Exception Handling in C++

Professor Hugh C. Lauer CS-2303, System Programming Concepts

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie, *Absolute C++*, by Walter Savitch, *The C++ Programming Language*, Special Edition, by Bjarne Stroustrup, and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

Outline

- What exceptions are and when to use them
- Using try, catch and throw to detect, handle and indicate exceptions, respectively
- To process uncaught and unexpected exceptions
- To declare new exception classes
- How stack unwinding enables exceptions not caught in one scope to be caught in another scope
- To handle new failures
- To understand the standard exception hierarchy



Introduction

Exceptions

- Indicate problems that occur during a program's execution
- Occur infrequently

Exception handling

- Can resolve exceptions
 - Allow a program to continue executing or
 - Notify the user of the problem and
 - Terminate the program in a controlled manner
- Makes programs robust and fault-tolerant

Exception Handling in C++

A standard mechanism for processing errors

 Especially important when working on a project with a large team of programmers

C++ exception handling is much like Java's

Java's exception handling is much like C++

Fundamental Philosophy

- Mechanism for sending an exception signal up the call stack
 - Regardless of intervening calls
- Note: there is a mechanism based on same philosophy in C
 - setjmp(), longjmp()
 - See man pages

Traditional Exception Handling

Intermixing program and error-handling logic

- Pseudocode outline
 - Perform a task
 If the preceding task did not execute correctly Perform error processing
 Perform next task
 If the preceding task did not execute correctly Perform error processing
- Makes the program difficult to read, modify, maintain and debug
 Note:- In most large system
- Impacts performance

...

Note:- In most large systems, code to handle errors and exceptions represents >80% of the total code of the system

- Remove error-handling code from the program execution's "main line"
- Programmers can handle any exceptions they choose
 - All exceptions
 - All exceptions of a certain type
 - All exceptions of a group of related types

Programs can

- Recover from exceptions
- Hide exceptions
- Pass exceptions up the "chain of command"
- Ignore certain exceptions and let someone else handle them

An exception is a class

- Usually derived from one of the system's exception base classes
- If an exceptional or error situation occurs, program throws an object of that class
 - Object crawls up the call stack
- A calling program can choose to *catch* exceptions of certain classes
 - Take action based on the exception object

Class exception

- The standard C++ base class for all exceptions
- Provides derived classes with virtual function what()
 - Returns the exception's stored error message

Example — Divide by Zero

- 1 // Fig. 27.1: DivideByZeroException.h
- 2 // Class DivideByZeroException definition.
- 3 #include <stdexcept> // stdexcept header file contains runtime_error
- 4 using std::runtime_error; // standard C++ library class runtime_error

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** Di vi deByZeroExcepti on: : Di vi deByZeroExcepti on()
- 13 : runtime_error("attempted to divide by zero") {}
- 14 }; // end class DivideByZeroException

```
Worcester Polytechnic Institute
1 // Fig. 27.2: Fig27_02.cpp
                                                                                example
2 // A simple exception-handling example that checks for
3 // divide-by-zero exceptions.
                                                                                 – (1 of 2)
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
  #include "DivideByZeroException. h" // DivideByZeroException class
9
10
11 // perform division and throw DivideByZeroException object if
12 // divide-by-zero exception occurs
13 double quotient( int numerator, int denominator )
14 {
      // throw DivideByZeroException if trying to divide by zero
15
16
     if ( denominator == 0 )
         throw DivideByZeroException(); // terminate function
17
18
     // return division result
19
     return static_cast< double >( numerator ) / denominator;
20
21 } // end function quotient
22
23 int main()
24 {
25
     int number1; // user-specified numerator
26
     int number2; // user-specified denominator
27
      double result; // result of division
28
      cout << "Enter two integers (end-of-file to end): ";</pre>
29
```

```
Worcester Polytechnic Institute
                                                                                    example
                                                                                     - (2 of 2)
30
31
      // enable user to enter two integers to divide
32
      while ( cin >> number1 >> number2 )
33
      {
34
         // try block contains code that might throw exception
35
         // and code that should not execute if an exception occurs
36
         try
37
         {
38
            result = quotient( number1, number2 );
39
            cout << "The quotient is: " << result << endl;</pre>
40
         } // end try
41
42
         // exception handler handles a divide-by-zero exception
43
         catch ( DivideByZeroException & divideByZeroException )
44
         {
45
            cout << "Exception occurred: "</pre>
               << di vi deByZeroExcepti on. what() << endl;</pre>
46
47
         } // end catch
48
49
         cout << "\nEnter two integers (end-of-file to end): ";</pre>
50
      } // end while
51
52
      cout << endl;</pre>
53
      return 0; // terminate normally
54 } // end main
```

Questions?

try Blocks

Keyword try followed by curly braces "{}"

Should enclose

- Statements that might cause exceptions
- Statements that should be skipped in case of an exception

Software Engineering Observation

Exceptions may surface

- through explicitly mentioned code in a try block,
- through calls to other functions and
- through deeply nested function calls initiated by code in a *try* block.

Catch Handlers

Immediately follow a try block

- One or more *catch* handlers for each *try* block
- Keyword catch
- Exception parameter enclosed in parentheses
 - Represents the type of exception to process
 - Can provide an optional parameter name to interact with the caught exception object

Executes if exception parameter type matches the exception thrown in the *try* block

Could be a base class of the thrown exception's class

Catch Handlers (continued)

```
try {
                                          Other classes of exceptions
  // code to try
                                          are not handled here
catch (exceptionClass1 &name1) {
  // handle exceptions of exceptionClass1
catch (exceptionClass2 &name2) {
  // handle exceptions of exceptionClass2
catch (exceptionClass3 &name3) {
                                           catch clauses attempted
  // handle exceptions of exceptionClass3
                                           in order; first match wins!
/* code to execute if no exception or
  if catch handler successfully handled exception*/
```

CS-2303, A-Term 2012

Common Programming Errors

Syntax error to place code between a *try* block and its corresponding *catch* handlers

Each *catch* handler can have only a single parameter

 Specifying a comma-separated list of exception parameters is a syntax error

Logic error to catch same type in two different catch handlers following a single try block

Termination model of exception handling

- try block expires when an exception occurs
 - Local variables in try block go out of scope
- Code within the matching catch handler executes
- Control resumes with the first statement after the last catch handler following the try block

Control *does not* return to throw point

Stack unwinding

- Occurs if no matching catch handler is found
- Program attempts to locate another enclosing *try* block in the calling function

Stack "Unwinding"

- Occurs when a thrown exception is not caught in a particular scope
- Unwinding a Function terminates that function
 - All local variables of the function are destroyed
 - Invokes destructors
 - Control returns to point where function was invoked
- Attempts are made to catch the exception in outer *try...catch* blocks
- If the exception is never caught, the function terminate() is called

Observations

- With exception handling, program can continue executing after dealing with a problem
 - rather than terminating
- Helps to support robust applications that contribute to mission-critical computing or business-critical computing
- When no exceptions occur, there is no performance penalty

Throwing an Exception

- Use keyword throw followed by an operand representing the type of exception
 - The throw operand can be of any type
 - If the *throw* operand is an object, it is called an *exception* object
- The throw operand initializes the exception parameter in the matching catch handler, if one is found

Notes

Catching an exception object by reference eliminates the overhead of copying the object that represents the *thrown* exception

Associating each type of runtime error with an appropriately named exception object improves program clarity.

When to Use Exception Handling

To process *synchronous* errors

Occur when a statement executes string

Not to process asynchronous errors

- Occur concurrently with, and independent of, program execution
- To process problems arising in predefined software elements
 - Such as predefined functions and classes
 - Error handling can be performed by customized program code based on the application needs

Don't use for routine stuff such as end-of-file or null string checking

Software Engineering Notes

- Incorporate exception-handling strategy into system design from the start
 - Very difficult to retrofit after the system has been implemented!
- Exception handling provides uniform technique for processing problems
 - Helps with understandability of each other's error handling code
- Avoid using exception handling as an alternate form of flow of control
 - These "additional" exceptions can "get in the way" of genuine error handling

Re-throwing an Exception

- Empty throw; statement
- Use when a *catch* handler cannot or can only partially process an exception
- Next enclosing *try* block attempts to match the exception with one of its *catch* handlers

Common Programming Error

Executing an empty *throw* statement outside a *catch* handler causes a function call to *terminate*

 Abandons exception processing and terminates the program immediately

Exception Specifications

- Also called throw lists
- Keyword throw
 - Comma-separated list of exception classes in parentheses

Example

 int someFunction(double value) throw (ExceptionA, ExceptionB, ExceptionC)

Indicates someFunction can throw types ExceptionA, ExceptionB and ExceptionC



Exception Specifications (continued)

- A function can *throw* only exceptions of types in its specification (or derived types)
 - If a function throws a non-specification exception, function *unexpected* is called
 - This usually terminates the program
- Absence of exception specification indicates that the function can *throw* any exception
- An empty exception specification, throw(), indicates the function cannot throw any exceptions

Error Note

Compiler will not generate compilation error if

- Function contains a *throw* expression for an exception not listed in exception specification
- Error occurs only when that function attempts to throw that exception
 - At run time

To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications

Questions?

Constructors and Destructors

Exceptions and constructors

- Exceptions enable constructors to report errors
 - Unable to return values
- Exceptions thrown by constructors cause any alreadyconstructed component objects to call their destructors
 - Only objects that have already been constructed will be destroyed

Exceptions and destructors

- Destructors are called for all automatic objects in the terminated try block when an exception is thrown
 - Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
- If a destructor invoked by stack unwinding throws an exception, function *terminate()* is called

Note

When an exception is thrown from the constructor for an object that is created in a new expression, ...

 ... the dynamically allocated memory for that object is released.

Exceptions and Inheritance

- 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
 - Enables catching related errors with a concise notation

Failure of calls to new

Some compilers throw a bad_alloc exception

Compliant with the C++ standard specification

Some compilers return 0

- C++ standard-compliant compilers also have a version of *new* that returns *0*
 - Use expression new(nothrow), where nothrow is of type nothrow_t

Some compilers throw bad_alloc if <new> is included

Standard Library exception classes



Standard Library Exception Hierarchy

Base-class exception

Contains virtual function what() for storing error messages

Exception classes derived from *exception*

- bad_alloc thrown by new
- bad_cast thrown by dynamic_cast
- bad_typeid thrown by typeid
- bad_exception thrown by unexpected
 - Instead of terminating the program or calling the function specified by set_unexpected
 - Used only if *bad_exception* is in function's throw list

Exception Handling Summary

- Exceptions are derived from class exception
- Exceptional or error condition is indicated by throwing an object of that class
 - Created by constructor in *throw* statement
- Calling programs can check for exceptions with try...catch construct
- Unified method of handling exceptions
 - Far superior to coding exception handling in long hand
- No performance impact when no exceptions

Exception Handling Summary (continued)

Many more details — see

- Absolute C++, Chapter 18
- Any other textbook
- C++ standard

Questions?