

CPP Chapter 11: Errors and Exception Handling

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Exceptions

- Exception: undesirable event detectable during program execution
- If exceptions occurred during execution
 - Programmer-supplied code terminated the program or
 - Program terminated with an appropriate error message
- Can add exception-handling code at point where an error can occur

Handling Exceptions within a Program

- Function `assert`:
 - Checks if an expression meets certain condition(s)
 - If conditions are not met, it terminates the program
- Example: division by 0
 - If divisor is zero, `assert` terminates the program with an error message

```

#include <iostream>

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    cout << "Line 2: Enter the dividend: ";    //Line 2
    cin >> dividend;                            //Line 3
    cout << endl;                               //Line 4

    cout << "Line 5: Enter the divisor: ";      //Line 5
    cin >> divisor;                             //Line 6
    cout << endl;                               //Line 7

    quotient = dividend / divisor;              //Line 8
    cout << "Line 9: Quotient = " << quotient  //Line 9
         << endl;

    return 0;                                   //Line 10
}

```

Sample Run 1:

Line 2: Enter the dividend: 12

Line 5: Enter the divisor: 5

Line 9: Quotient = 2

Sample Run 2:

Line 2: Enter the dividend: 24

Line 5: Enter the divisor: 0

abcfgh.exe has encountered a problem and needs to close. We are sorry for the inconvenience.



```
#include <iostream>

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    cout << "Line 2: Enter the dividend: ";    //Line 2
    cin >> dividend;                            //Line 3
    cout << endl;                              //Line 4

    cout << "Line 5: Enter the divisor: ";      //Line 5
    cin >> divisor;                             //Line 6
    cout << endl;                              //Line 7

    if (divisor != 0)                          //Line 8
    {
        quotient = dividend / divisor;         //Line 9
        cout << "Line 10: Quotient = " << quotient //Line 10
            << endl;
    }
    else                                       //Line 11
        cout << "Line 12: Cannot divide by zero." //Line 12
            << endl;
    return 0;                                //Line 13
}
```

Sample Run 1:

Line 2: Enter the dividend: 12

Line 5: Enter the divisor: 5

Line 10: Quotient = 2

Sample Run 2:

Line 2: Enter the dividend: 24

Line 5: Enter the divisor: 0

Line 12: Cannot divide by zero.

```

#include <iostream>
#include <cassert>

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    cout << "Line 2: Enter the dividend: ";   //Line 2
    cin >> dividend;                           //Line 3
    cout << endl;                             //Line 4

    cout << "Line 5: Enter the divisor: ";     //Line 5
    cin >> divisor;                             //Line 6
    cout << endl;                             //Line 7

    assert(divisor != 0);                      //Line 8
    quotient = dividend / divisor;             //Line 9

    cout << "Line 10: Quotient = " << quotient //Line 10
         << endl;

    return 0;                                 //Line 11
}

```

Sample Run 1:

Line 2: Enter the dividend: 26

Line 5: Enter the divisor: 7

Line 10: Quotient = 3

Sample Run 2:

Line 2: Enter the dividend: 24

Line 5: Enter the divisor: 0

Assertion failed: divisor != 0, file c:\chapter16 source code\ch16_exp3.cpp, line 19

Exception Handling in C++

- The `try/catch` block handles exceptions
- Exception must be thrown in a `try` block and caught by a catch block
- C++ provides support to handle exceptions via a hierarchy of classes

try/catch Block

- Statements that may generate an exception are placed in a `try` block
- The `try` block also contains statements that should not be executed if an exception occurs
- The `try` block is followed by one or more `catch` blocks

try/catch Block (continued)

- The `catch` block:
 - Specifies the type of exception it can catch
 - Contains an exception handler
- If the heading of a `catch` block contains ... (ellipses) in place of parameters
 - Catch `block` can catch exceptions of all types

- General syntax of the try/catch block:

```
try
{
    //statements
}
catch (dataType1 identifier)
{
    //exception handling code
}
.
.
.
catch (dataType2 identifier)
{
    //exception handling code
}
.
.
.
catch (...)
{
    //exception handling code
}
```

try/catch Block (continued)

- If no exception is thrown in a `try` block
 - All `catch` blocks for that `try` block are ignored
 - Execution resumes after the last `catch` block
- If an exception is thrown in a `try` block
 - Remaining statements in that `try` block are ignored

try/catch Block (continued)

- The program searches `catch` blocks in order, looking for an appropriate exception handler
- If the type of thrown exception matches the parameter type in one of the `catch` blocks:
 - Code of that `catch` block executes
 - Remaining `catch` blocks are ignored

try/catch Block (continued)

```
catch (int x)
{
    //exception handling code
}
```

In this `catch` block:

- The identifier `x` acts as a parameter. In fact, it is called a `catch` block parameter.
- The data type `int` specifies that this `catch` block can catch an exception of type `int`.
- A `catch` block can have *at most* one `catch` block parameter.

Throwing an Exception

- For `try/catch` to work, the exception must be thrown in the `try` block
- General syntax to throw an exception is:

```
throw expression;
```

where `expression` is a constant value, variable, or object

Throwing an Exception (continued)

- The object being thrown can be:
 - Specific object
 - Anonymous object
- In C++
 - An exception is a value
 - `throw` is a reserved word

Throwing an Exception (continued)

Suppose we have the following declaration:

```
int num = 5;  
string str = "Something is wrong!!!";
```

throw expression

```
throw 4;  
throw x;  
throw str;  
throw string("Exception found!");
```

Effect

The constant value 4 is thrown.
The value of the variable x is thrown.
The object str is thrown.
An anonymous string object with the string "Exception found!" is thrown.

Order of catch Blocks

- Catch block can catch
 - All exceptions of a specific type
 - All types of exceptions
- A `catch` block with an ellipses (three dots) catches any type of exception
- In a sequence of `try/catch` blocks, if the `catch` block with an ellipses is needed
 - It should be the last `catch` block of that sequence

Using try/catch Blocks in a Program:

```
#include <iostream>

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    try                                         //Line 2
    {
        cout << "Line 3: Enter the dividend: "; //Line 3
        cin >> dividend;                        //Line 4
        cout << endl;                          //Line 5

        cout << "Line 6: Enter the divisor: ";  //Line 6
        cin >> divisor;                        //Line 7
        cout << endl;                          //Line 8

        if (divisor == 0)                     //Line 9
            throw 0;                          //Line 10

        quotient = dividend / divisor;        //Line 11

        cout << "Line 12: Quotient = " << quotient //Line 12
            << endl;

    }
    catch (int)                               //Line 13
```

Using try/catch Blocks in a Program:

```

    {
        cout << "Line 14: Division by 0." << endl;    //Line 14
    }

    return 0;                                         //Line 15
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 17

Line 6: Enter the divisor: 8

Line 12: Quotient = 2

Sample Run 2: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 34

Line 6: Enter the divisor: 0

Line 14: Division by 0.

```

#include <iostream>

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    try                                         //Line 2
    {
        cout << "Line 3: Enter the dividend: "; //Line 3
        cin >> dividend;                       //Line 4
        cout << endl;                          //Line 5

        cout << "Line 6: Enter the divisor: ";  //Line 6
        cin >> divisor;                         //Line 7
        cout << endl;                          //Line 8

        if (divisor == 0)                     //Line 9
            throw divisor;                    //Line 10

        quotient = dividend / divisor;        //Line 11

        cout << "Line 12: Quotient = " << quotient
              << endl;                        //Line 12
    }
}

```

```
    catch (int x)                                     //Line 13
    {
        cout << "Line 14: Division by " << x
              << endl;                                //Line 14
    }

    return 0;                                         //Line 15
}
```

Sample Run 1: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 14

Line 6: Enter the divisor: 5

Line 12: Quotient = 2

Sample Run 2: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 23

Line 6: Enter the divisor: 0

Line 14: Division by 0

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int dividend, divisor = 1, quotient;           //Line 1

    string inpStr
        = "The input stream is in the fail state."; //Line 2

    try                                             //Line 3
    {
        cout << "Line 4: Enter the dividend: ";   //Line 4
        cin >> dividend;                           //Line 5
        cout << endl;                             //Line 6

        cout << "Line 7: Enter the divisor: ";     //Line 7
        cin >> divisor;                             //Line 8
        cout << endl;                             //Line 9
    }
```

```

    if (divisor == 0)                                //Line 10
        throw divisor;                               //Line 11
    else if (divisor < 0)                             //Line 12
        throw string("Negative divisor.");           //Line 13
    else if (!cin)                                    //Line 14
        throw inpStr;                                //Line 15

    quotient = dividend / divisor;                    //Line 16

    cout << "Line 17: Quotient = " << quotient
          << endl;                                    //Line 17
}
catch (int x)                                         //Line 18
{
    cout << "Line 19: Division by " << x
          << endl;                                    //Line 19
}
catch (string s)                                     //Line 20
{
    cout << "Line 21: " << s << endl;                //Line 21
}

return 0;                                            //Line 22
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 4: Enter the dividend: 23

Line 7: Enter the divisor: 6

Line 17: Quotient = 3

Sample Run 2: In this sample run, the user input is shaded.

Line 4: Enter the dividend: 34

Line 7: Enter the divisor: -6

Line 21: Negative divisor.

Sample Run 3: In this sample run, the user input is shaded.

Line 4: Enter the dividend: 34

Line 7: Enter the divisor: g

Line 21: The input stream is in the fail state.

Using C++ Exception Classes

- C++ provides support to handle exceptions via hierarchy of classes
- The function *what* returns the string containing exception object thrown by C++'s built-in exception classes
- The `class exception` is:
 - The base class of the exception classes provided by C++
 - Contained in the header file `exception`

Using C++ Exception Classes

- Two classes derived from `exception`:
 - `logic_error`
 - `runtime_error`
- `logic_error` **and** `runtime_error` are defined in header file `stdexcept`
- The `class` `invalid_argument` deals with illegal arguments used in a function call

Using C++ Exception Classes

- The `class` `out_of_range` deals with the `string subscript out_of_range` error
- The `class` `length_error` handles the error if
 - A length greater than the maximum allowed for a `string` object is used

Using C++ Exception Classes

- If the operator `new` cannot allocate memory space
 - It throws a `bad_alloc` exception
- The `class runtime_error` deals with errors that occur only during program execution
- Classes `overflow_error` and `underflow_error`
 - Deal with arithmetic overflow and under-flow exceptions
 - Derived from the `class runtime_error`

exceptions out_of_range and length_error

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string sentence;                                //Line 1
    string str1, str2, str3;                          //Line 2

    try                                              //Line 3
    {
        sentence = "Testing string exceptions!";      //Line 4
        cout << "Line 5: sentence = " << sentence
              << endl;                                //Line 5
        cout << "Line 6: sentence.length() = "
              << static_cast<int>(sentence.length())
              << endl;                                //Line 6
    }
```

```

    str1 = sentence.substr(8, 20);           //Line 7
    cout << "Line 8: str1 = " << str1
          << endl;                          //Line 8

    str2 = sentence.substr(28, 10);         //Line 9
    cout << "Line 10: str2 = " << str2
          << endl;                          //Line 10

    str3 = "Exception handling. " + sentence; //Line 11
    cout << "Line 12: str3 = " << str3
          << endl;                          //Line 12

}
catch (out_of_range re)                    //Line 13
{
    cout << "Line 14: In the out_of_range "
          << "catch block: " << re.what()
          << endl;                          //Line 14
}
catch (length_error le)                   //Line 15
{
    cout << "Line 16: In the length_error "
          << "catch block: " << le.what()
          << endl;                          //Line 16
}

return 0;                                 //Line 17
}

```



Sample Run:

```

Line 5: sentence = Testing string exceptions!
Line 6: sentence.length() = 26
Line 8: str1 = string exceptions!
Line 14: In the out_of_range catch block: invalid string position

```

exception bad_alloc

```
#include <iostream>

using namespace std;

int main()
{
    int *list[100];                                //Line 1

    try                                             //Line 2
    {
        for (int i = 0; i < 100; i++)              //Line 3
        {
            list[i] = new int[500000000];          //Line 4
            cout << "Line 4: Created list[" << i
                  << "] of 500000000 components."
                  << endl;                          //Line 5
        }
    }
    catch (bad_alloc be)                          //Line 6
    {
        cout << "Line 7: In the bad_alloc catch "
              << "block: " << be.what() << "."
              << endl;                              //Line 7
    }

    return 0;                                     //Line 8
}
```

exception `bad_alloc`

Sample Run:

```
Line 4: Created list[0] of 50000000 components.  
Line 4: Created list[1] of 50000000 components.  
Line 4: Created list[2] of 50000000 components.  
Line 4: Created list[3] of 50000000 components.  
Line 4: Created list[4] of 50000000 components.  
Line 4: Created list[5] of 50000000 components.  
Line 4: Created list[6] of 50000000 components.  
Line 4: Created list[7] of 50000000 components.  
Line 7: In the bad_alloc catch block: bad allocation.
```


Creating Your Own Exception Classes



- Programmers can create exception classes to handle exceptions not covered by C++'s exception classes and their own exceptions
- C++ uses the same mechanism to process the exceptions you define as for built-in exceptions
- You must throw your own exceptions using the `throw` statement
- Any `class` can be an exception class

Creating Your Own Exception Classes

```
class dummyExceptionClass
{
};

#include <iostream>

using namespace std;

class divByZero
{};

int main()
{
    int dividend, divisor, quotient;                //Line 1
```

Creating Your Own Exception Classes

```

try                                     //Line 2
{
    cout << "Line 3: Enter the dividend: "; //Line 3
    cin >> dividend;                       //Line 4
    cout << endl;                          //Line 5

    cout << "Line 6: Enter the divisor: ";  //Line 6
    cin >> divisor;                        //Line 7
    cout << endl;                          //Line 8

    if (divisor == 0)                    //Line 9
        throw divByZero();              //Line 10

    quotient = dividend / divisor;       //Line 11
    cout << "Line 12: Quotient = " << quotient //Line 12
        << endl;

}
catch (divByZero)                       //Line 13
{
    cout << "Line 14: Division by zero!"    //Line 14
        << endl;

}

return 0;                               //Line 15
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 34

Line 6: Enter the divisor: 5

Line 12: Quotient = 6

Sample Run 2: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 56

Line 6: Enter the divisor: 0

Line 14: Division by zero!



```
#include <iostream>
#include <string>

using namespace std;

class divisionByZero                                //Line 1
{                                                    //Line 2
public:                                              //Line 3
    divisionByZero()                                //Line 4
    {
        message = "Division by zero";              //Line 5
    }                                                //Line 6

    divisionByZero(string str)                      //Line 7
    {                                                //Line 8
        message = str;                             //Line 9
    }                                                //Line 10

    string what()                                   //Line 11
    {                                                //Line 12
        return message;                            //Line 13
    }                                                //Line 14

private:                                           //Line 15
    string message;                                //Line 16
};                                                  //Line 17
```

```

#include <iostream>
#include "divisionByZero.h"

using namespace std;

int main()
{
    int dividend, divisor, quotient;           //Line 1

    try                                         //Line 2
    {
        cout << "Line 3: Enter the dividend: "; //Line 3
        cin >> dividend;                       //Line 4
        cout << endl;                         //Line 5

        cout << "Line 6: Enter the divisor: "; //Line 6
        cin >> divisor;                       //Line 7
        cout << endl;                         //Line 8

        if (divisor == 0)                     //Line 9
            throw divisionByZero();           //Line 10

        quotient = dividend / divisor;         //Line 11
        cout << "Line 12: Quotient = " << quotient //Line 12
            << endl;

    }
}

```

```

catch (divisionByZero divByZeroObj)           //Line 13
{
    cout << "Line 14: In the divisionByZero "
        << "catch block: "
        << divByZeroObj.what() << endl;       //Line 14
}

return 0;                                       //Line 15
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 34

Line 6: Enter the divisor: 5

Line 12: Quotient = 6

Sample Run 2: In this sample run, the user input is shaded.

Line 3: Enter the dividend: 56

Line 6: Enter the divisor: 0

Line 14: In the divisionByZero catch block: Division by zero

```
#include <iostream>
#include "divisionByZero.h"

using namespace std;

void doDivision();

int main()
{
    doDivision(); //Line 1

    return 0; //Line 2
}

void doDivision()
{
    int dividend, divisor, quotient; //Line 3

    try
    {
        cout << "Line 4: Enter the dividend: "; //Line 4
        cin >> dividend; //Line 5
        cout << endl; //Line 6
    }
}
```

```

        cout << "Line 7: Enter the divisor: ";           //Line 7
        cin >> divisor;                                   //Line 8
        cout << endl;                                     //Line 9

        if (divisor == 0)                                 //Line 10
            throw divisionByZero();                       //Line 11

        quotient = dividend / divisor;                   //Line 12
        cout << "Line 13: Quotient = " << quotient
              << endl;                                    //Line 13
    }
    catch (divisionByZero divByZeroObj)                 //Line 14
    {
        cout << "Line 15: In the function "
              << "doDivision: "
              << divByZeroObj.what() << endl;           //Line 15
    }
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 4: Enter the dividend: 34

Line 7: Enter the divisor: 5

Line 13: Quotient = 6

Sample Run 2: In this sample run, the user input is shaded.

Line 4: Enter the dividend: 56

Line 7: Enter the divisor: 0

Line 15: In the function doDivision: Division by zero

Rethrowing and Throwing

- When an exception occurs in a `try` block
 - Control immediately passes to one of the catch blocks
- A `catch` block either
 - Handles the exception or partially processes the exception and then rethrows the same exception OR
 - Rethrows another exception for the calling environment to handle

Rethrowing and Throwing

- The general syntax to rethrow an exception caught by a catch block is:

```
throw;
```

(in this case, the same exception is rethrown) or:

```
throw expression;
```

where `expression` is a constant value, variable, or object

Rethrowing and Throwing

- The object being thrown can be
 - A specific object
 - An anonymous object
- A function specifies the exceptions it throws in its heading using the throw clause

Rethrowing and Throwing

- A function specifies the exceptions it throws (to be handled somewhere) in its heading using the throw clause.
- For example, the following function specifies that it throws exceptions of type `int`, `string`, and `divisionByZero`, where `divisionByZero` is the `class` as defined previously.

```
void exmpThrowExcep(int x) throw (int, string, divisionByZero)
{
    *
    *
    *
    //include the appropriate throw statements
    *
    *
    *
}
```


```
#include <iostream>
#include "divisionByZero.h"

using namespace std;

void doDivision() throw (divisionByZero);

int main()
{
    try //Line 1
    {
        doDivision(); //Line 2
    }
    catch (divisionByZero divByZeroObj) //Line 3
    {
        cout << "Line 4: In main: "
              << divByZeroObj.what() << endl; //Line 4
    }

    return 0; //Line 5
}
```



```

void doDivision() throw (divisionByZero)
{
    int dividend, divisor, quotient;                                //Line 6

    try                                                            //Line 7
    {
        cout << "Line 8: Enter the dividend: ";                  //Line 8
        cin >> dividend;                                          //Line 9
        cout << endl;                                           //Line 10

        cout << "Line 11: Enter the divisor: ";                  //Line 11
        cin >> divisor;                                          //Line 12
        cout << endl;                                           //Line 13

        if (divisor == 0)                                         //Line 14
            throw divisionByZero("Found division by 0!");      //Line 15

        quotient = dividend / divisor;                            //Line 16
        cout << "Line 17: Quotient = " << quotient              //Line 17
            << endl;
    }
    catch (divisionByZero)                                       //Line 18
    {
        throw;                                                    //Line 19
    }
}

```

Sample Run 1: In this sample run, the user input is shaded.

Line 8: Enter the dividend: 34

Line 11: Enter the divisor: 5

Line 17: Quotient = 6

Sample Run 2: In this sample run, the user input is shaded.

Line 8: Enter the dividend: 56

Line 11: Enter the divisor: 0

Line 4: In main: Found division by 0!

Exception Handling Techniques

- When an exception occurs, the programmer usually has three choices:
 - Terminate the program
 - Include code to recover from the exception
 - Log the error and continue

Terminate the Program

- In some cases, it is best to let the program terminate when an exception occurs
- For example, if the input file does not exist when the program executes
 - There is no point in continuing with the program
- The program can output an appropriate error message and terminate

Fix the Error and Continue

- In some cases, you will want to handle the exception and let the program continue
- For example, if a user inputs a letter in place of a number
 - The input stream will enter the fail state
- You can include the necessary code to keep prompting the user to input a number until the entry is valid

Log the Error and Continue

- For example, if your program is designed to run a nuclear reactor or continuously monitor a satellite
 - It cannot be terminated if an exception occurs
- When an exception occurs
 - The program should write the exception into a file and continue to run

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int number; //Line 1
    bool done = false; //Line 2

    string str =
        "The input stream is in the fail state."; //Line 3

    do //Line 4
    { //Line 5
        try //Line 6
        { //Line 7
            cout << "Line 8: Enter an integer: "; //Line 8
            cin >> number; //Line 9
            cout << endl; //Line 10

            if (!cin) //Line 11
                throw str; //Line 12

            done = true; //Line 13
            cout << "Line 14: Number = " << number
                << endl; //Line 14
        } //Line 15
        catch (string messageStr) //Line 16
        { //Line 17
            cout << "Line 18: " << messageStr
                << endl; //Line 18
            cout << "Line 19: Restoring the "
                << "input stream." << endl; //Line 19
            cin.clear(); //Line 20
            cin.ignore(100, '\n'); //Line 21
        } //Line 22
    }
    while (!done); //Line 23

    return 0; //Line 24
}
```

Sample Run: In this sample run, the user input is shaded.

Line 8: Enter an integer: **r5**

Line 18: The input stream is in the fail state.

Line 19: Restoring the input stream.

Line 8: Enter an integer: **d45**

Line 18: The input stream is in the fail state.

Line 19: Restoring the input stream.

Line 8: Enter an integer: **hw3**

Line 18: The input stream is in the fail state.

Line 19: Restoring the input stream.

Line 8: Enter an integer: **48**

Line 14: Number = 48

The End!

