LAB13: Exception Handling

Array

Rung-Bin Lin

International Bachelor Program in Informatics
Yuan Ze University
5/19/2022

Exception

- **An exception is an indication of a problem that occurs during a program's execution.**
  - **An exception can be caused by divided by zero, not enough memory during memory allocation, invalid arguments, overflow, underflow, out of range for access to an array, etc.**
- **Exception handling enables us to create applications that can resolve (or handle) exceptions.**
- **Exception handling may allow a program to continue its execution if no problem had been met. Otherwise, its execution may be terminated in response to a severe problem.**

Construct for Watching and Handling Exceptions

- Use a try block for watching exceptions
- Use a catch block for capturing exceptions
- A try block should be immediately followed by at least one catch block.
  - Typically, there is a throw statement in a try block.
- A catch block should have a corresponding try block.
- If an exception is thrown in a try block, the try block is expired. This means the program execution exits from the try block at the place where the throw statement is executed or the statement causes exception.
- A catch block is executed if the type specified by the throw is matched the type specified by the catch block.
- The program resumes execution from the last catch block after handling an exception.

Standard Library Exception classes

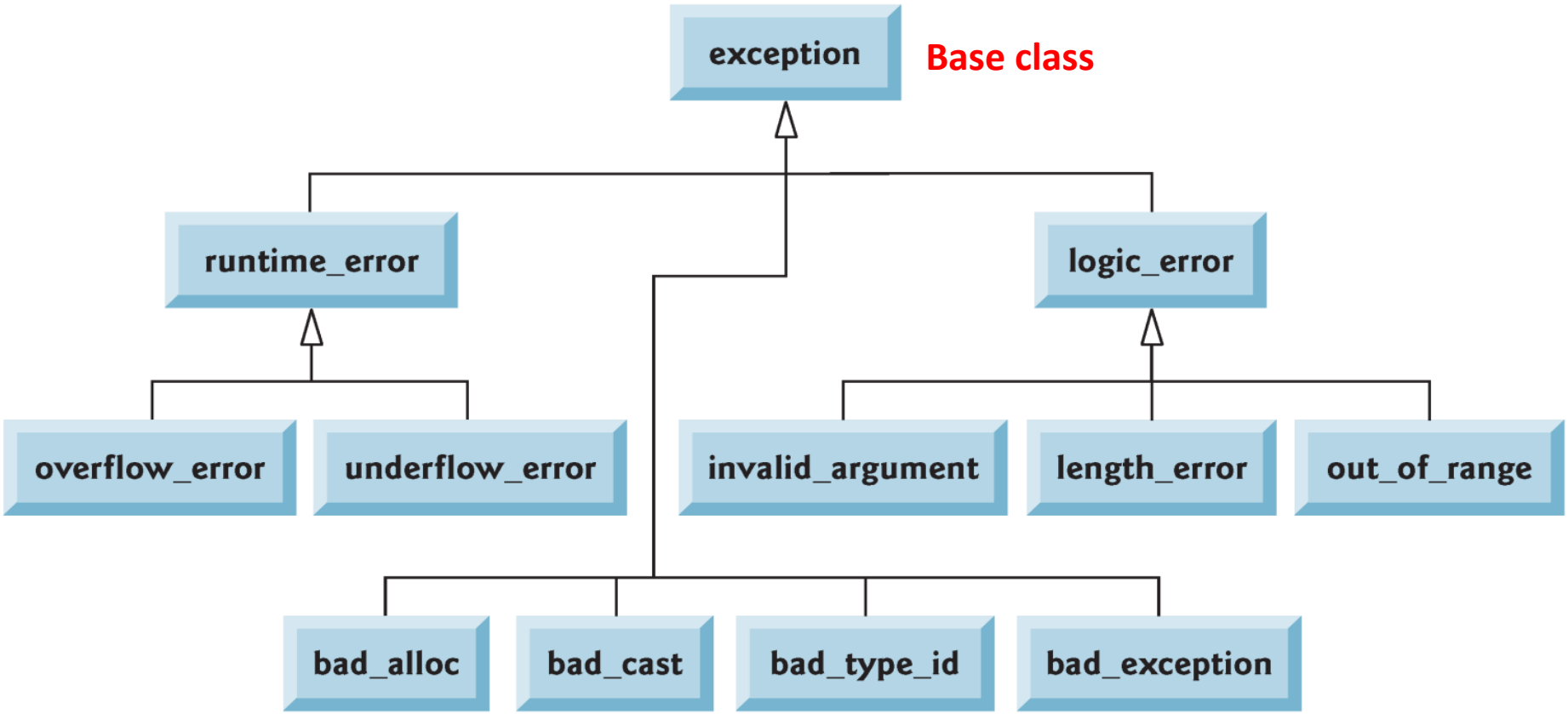

Fig. 16.10 | Some of the Standard Library exception classes.

Derived our Own Exception Class

```
1  // Fig. 16.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header file contains runtime_error
4  using namespace std;
5
6  // DivideByZeroException objects should be thrown by functions
7  // upon detecting division-by-zero exceptions
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // constructor specifies default error message
12    DivideByZeroException()
13       : runtime_error( "attempted to divide by zero" ) {}
14 }; // end class DivideByZeroException
```

Fig. 16.1 | Class `DivideByZeroException` definition.

DivideByZeroException is derived from runtime_error.

Example of Divide-by-Zero Exception (1)

```
1  // Fig. 16.2: Fig16_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  #include "DivideByZeroException.h" // DivideByZeroException class
6  using namespace std;
7
8  // perform division and throw DivideByZeroException object if
9  // divide-by-zero exception occurs
10 double quotient( int numerator, int denominator )
11 {
12    // throw DivideByZeroException if trying to divide by zero
13    if ( denominator == 0 )  // Calling Constructor to create an object of DivideByZeroEX....
14       throw DivideByZeroException(); // terminate function
15
16    // return division result
17    return static_cast< double >( numerator ) / denominator;
18 } // end function quotient
19
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 3.)

Example of Divide-by-Zero Exception (2)

```
20  int main()
21  {
22     int number1; // user-specified numerator
23     int number2; // user-specified denominator
24     double result; // result of division
25
26     cout << "Enter two integers (end-of-file to end): ";
27
28     // enable user to enter two integers to divide
29     while ( cin >> number1 >> number2 )
30     {
31        // try block contains code that might throw exception
32        // and code that should not execute if an exception occurs
33        try
34        {
35           result = quotient( number1, number2 );
36           cout << "The quotient is: " << result << endl;
37        } // end try
38        catch ( DivideByZeroException &divideByZeroException )
39        {
40           cout << "Exception occurred: "
41              << divideByZeroException.what() << endl;
42        } // end catch
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 3.)

Output

```
43
44          cout << "\nEnter two integers (end-of-file to end): ";
45       } // end while
46
47       cout << endl;
48    } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 3 of 3.)

Exception for Dynamic Memory Allocation with new Function (1)

```
1  // Fig. 16.5: Fig16_05.cpp
2  // Demonstrating standard new throwing bad_alloc when memory
3  // cannot be allocated.
4  #include <iostream>
5  #include <new> // bad_alloc class is defined here
6  using namespace std;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // aim each ptr[i] at a big block of memory
13    try
14    {
15       // allocate memory for ptr[ i ]; new throws bad_alloc on failure
16       for ( int i = 0; i < 50; i++ )
17       {                                                  // Without a throw statement.
18          ptr[ i ] = new double[ 50000000 ]; // may throw exception
19          cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
20       } // end for
21    } // end try
```

Fig. 16.5 | new throwing bad_alloc on failure. (Part 1 of 2.)

Exception for Dynamic Memory Allocation with new Function (2)

```
22      catch ( bad_alloc &memoryAllocationException )
23      {
24         cerr << "Exception occurred: "
25            << memoryAllocationException.what() << endl;
26      } // end catch
27   } // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation
```

Fig. 16.5 | new throwing bad_alloc on failure. (Part 2 of 2.)

More than One catch Block

- **To handle different types of exceptions occurring within the same try block.**

```
double quotient(int numerator, int denominator)
{
   if(denominator == 0)
      throw DivideByZeroException();  // DivideByZeroException type
   else if(numerator == 0)
      throw numerator;  // integer type

   return static_cast<double>(numerator)/denominator;
}
```

```
int main() {
   int number1;
   int number2;
   double result;
   while(cin >> number1 >> number2) {
         try {
            result = quotient(number1, number2);
            cout << "The quotient of " << number1 << " to " << number2 << " is "
            << result << endl;
         }
         catch (DivideByZeroException &divideByZeroHandler) {
            cout << "Exception of Divided-by-Zero occurs: "
            << divideByZeroHandler.what() << endl;
         }
         catch (int anInt){
         cout << "Exception because numerator is zero: "
            <<  anInt << endl;
         cout << "The quotient of " << number1 << " to " << number2 << " is "
            << number1 << endl;
         }
         catch (...){  // Must be the last catch block
         cout << "Catch any exceptions!" << endl;
         }
      }
   cout << endl;
   return 0;
}
```

Rethrow an Exception

```
1  // Fig. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
6
7  // throw, catch and rethrow exception
8  void throwException()
9  {
10    // throw exception and catch it immediately
11    try
12    {
13       cout << "  Function throwException throws an exception\n";
14       throw exception(); // generate exception
15    } // end try
16    catch ( exception & ) // handle exception
17    {
18       cout << "  Exception handled in function throwException"
19          << "\n  Function throwException rethrows exception";
20       throw; // rethrow exception for further processing
21    } // end catch
22
23    cout << "This also should not print\n";
24 } // end function throwException
```

Fig. 16.3 | Rethrowing an exception. (Part 1 of 3.)

```
25
26  int main()
27  {
28     // throw exception
29     try
30     {
31        cout << "\nmain invokes function throwException\n";
32        throwException();
33        cout << "This should not print\n";
34     } // end try
35     catch ( exception & ) // handle exception
36     {
37        cout << "\n\nException handled in main\n";
38     } // end catch
39
40     cout << "Program control continues after catch in main\n";
41  } // end main
```

Fig. 16.3 | Rethrowing an exception. (Part 2 of 3.)

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Fig. 16.3 | Rethrowing an exception. (Part 3 of 3.)

Lab13 : Exception Handling for Array

- **Modify the code in Fig. 11.6, 11.7 & 11.8 so that the following exceptions must be issued.**
  - **An exception will occur when there is an out-of-range access to an array.**
  - **When there is not enough memory for creating an array.**
  - **When an array is assigned to itself.**
- **For this lab, this means that you have to add a try block and a catch block to the operator functions and modify part of main() function.**
  - **In a try block, when an exception occurs, you may have to rethrow an exception.**
- **Only the designated part of main() function can be modified.**

Main Program (1)

//Remember to add #include <stdexcept> and <new> to the header file.

```
6   int main()
7   {
8      Array integers1( 5 ); // seven-element Array
9      Array integers2(7); // 7-element Array by default
10     cout << "\nEnter 12 integers:" << endl;
11     cin >> integers1 >> integers2;
12
13     cout << "\nAfter input, the Arrays contain:\n"
14        << "integers1:\n" << integers1
15        << "integers2:\n" << integers2;
16     cout << "Execute  integers1 = integers1" << endl;
17      integers1 = integers1;
18     try{
19     cout << "\nintegers2[25] is " << integers2[ 25 ] << endl;
20     cout << "\nintegers1[2] is " << integers1[ 2] << endl;
21     }
22     catch (int &inx) {
23        cout << "Array reading is not done due to bad index " << inx <<endl;
24     }
25     try {
26     cout << "\n\nAssigning 1000 to integers2[6]" << endl;
27     integers2[ 6 ] = 1000;
28     cout << "integers2:\n" << integers2;
```

Main Program (2)

```
29      //attempt to use out-of-range subscript
30      cout << "\nAttempt to assign 1000 to integers1[23]" << endl;
31      integers1[ 23 ] = 1000; // ERROR: out of range
32      cout << "integers1:\n" << integers1;
33      cout << "\n\nAssigning 1000 to integers1[4]" << endl;
34      integers1[ 4 ] = 5000;
35      cout << "integers1:\n" << integers1;
36      }
37       catch (int &inx) {
38          cout << "Array writing is not done due to bad index " << inx << endl << endl;
39      }
40        cout << "integers1:\n" << integers1;
41
42      Array ptr[50];
43      cout << "Memory allocation for creating large arrays.\n";
44
45      for (int i=0; i<50; i++){
46          Array Integerx(30000000);
47          ptr[i] = Integerx;
48          cout << "ptr[" << i << "] points to 30,000,000 new integers\n";
49      }
50
51
52
53
54       return 0;
55  } // end main
```

Make the above for loop into a try block and add a catch block here after the try block. However, the statements before line 44 should not be modified.

Example Output

```
Enter 12 integers:
1 2 3 4 5 6 7 8 9 10 11 12

After input, the Arrays contain:
integers1:
         1          2          3          4
         5
integers2:
         6          7          8          9
        10         11         12
Execute  integers1 = integers1
ARRAY TRIES TO ASSIGN TO ITSELF!!

Bad array index(left value): 25
Array reading is not done due to bad index 25


Assigning 1000 to integers2[6]
integers2:
         6          7          8          9
        10         11       1000

Attempt to assign 1000 to integers1[23]
Bad array index(left value): 23
Array writing is not done due to bad index 23

integers1:
         1          2          3          4
         5
Memory allocation for creating large arrays.
ptr[0] points to 30,000,000 new integers
ptr[1] points to 30,000,000 new integers
ptr[2] points to 30,000,000 new integers
ptr[3] points to 30,000,000 new integers
ptr[4] points to 30,000,000 new integers
ptr[5] points to 30,000,000 new integers
ptr[6] points to 30,000,000 new integers
ptr[7] points to 30,000,000 new integers
ptr[8] points to 30,000,000 new integers
ptr[9] points to 30,000,000 new integers
ptr[10] points to 30,000,000 new integers
ptr[11] points to 30,000,000 new integers
ptr[12] points to 30,000,000 new integers
ptr[13] points to 30,000,000 new integers
Memory allocation exception occurs: std::bad_alloc
```

To print out this message, you have to make a throw and a catch block in the operator=. The catch block may have to receive a bool type object in the parameter list.

The number of lines printed may be different from computer to computer used for running the program.

Key Points for TA Grading

- Both &operator [ ] and operator[ ] must have a try and catch blocks.
- Should add a try and a catch block in the main() function after line 44. However, the statements before line 44 should not be modified.
- The output should be correct, especially the part highlighted in the red rectangles. No other outputs should be printed.