caught Exception #: 2 Caught a string: Value is zero Caught Exception #: 3 End

Explanation: Each **catch** block exclusively handles exceptions of its specified type. Typically, **catch** blocks are examined in the sequence they appear in the program. Only the **catch** block that matches the thrown exception type is executed, while all other catch blocks are disregarded.

3. Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of catch.

```
catch (...) {
// process all exceptions
}
```

Example 01: A simple program that catches all exceptions.

```
#include "iostream"
using namespace std;

void exceptionHandler(int test){
    try{
        if(test==0) throw test;
        if(test==1) throw 'a';
        if(test==2) throw 123.23;
    }
    catch(...) { // catch all exceptions
        cout << "Caught One!" << endl ;
    }
}
```

```
int main(){
    cout << "Start" << endl;
    exceptionHandler(0);
    exceptionHandler(1);
    exceptionHandler(2);
    cout << "End";

    return 0;
}
```

Output

```
Start
Caught One!
Caught One!
Caught One!
End
```

Explanation: As you can see, all three throws were caught using the one **`catch(...)`** statement.

Example 02: A simple program that uses `catch(...)` as a default

```
#include "iostream"
using namespace std;

void exceptionHandler(int test){
    try{
        if(test==0) throw test;
        if(test==1) throw 'a';
        if(test==2) throw 123.23;
    }
    catch (int i){
        cout << "Caught an integer" << endl;
    }
    catch(...) { // catch all exceptions
        cout << "Caught One!" << endl;
    }
}

int main(){
    cout << "Start" << endl;
    exceptionHandler(0);
    exceptionHandler(1);
    exceptionHandler(2);
    cout << "End";

    return 0;
}
```

Output

```
Start
Caught an integer
Caught One!
Caught One!
End
```

Exception Handling: Final Program

Example: A Generic Program for Division with Error Handling

```
#include "iostream"
using namespace std;

template <typename T>
class Division {
public:
    void divide(T a, T b) {
        try {
            if (!b) throw b; // check for divide-by-zero
            cout << "Result: " << a / b << endl;
        }
        catch (T b) {
            cout << "Can't divide by zero." << endl;
        }
    }
};

int main() {

    Division <double> d1;
    d1.divide(15.5,0.0);
    Division <int> d2;
    d2.divide(15,0);
    Division <double> d3;
    d3.divide(12.56,17.92);

    return 0;
}
```

Output	Can't divide by zero. Can't divide by zero. Result: 0.700893

Explanation: In this program we have define a `template class Division` that performs division operations. It checks for division by zero and handles it with an exception. The `main` function creates objects of `Division` with different data types (`double` and `int`) and performs divisions, **including division by zero**, which triggers the **exception handling**.

OOP LAB 13 TASKS

1. Write a program where an exception is thrown from a function deep down the call stack. Create a chain of function calls (e.g., funcA() -> funcB() -> funcC() -> throwException()). Inside throwException(), throw a custom exception. Ensure that each function in the call chain has appropriate try and catch blocks to catch and handle this exception. Observe how the exception is propagated up the call stack and how stack unwinding occurs.

2. Create a program with multiple levels of function calls. Implement a function **processData()** that calls another function **parseData()**, which in turn may throw exceptions related to data parsing errors (e.g., invalid format). Use exception handling in **processData** to catch these exceptions, log an error message, and rethrow the exception with additional context (if needed). Catch the rethrown exception in **main** and display an appropriate error message.

3. Create a template function **matrixMultiply** that performs matrix multiplication for two-dimensional arrays (matrices) of any numeric type (**int**, **double**, etc.).

4. Create a template function **factorial** that computes the factorial of a given non-negative integer using template metaprogramming (compile-time computation).'

5. Create a class containing a function template capable of returning the sum of all the elements in an array being passed as parameter. Show the results for two arrays, one integer type and the other double type.