OBJECT PROGRAMMING

- LECTURE 10 -

(1ST YEAR OF STUDY)

Skopje, 2023/24

Contents

10. Exception Handling

10.1. Introduction

10.2. Properties of Exception Handling

10.3. Exception Handling using Classes

10.4. Stack Unwinding in C++

10.5. User-defined Custom Exception

10.1. Introduction

- In C++, **exceptions** are run-time anomalies, or abnormal conditions, that a program encounters during its execution.
- The process of handling these exceptions is called **exception handling**. Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
- So basically using *exception handling*, we can handle the exceptions so that our program keeps running.

10.1. Introduction

- An **exception** is an unexpected problem that arises during the execution of a program, when our program terminates suddenly with some errors/issues. An *exception* occurs during the running of the program (run-time).
- There are *two* types of exceptions in C++:
  - **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as *dividing a number by zero*.
  - **Asynchronous**: Exceptions that are beyond the program's control, such as *disc failure*, *keyboard interrupts*, etc.

10.1. Introduction

- C++ provides an in-built feature for Exception Handling. It can be done using the following specialized keywords: ***try***, ***catch***, and ***throw***, with each having a different purpose.
- Syntax of **try-catch** in C++:

```
try {
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
    // catch block catches the exception that is thrown from try block
 }
```

10.1. Introduction

- **1. try in C++**
  - The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.
- **2. catch in C++**
  - The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

10.1. Introduction

- **3. throw in C++**
  - An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.
- **Note:** Multiple catch statements can be used to catch different type of exceptions thrown by try block.
- The ***try*** and ***catch*** keywords come *in pairs*: We use the *try* block to test some code and If the code throws an exception, we will handle it in our *catch* block.

10.1. Introduction

- **Why do we need Exception Handling in C++?**
- The following are the main advantages of exception handling over traditional error handling:
  - **1. Separation of Error Handling Code from Normal Code**: With try/catch blocks, the code for error handling becomes separate from the normal flow.
  - **2. Functions/Methods can handle only the exceptions they choose**: A function can throw many exceptions, but may choose to handle some of them.
  - **3. Grouping of Error Types**: In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

10.1. Introduction

- **Example 1**: This example demonstrates throw exceptions.

```
// C++ program to demonstate the use of try,catch and throw
// in exception handling.

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

int main()
{

    // try block
    try {
        int numerator = 10;
        int denominator = 0;
        int res;

        // check if denominator is 0 then throw runtime
        // error.
        if (denominator == 0) {
            throw runtime_error(
                "Division by zero not allowed!");
        }
        // calculate result if no exception occurs
        res = numerator / denominator;
        //[printing result after division
        cout << "Result after division: " << res << endl;
    }
    // catch block to catch the thrown exception
    catch (const exception& e) {
        // print the exception
        cout << "Exception " << e.what() << endl;
    }

    return 0;
}
```

Output

```
Exception Division by zero not allowed!
```

10.1. Introduction

- **Example 2:** The output of the program explains the flow of execution of try/catch blocks.

```cpp
// C++ program to demonstate the use of try,catch and throw
// in exception handling.

#include <iostream>
using namespace std;

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";

    // try block
    try {
        cout << "Inside try \n";
        if (x < 0) {
            // throwing an exception
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }

    // catch block
    catch (int x) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

Output

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

10.2. Properties of Exception Handling

- **Property 1:**
  - There is a special catch block called the ***'catch-all'*** block, written as catch (…), that can be used to catch all types of exceptions.
- **Example**
  - In the following program, an ***int*** is thrown as an exception, but there is no catch block for ***int***, so the catch (…) block will be executed.

10.2. Properties of Exception Handling


```cpp
// C++ program to demonstate the use of catch all
// in exception handling.

#include <iostream>
using namespace std;

int main()
{
    // try block
    try {

        // throw
        throw 10;
    }

    // catch block
    catch (char* excp) {
        cout << "Caught " << excp;
    }

    // catch all
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output

```
Default Exception
```

10.2. Properties of Exception Handling

- **Property 2:**
  - Implicit type conversion does not happen for primitive types.
- **Example**
  - In the following program, ‘a’ is not implicitly converted to ***int***.

10.2. Properties of Exception Handling


```cpp
//// C++ program to demonstate property 2: Implicit type
/// conversion doesn't happen for primitive types.
// in exception handling.

#include <iostream>
using namespace std;

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output

```
Default Exception
```

10.2. Properties of Exception Handling

- **Property 3:**
  - If an exception is thrown and not caught anywhere, the program terminates abnormally.
- **Example**
  - In the following program, a ***char*** is thrown, but there is no catch block to catch the ***char***.

10.2. Properties of Exception Handling


```cpp
// C++ program to demonstate property 3: If an exception is
// thrown and not caught anywhere, the program terminates
// abnormally in exception handling.

#include <iostream>
using namespace std;

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

Output

```
terminate called after throwing an instance of 'char'
```

10.2. Properties of Exception Handling

- **Property 4:**
  - Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler does not check whether an exception is caught or not. So, it is not necessary to specify all uncaught exceptions in a function declaration. However, exception-handling it's a recommended practice to do so.
- **Example**
  - The following program compiles fine, but ideally, the signature of fun() should list the unchecked exceptions.

10.2. Properties of Exception Handling


```cpp
// C++ program to demonstate property 4 in exception
// handling.

#include <iostream>
using namespace std;

// This function signature is fine by the compiler, but not
// recommended. Ideally, the function should specify all
// uncaught exceptions and function signature should be
// "void fun(int *ptr, int x) throw (int *, int)"
void fun(int* ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
```

```cpp
int main()
{
    try {
        fun(NULL, 0);
    }
    catch (...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output

```
Caught exception from fun()
```

10.2. Properties of Exception Handling

- A better way to write the above code:

```
// C++ program to demonstate property 4 in better way

#include <iostream>
using namespace std;

// Here we specify the exceptions that this function
// throws.
void fun(int* ptr, int x) throw(
    int*, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
```

```
int main()
{
    try {
        fun(NULL, 0);
    }
    catch (...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output

```
Caught exception from fun()
```

10.2. Properties of Exception Handling

- **Property 5:**
  - In C++, **try/catch** blocks can be nested. Also, an exception can be re-thrown using “throw;“.
- **Example**
  - The following program shows *try/catch* blocks nesting.

10.2. Properties of Exception Handling


```
// C++ program to demonstrate try/catch blocks can be nested
// in C++

#include <iostream>
using namespace std;

int main()
{

    // nesting of try/catch
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output

```
Handle Partially Handle remaining
```

10.2. Properties of Exception Handling

- **Property 6:**
  - When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.
- **Example**
  - The following program demonstrates the above property.

10.2. Properties of Exception Handling


```cpp
// C++ program to demonstrate

#include <iostream>
using namespace std;

// Define a class named Test
class Test {
public:
    // Constructor of Test
    Test() { cout << "Constructor of Test " << endl; }
    // Destructor of Test
    ~Test() { cout << "Destructor of Test " << endl; }
};
```

```cpp
int main()
{
    try {
        // Create an object of class Test
        Test t1;

        // Throw an integer exception with value 10
        throw 10;
    }
    catch (int i) {
        // Catch and handle the integer exception
        cout << "Caught " << i << endl;
    }
}
```

Output

```
Constructor of Test
Destructor of Test
Caught 10
```

10.2. Properties of Exception Handling

- The exception handling in C++ also has a few limitations:
  - Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
  - If exception handling is not done properly can lead to resource leaks as well.
  - It is hard to learn how to write Exception code that is safe.
  - There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.

10.2. Properties of Exception Handling

- *Exception handling* in C++ is used to handle unexpected happening using "try" and "catch" blocks to manage the problem efficiently.
- This *exception handling* makes our programs more ***reliable*** as errors at run-time can be handled separately, and it also helps prevent the program from crashing and abrupt termination of the program when error is encountered.

10.3. Exception Handling using Classes

- **Exceptions** are run-time anomalies or abnormal conditions that a program encounters during its execution.
- There are two types of exceptions:
  - Synchronous Exception
  - Asynchronous Exception (e.g. which are beyond the program's control, disc failure, etc).
- C++ provides the following specialized keywords:
  - **try:** represents a block of code that can throw an exception.
  - **catch:** represents a block of code that is executed when a particular exception is thrown.
  - **throw:** Used to throw an exception. Also used to list the exceptions that a function throws, but does not handle itself.

10.3. Exception Handling using Classes

- **Problem Statement:**
  - Create a class Numbers which has two data members **a** and **b.**
  - Write iterative functions to find the **GCD of two numbers.**
  - Write an iterative function to check whether any given number is prime or not. If found to be **true,** then throws an exception to class **MyPrimeException.**
  - Define your own **MyPrimeException** class.

10.3. Exception Handling using Classes

- **Solution:**
  - Define a class named **Number** which has two private data members as **a** and **b.**
  - Define two member functions as:
    - **int gcd():** to *calculate the HCF of the two numbers.*
    - **bool isPrime():** to *check if the given number is prime or not.*
  - Use **constructor** which is used to initialize the data members.
  - Take another class named **Temporary** which will be called when an exception is thrown.

10.3. Exception Handling using Classes


```cpp
// C++ program to illustrate the concept
// of exception handling using class

#include <bits/stdc++.h>
using namespace std;

// Class declaration
class Number {
private:
    int a, b;

public:
    // Constructors
    Number(int x, int y)
    {
        a = x;
        b = y;
    }

// Function that find the GCD
// of two numbers a and b
int gcd()
{
    // While a is not equal to b
    while (a != b) {

        // Update a to a - b
        if (a > b)
            a = a - b;

        // Otherwise, update b
        else
            b = b - a;
    }

    // Return the resultant GCD
    return a;
}

    // Function to check if the
    // given number is prime
    bool isPrime(int n)
    {
        // Base Case
        if (n <= 1)
            return false;

        // Iterate over the range [2, N]
        for (int i = 2; i < n; i++) {

            // If n has more than 2
            // factors, then return
            // false
            if (n % i == 0)
                return false;
        }

        // Return true
        return true;
    }
};

// Empty class
class MyPrimeException {
};
```

10.3. Exception Handling using Classes


```cpp
// Driver Code
int main()
{
    int x = 13, y = 56;

    Number num1(x, y);

    // Print the GCD of X and Y
    cout << "GCD is = "
         << num1.gcd() << endl;

    // If X is prime
    if (num1.isPrime(x))
        cout << x
             << " is a prime number"
             << endl;

    // If Y is prime
    if (num1.isPrime(y))
        cout << y
             << " is a prime number"
             << endl;

    // Exception Handling
    if ((num1.isPrime(x))
        || (num1.isPrime(y))) {

        // Try Block
        try {
            throw MyPrimeException();
        }

        // Catch Block
        catch (MyPrimeException t) {

            cout << "Caught exception "
                 << "of MyPrimeException "
                 << "class." << endl;
        }
    }

    return 0;
}
```

Output:

```
GCD is = 1
13 is a prime number
Caught exception of MyPrimeException class.
```

10.4. Stack Unwinding in C++

- **Stack Unwinding** is the process of removing function entries from function call stack at run-time. The local objects are destroyed in reverse order in which they were constructed.
- *Stack Unwinding* is generally related to *Exception Handling*. In C++, when an exception occurs, the function call stack is linearly searched for the *exception handler*, and all the entries before the function with exception handler are removed from the function call stack.

10.4. Stack Unwinding in C++

- So, *Exception Handling* involves *Stack Unwinding* if an exception is not handled in the same function (where it is thrown).
- Basically, Stack Unwinding is a process of calling the destructors (whenever an exception is thrown) for all the automatic objects constructed at run-time.
- For example, let us consider the following program:

10.4. Stack Unwinding in C++


```cpp
// CPP Program to demonstrate Stack Unwinding
#include <iostream>
using namespace std;

// A sample function f1() that throws an int exception
void f1() throw(int)
{
    cout << "\n f1() Start ";
    throw 100;
    cout << "\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw(int)
{
    cout << "\n f2() Start ";
    f1();
    cout << "\n f2() End ";
}

// Another sample function f3() that calls f2() and handles
// exception thrown by f1()
void f3()
{
    cout << "\n f3() Start ";
    try {
        f2();
    }
    catch (int i) {
        cout << "\n Caught Exception: " << i;
    }
    cout << "\n f3() End";
}

// Driver Code
int main()
{
    f3();

    getchar();
    return 0;
}
```

Output

```
f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End
```

10.4. Stack Unwinding in C++

- **Explanation:**
  - When f1() throws exception, its entry is removed from the function call stack, because f1() does not contain exception handler for the thrown exception, then next entry in call stack is looked for exception handler.
  - The next entry is f2(). Since f2() also does not have a handler, its entry is also removed from the function call stack.
  - The next entry in the function call stack is f3(). Since f3() contains an exception handler, the catch block inside f3() is executed, and finally, the code after the catch block is executed.

10.4. Stack Unwinding in C++

- Note that the following lines inside f1() and f2() are not executed at all.

```
cout<<"\n f1() End ";   // inside f1()

cout<<"\n f2() End ";   // inside f2()
```

- If there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in the Stack Unwinding process.
- **Note:** Stack Unwinding also happens in Java when exception is not handled in same function.

10.5. User-defined Custom Exception

- We can use **Exception Handling** with class, too. Even we can throw an exception of **user-defined** class types. For throwing an exception of say **demo** class type within **try** block we may write

```
throw demo();
```

- **Example 1:** Program to implement exception handling with single class.
  - We have declared an empty class. In the **try** block we are throwing an object of **demo** class type. The **try** block catches the object and displays.

10.5. User-defined Custom Exception


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

class demo {
};

int main()
{
    try {
        throw demo();
    }

    catch (demo d) {
        cout << "Caught exception of demo class \n";
    }
}
```

Output:

```
Caught exception of demo class
```

10.5. User-defined Custom Exception

- **Example 2:** Program to implement exception handling with two classes.

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

class demo1 {
};

class demo2 {
};

int main()
{
    for (int i = 1; i <= 2; i++) {
        try {
            if (i == 1)
                throw demo1();

            else if (i == 2)
                throw demo2();
        }
        catch (demo1 d1) {
            cout << "Caught exception of demo1 class \n";
        }

        catch (demo2 d2) {
            cout << "Caught exception of demo2 class \n";
        }
    }
}
```

Output:

```
Caught exception of demo1 class
Caught exception of demo2 class
```

10.5. User-defined Custom Exception

- **Exception handling with inheritance:**
  - Exception handling can also be implemented with the help of ***inheritance***. In case of *inheritance* object thrown by derived class is caught by the first catch block.

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

class demo1 {
};

class demo2 : public demo1 {
};

int main()
{
    for (int i = 1; i <= 2; i++) {
        try {
            if (i == 1)
                throw demo1();

            else if (i == 2)
                throw demo2();
        }
        catch (demo1 d1) {
            cout << "Caught exception of demo1 class \n";
        }
        catch (demo2 d2) {
            cout << "Caught exception of demo2 class \n";
        }
    }
}
```

Output:

```
Caught exception of demo1 class
Caught exception of demo1 class
```

10.5. User-defined Custom Exception

- **Explanation:** The program is similar to previous one, but here we have made **demo2** as derived class for **demo1**. Note the **catch** block for **demo1** is written first. As **demo1** is base class for **demo2**, an object thrown of **demo2** class will be handled by first **catch** block. That is why output is as shown.
- **Exception handling with constructor:**
  - Exception handling can also be implemented by using ***constructor***. Though we cannot return any value from the constructor, but with the help of **try** and **catch** block we can.

10.5. User-defined Custom Exception


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

class demo {
    int num;

public:
    demo(int x)
    {
        try {

            if (x == 0)
                // catch block would be called
                throw "Zero not allowed ";

            num = x;
            show();
        }

        catch (const char* exp) {
            cout << "Exception caught \n ";
            cout << exp << endl;
        }
    }

    void show()
    {
        cout << "Num = " << num << endl;
    }
};

int main()
{

    // constructor will be called
    demo(0);
    cout << "Again creating object \n";
    demo(1);
}
```

Output:

```
Exception caught
 Zero not allowed
Again creating object
Num = 1
```