

605.404 Object-Oriented Programming with C++
Summer 2007
Lectures 10

10	Exception Handling	S: Chap 14
----	--------------------	------------

Additional Resources Related to This Lecture:

Booch, G. (1994) *Object-Oriented Analysis and Design With Applications, Second Edition*. Benjamin Cummings.

Coplien, J. (1992). *Advanced C++ Programming Styles and Idioms*. Addison-Wesley.

Fairley, R.E., *Software Engineering Concepts*, McGraw-Hill, New York, NY, 1985.

Meyer, B., *Introduction to the Theory of Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Meyer, B. (1988). *Object-oriented Software Construction*. Prentice Hall.

Sebesta, Robert W. (1993). *Concepts of Programming Languages, Second Edition*. Benjamin-Cummings.

(Remember to refer to notes in Lecture 01 for exception handling models and background)

Chapter 14 Exception Handling

I. 14.1 Error Handling

- A. fundamental setup – a function that finds a problem that it cannot resolve can throw an exception hoping that its caller or a program in the call hierarchy can resolve (handle) the problem. Functions that can resolve problems indicate the exception(s) it is willing to solve
- B. see notes later in this lecture set about Eiffel, Ada, and PL/I exception handling models
- C. important paragraphs top page 357.
- D. 14.1.1 Alternative Views on Exceptions
 1. think about exceptions in terms of acceptable and unacceptable system states
 2. exception handler intended to support error handling in programs (systems) composed of independently developed components (e.g., libraries, other team members' code, reusable components)
 3. exception handling mechanism is designed to handle only synchronous exceptions
 4. exception handling mechanism is a nonlocal control structure based on stack unwinding that can be seen as an alternative return mechanism, but focus is on error handling and fault tolerance
 5. Text will not discuss exception handling for concurrent processes
 6. exception handling mechanisms provided to report and handle errors and exceptional events – you have to decide what is an error or exceptional event.

- II. 14.2 Grouping Exceptions
 - A. effect of a throw is to unwind the stack until a suitable catch is found
 - B. remember that a child can do everything the parent can do and possibly more and that an instance of a child object can be used anywhere the parent is specified..
 - C. 14.2.1 Derived Exceptions
 - 1. an exception is typically caught by a handler for its base class rather than be a handler for its exact class
 - 2. importance of slicing in exception handling and planning of parameters to catch function.
 - D. 14.2.2 Composite Exceptions
 - 1. use of multiple inheritance in exception handlers to group related error handlers that are not in a tree structure.
- III. 14.3 Catching Exceptions
 - A. rules for type match between type thrown and type caught on page 361
 - B. note that the exception is copied when it is thrown so cannot throw an exception that cannot be copied.
 - C. 14.3.1 Re-Throw
 - 1. rethrow indicated by *throw*
 - a) it has no operand
 - b) *terminate()* will be called if a re-throw is attempted when there is no exception to re-throw.
 - D. 14.3.2 Catch Every Exception
 - 1. default with meaning “catch any exception”.
 - 2. 14.3.2.1 Order of Handlers
 - a) handlers are tried in order.
- IV. 14.4 Resource Management
 - A. importance of releasing acquired resources
 - B. usually release in reverse order of acquisition
 - C. use classes with constructors and destructors to ensure exception-safe resource acquisition and release.
 - D. 14.4.1 Using Constructors and Destructors
 - 1. object not considered constructed until its constructor has completed (and all included objects are constructed)
 - 2. must guard against errors not handled well before an object is fully constructed.
 - E. 14.4.2 Auto_ptr
 - 1. supports “resource acquisition is initialization” technique
 - 2. object pointed to will be implicitly deleted at the end of the auto_ptr’s scope (whether or not an exception is thrown)
 - 3. when one auto_ptr is copied to another, the source no longer points to anything
 - 4. a const auto_ptr cannot be copied
 - 5. not compatible with standard library or algorithms.
 - F. 14.4.3 Caveat
 - G. 14.4.4 Exception and New
 - 1. if a class overloads new() you should also overload delete().
 - H. 14.4.5 Resource Exhaustion
 - 1. resumption - supported by function-call mechanisms
 - 2. termination – supported by exception-handling mechanism

3. C++ is required to have enough spare memory to be able to throw `bad_alloc` in case of memory exhaustion.
- I. 14.4.6 Exceptions in Constructor
 1. 14.4.6.1 Exceptions and Member Initialization
 - a) note that the member initializer list can be inside the try block of the constructor.
 2. 14.4.6.2 Exceptions and Copying
 3. 14.4.7 Exceptions in Destructors
 - a) exiting from a destructor by throwing an exception is a violation of the standard library requirements
 - b) may make use of `uncaught_exception()`.
 - V. 14.5 Exceptions that are not errors
 - A. due to efficiency, exceptions should be used only where the more traditional control structures are inelegant or impossible to use.
 - VI. 14.6 Exception Specifications
 - A. making exceptions thrown part of the interface of a method visible to other methods
 - B. meaning of this construct pgs 375-276
 - C. functions declared without an exception specification are assumed to throw every exception
 - D. you can define a function that will throw no exceptions as well.
 - E. 14.6.1 Checking Exception Specifications
 1. if any declaration of a function has an exception-specification then every declaration of that function, including the definition, must have an exception specification with exactly the same set of exception types
 2. a virtual function may be overridden only by a function that has an exception-specification at least as restrictive as its own
 3. a pointer to a function that has a more restrictive exception-specification to a pointer to a function that has a less restrictive exception-specification is allowed.
 - F. 14.6.2 Unexpected Exceptions
 1. calls to `unexpected()` can be intercepted and rendered harmless
 2. a well-defined subsystem Y will often have all its exceptions derived from a class `Yerr`
 3. all exceptions thrown by the standard library are derived from class `exception`.
 - G. 14.6.3 Mapping Exceptions
 1. modification of `unexpected()` can be done to execute something that will not call `terminate()` which is the default for `unexpected()`.
 2. 14.6.3.1 User mapping of exceptions
 - a) map an unexpected exception into an expected one is more flexible than behavior provided by `bad_exception`
 3. 14.6.3.2 Recovering the Type of an exception
 - a) re-throw the exception and catch it to determine the type of the exception thrown
 - b) `clone()` allocates a copy of an exception on free store (so you must remember to de-allocate it later) and survives the exception handler's cleanup of a local variable.
 - VII. 14.7 Uncaught exceptions
 - A. if an exception is thrown and not caught, `std::terminate()` is called
 - B. also called if exception-handling mechanism finds the stack corrupted

- C. also called if a destructor is called during stack unwinding caused by an exception tries to exit using an exception
 - D. example page 381 – note that exceptions thrown by construction and destruction of global variables will not be caught.
- VIII. 14.8 Exceptions and Efficiency.
- IX. 14.9 Error-Handling Alternatives
- A. shoot for multilevel fault-tolerance
 - B. (should use axioms for pre and post-conditions to ensure that no unplanned for states exist at start or end of a module)
 - C. text says separate the program into distinct subsystems that either complete successfully or fail in a well-defined ways is essential, feasible, and economical.
- X. 14.10 Standard Exceptions