COEN 244 – PROGRAMMING METHODOLOGY II

Exception Handling

Introduction

- Exception
  - Undesirable event detectable during program execution
- Code to handle exceptions depends on the type of application being developed
  - May or may not want the program to terminate when an exception occurs
- Can add exception-handling code at a point where an error can occur

Introduction (cont.)

- Handling exceptions within a program
  - Assert function
    - 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

C++ Mechanisms of Exception Handling

- `try`/`catch` block: used to handle 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
- `try` block is followed by one or more `catch` blocks

```
try
{
   // statements
}
⋮
catch (type exception)
{
   // exception-handling code
}
⋮
```

`try/catch` Block (cont.)

- `catch` block
  - Specifies the type of exception it can catch
  - Contains an exception handler
- If the heading of a `catch` block has an ellipsis (three dots) in place of parameters
  - Block can catch exceptions of all types (e.g. int, double, object)
- If no exception is thrown in a `try` block
  - All `catch` blocks are ignored
  - Execution resumes after the last `catch` block

`try`/`catch` Block (cont.)

- If an exception is thrown in a `try` block
  - Remaining statements (in `try` block) are ignored
- A `catch` block can have at most one `catch` block parameter
  - `catch` block parameter becomes a placeholder for the value thrown (i.e., thrown value stored in the `catch` block parameter)

`try/catch` Block (cont.)

- Program searches `catch` blocks in the order they appear after the `try` block and looks 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

Throwing an Exception

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

```
throw expression;
```

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

- Object being thrown can be a specific object or an anonymous object

Throwing an Exception (cont.)

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

int main ()  {
      int a, b, c;
      try {
            cout << “Enter first number:” << a ;
            cin >> a;
            cout << “Enter second number:” << b ;
            cin >> b;
            if (b == 0)
                  throw “Divide by zero exception";   // This forces the program to go to “catch”
            c = a/b;
            cout  << c << endl;
      } catch (char * str)   {
            cout << "Exception: " << str << endl;
      }
      return 0;
}
```

Program execution

Exception: Divide by zero exception

Throwing an Exception (cont.)

- In the previous example, if b equals 0 an exception is thrown
  - When **throw** is executed, the remaining statements in that **try** block are ignored and every object created within that **try** block is destroyed
  - Then, the control is passed to the corresponding **catch** block
  - The program execution resumes after the **catch** block (in this case: **return 0;)**

Multiple Catch Blocks

```
#include <iostream>
#include “account.h”
using namespace std;

int main ()  {
      double amount;
      try {
            Account *a = new Account(1000); // Creates account object with balance 1000
            if (a == nullptr)
                  throw “Allocation failed”;
             cout << “Enter Amount to deposit (>0)”;
             cin >> amount;
             if (amount < 0)
                  throw amount;
            a->deposit(amount);
            // rest of program
      } catch (char * str)   {
             cout << "Exception: " << str << endl;
      } catch (double x)  {
             cout << “Amount:” <<  x << " should be > 0" << endl;
      }
      return 0;
}
```

Multiple Catch Blocks (cont.)

- We can also define a **catch** block that captures all the exceptions independent of the type used in the call to **throw**
  - In this case, we have to write an ellipsis (three dots) in place of parameters in the heading of a `catch` block

```
try {
   // statements
}
catch (...)
{
    cout << "Exception occurred";
}
```

Order of `catch` Blocks

- `catch` block can catch:
  - All exceptions of a specific type
  - All types of exceptions
- A `catch` block with an ellipsis (. . .) catches any type of exception
  - If used, it should be the last `catch` block of that sequence
- Be careful about the order in which you list `catch` blocks

Functions Throwing Exceptions

- A function can throw an exception that will be caught by a try block that calls this function

```
void test(int a)
{
        if (a >= 10)  throw a;
}

int main()
{
    int a;

    try  {

        cout << “Enter a (< 10): \n”;  cin >> a;
        test(a);
        cout << a << “ is less than 10\n”;

    } catch (int i) {
        cout << a << “ is greater than or equal to 10\n”;
    }
    return 0;
}
```

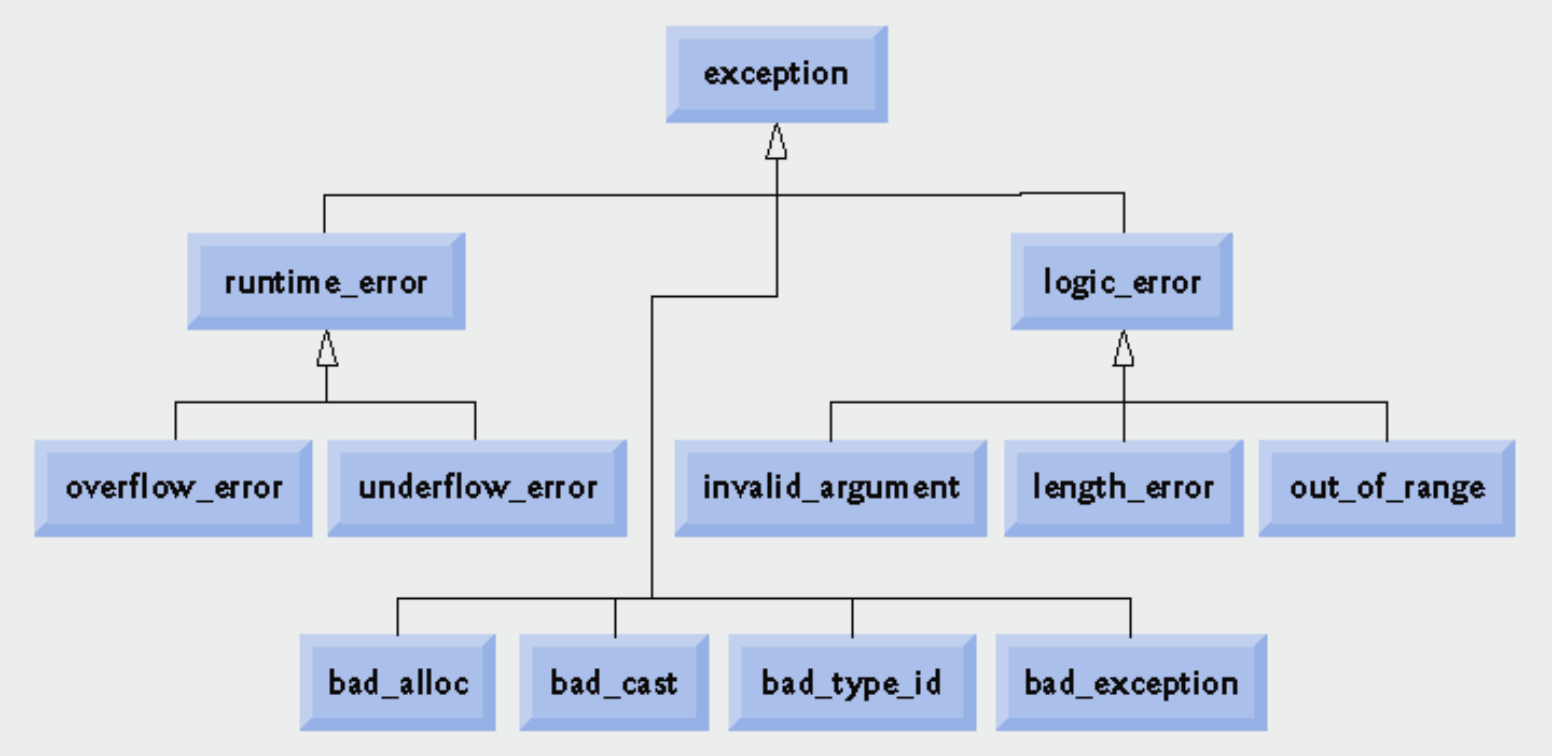

Fig. 16.11 | Standard Library exception classes.

Using C++ Exception Classes

- C++ provides support to handle exceptions via a hierarchy of classes
- `class exception:` base class of the exception classes provided by C++
  - Contained in the header file `exception`
- `what` function
  - Included in the class `exception`
  - Returns a string containing the exception object thrown by C++ built-in exception classes
  - All derived classes of the class `exception` override the function `what` to issue their own error messages

Using C++ Exception Classes (cont.)

- Two classes are immediately derived from the class `exception` (defined in the header file `stdexcept`)
  - `logic_error:` includes several derived classes
    - `invalid_argument`: for use when illegal arguments are used in a function call
    - `out_of_range`: string subscript out of range error
    - `length_error`: if a length greater than the maximum allowed for a string object is used
  - `runtime_error:` includes several derived classes
    - `overflow_error` and `underflow_error`

Using C++ Exception Classes (cont.)

- Some functions of the C++ standard library throw exceptions automatically that can be captured if we include them within a `try` block
  - The list of C++ predefined exceptions is very short and can easily be found on the web
- The following code shows how to handle exceptions `out_of_range` and `length_error`
  - Exceptions are thrown by the string function `substr` and the string concatenation operator +
  - No need to include any throw statement in the `try` block

Using C++ Exception Classes (cont.)

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

int main()
{
   string sentence;
   string str1, str2, str3;
   try
   {
      sentence = "Testing string exceptions!";
      cout << "sentence = " << sentence
           << endl;
      cout << "sentence.length() = "
           << sentence.length()
           << endl;

      str1 = sentence.substr(8, 18);
      cout << "str1 = " << str1 << endl;

      str2 = sentence.substr(28, 10);
      cout << "str2 = " << str2 << endl;

      str3 = "Exception handling. " + sentence;
      cout << "str3 = " << str3 << endl;
   }
```

Using C++ Exception Classes (cont.)

```
    catch (out_of_range& re)
    {
        cout << "In the out_of_range catch"
             << " block: " << re.what() << endl;
    }
    catch (length_error& le)
    {
        cout << "In the length_error catch"
             << " block: " << le.what() << endl;
    }

    return 0;
}
```

Program execution

```
sentence = Testing string exceptions!
sentence.length() = 26
str1 = string exceptions!
In the out_of_range catch block: invalid string position
```

Using C++ Exception Classes (cont.)

- The following code shows how to handle the exception `bad_alloc` due to bad memory allocation and thrown by the operator `new`

```
#include <iostream>
using namespace std;
#include <new>              // standard operator new
using std::bad_alloc;       // type of object thrown by the allocation functions to report failure to allocate storage

int main()
{
  int *p;
  try  {
     p = new int[32];
  }
  catch (bad_alloc &xa) {
     cout << "Allocation failed: " << xa.what() << endl;
  }
  return 0;
}
```

Output:
Allocation failed: std::bad_alloc

Creating Your Own Exception Classes

- Can create your own exception classes to handle specific exceptions not covered by C++ exception classes
  - C++ uses the same mechanism to process these exceptions
- `throw` statement
  - Must be used to throw your own exceptions
- In C++, any `class` can be an exception class
- Exception class with members typically includes:
  - Constructors
  - The function `what`

- Inheritance with exception classes
  - New exception classes can be defined to inherit from existing exception classes
  - A catch handler for a particular exception class can also catch exceptions of classes derived from that class
  - Placing a `catch` handler that catches a base-class object before a `catch` that catches an object of a class derived from that base class is a logic error. The base-class `catch` catches all objects of classes derived from that base class, so the derived-class `catch` will never execute.

Creating Your Own Exception Classes (cont.)

- The following example is taken from the book of Deitel and Deitel
  - It shows how to create a derived class of the class `exception`

```
// A simple exception-handling example that checks
// for divide-by-zero exceptions
#include <iostream>
using namespace std;

#include <exception>
using std::exception;

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public exception
{
   public:
      // constructor specifies default error message
      DivideByZeroException::DivideByZeroException() : exception( "attempted
      to divide by zero" ) {}
};  // end class DivideByZeroException
```

```
// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs

// user-defined function quotient to manipulate the data
double quotient( int num, int den )
{
   if ( den == 0 )
      throw DivideByZeroException();  // terminate function
return num / den;
}

int main()
{
     int number1;      // user-specified numerator
     int number2;      // user-specified denominator
     double result;    // result of division
     cout << "Enter two integers: ";
     // enable user to enter two integers to divide
     cin >> number1 >> number2;
```

```
    // try block contains code that might throw exception
    // and code that should not execute if an exception occurs
    try {
       result = quotient(number1, number2);
       cout << "The quotient is: " << result << endl;
    }
    catch ( DivideByZeroException& divideByZeroException ) {
       cout << "Exception occurred: " <<
          divideByZeroException.what() << endl;
    }

    return 0;
}
```

Program execution

Enter two integers: 14 0
Exception occurred: attempted to divide by zero

Creating Your Own Exception Classes (cont.)

- The preceding program works as follows:
  - The program read two integers number1 and number2
  - It computes the quotient by calling the function quotient(number1, number2)
  - If number2 is equal to 0, an exception is thrown
    - That is, an object of the class **DivideByZeroException** is created and the throw statement throws the object (i.e., exception)
    - The parameter **divideByZeroException** in the catch block catches the value of the thrown object and then uses the what() function to return the information stored in the object
  - Most of the exceptions that you will create will look like this one

Rethrowing and Throwing an Exception

- When an exception occurs in a `try` block, control immediately passes to one of the `catch` blocks, which
  - Handles the exception, or partially processes the exception, then rethrows the same exception
  - Or, rethrows another exception for the calling environment or block to handle the exception
- This allows you to provide exception-handling code all in one place

Rethrowing and Throwing an Exception (cont.)

- Rethrow an exception caught by a `catch` block
  - If the same exception is to be rethrown:
    ```
    throw;
    ```
  - If a different exception is to be thrown
    ```
    throw expression;
    ```
    where `expression` is a constant value, variable, or object
- 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

```
double quotient(int num, int den)
  throw(DivideByZeroException());
```

Exception-Handling Techniques

- Usually, three choices are available to the programmer when an exception occurs
  - Terminate the program
  - Include code to recover from the exception
  - Log the error and continue

Terminate the Program

- In some cases, it is better to terminate the program when an exception occurs
- For example, if an input file does not exist when the program executes
  - There is no point in continuing with the program
  - 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 instead of a number
  - The input stream will enter the fail state
  - 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 the 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

Stack Unwinding

- When an exception is thrown in a function, the function can do the following:
  - Do nothing
  - Partially process the exception and throw the same exception or a new exception
  - Throw a new exception
- In each case, the function-call stack is unwound so that the exception can be caught in the next `try/catch` block

Stack Unwinding (cont.)

- When the function-call stack is unwound
  - The function in which the exception was not caught and/or rethrown terminates
  - Memory for its local variables is destroyed
    - C++ run time calls destructors for all automatic objects constructed since the beginning of the `try` block
- Stack unwinding continues until
  - A `try`/`catch` handles the exception, *or*
  - The program does not handle the exception
    - The function `terminate` is called to terminate the program