Object-Oriented Principles and Practice / C++

Alice E. Fischer

June 17, 2013

Handling Program Errors

Old-old Technique

Better Techniques

Full C-Style Error Handling

The Modern Technique

Old-old Techniques

Years ago, programs might have responded to errors like this:

- Do nothing. Don't check for errors; hope they dont happen. Let the program crash or produce garbage answers if an error happens.
- Use a long-distance goto to return to the program's top level. Any information about the error would have to be stored in global variables before executing the goto. This is like programming in BASIC. Use of this control pattern destroys the modularity that C programs can otherwise achieve. It is not recommended.
- Use assert() to check for errors. This gives no opportunity to print out information about the error or its cause.

Better Techniques

We hope that our intermediate students will do one of these things:

- Identify the error and call an error function. This defers the problem instead of solving it. The error function still must do something about the error.
- Identify the error, print an error comment, and call exit(). (Equivalent to using fatal().) However, aborting execution is not permissible in many real-life situations. For example, aborting execution while processing bank accounts could leave those accounts in an inconsistent or incorrect state.

Full C-Style Error Handling

To handle an error properly, the user must be notified of the cause of the error and, if possible, given a chance to correct it and proceed with execution.

- The function being executed could return with an error code.
- The function that called it would need to check for that error code and return to its caller with an error code, and so on,
- . . . until control returned to the level at which the error could be handled.

This works but distorts the logic of the program and clutters it with a large amount of code that is irrelevant 99% of the time.

Worse, this technique discourages the use of functions and modular code.

The Modern Technique

Modern languages support exception systems.

- If possible, handle the error in function that detects it.
- Otherwise, throw an exception.
- In the exception object, store all possible information about the error and define a print function.
- Define the exception handler in the class and function that is able to correct the problem.
- Call the exception's print function from the handler, then either abort or correct the problem and continue.

Exceptions, Chapter 17

What is an exception?

Standard exceptions

Exception definition syntax

Throwing an exception

Exception handler syntax

Faster than a speeding bullet

When you throw an exception, it takes control

- From the level at which an error is discovered, often deep within the code, at a stage when several functions have been called and have not yet returned ...
- Carrying as much information as necessary about the error ...
- To the level at which the error can be handled, often in the main function or a high-level function that handles the overall progress of the program or operator interaction ...
- And back into normal execution, if that is appropriate.

Exceptions are Objects

The form and behavior of an exception is defined by a class, just like any other object.

- The exception class might have public data members, since it purpose is to carry information OUT OF the class and into another context.
- It needs a constructor and, possibly, a destructor.
- It should have a print function.
- Exceptions are not declared, and using `new` is not necessary to create one. Calling `throw` creates and initializes the exception.

Standard Exceptions

Many exceptions are predefined and used by the C++ system.

- `bad_alloc` is thrown when the system cannot allocate enough memory for a call on `new`. You no longer need to include a test for allocation error after every call on malloc or new.
- `bad_cast` is thrown by `dynamic_cast` when the run-time check fails.
- `bad_typeid` is thrown when `typeid` is applied to a NULL polymorphic pointer.
- `bad_exception` is thrown when a function *has* a declared list of exceptions that it might throw but actually *throws* one that is not on the list.

Standard Exception Base Classes

An exception class can be derived from another exception class.

- `exception` is the base class for standard exceptions.
- `logic_error` is a base class for preventable runtime errors, including `domain_error`, `invalid_argument`, `length_error`, and `out_of_range`. These are designed so that any program, not just the elements of the standard library, can throw them. Several STL classes do.
- `runtime_error` is a base class for exceptions to report errors that can only be detected at runtime, including `range_error`, `overflow_error`, and `underflow_error`.
- `ios_base::failure` is a base class for exceptions thrown by functions in the iostream class hierarchy.

Announcing Exceptions

Unlike Java, a programmer is not required to declare the list of exceptions that a function can throw.

The list of expected exceptions *can be* declared between the end of the function's parameter list and beginning of the function body.:
`void myfunction () throw (Bad, BadCard) { ...`
If this is done, and the function throws an exception that is `not` on the list, a `bad_exception` is thrown instead, and the result will be immediate termination.

A Basic Exception Class

An exception class is just like any other, except all the members can be public.

- The data members store information about the data that caused the exception and abour the context in which the exception occurred.
- The constructor initializes the data members from its parameters.
- There should be no need for dynamic allocation, and therefore no need for a destructor.
- The print function formats this information in a readable form and displays it on an error stream.

Example: Bad Cards

The Bad exception hierarchy is designed to report input errors that occur while inputting the cards for a card game.

- Bad is a base exception class, from which BadSuit and BadCard are derived.
- Bad contains the data members that describe a card (its suit and face value).
- Bad defines a virtual void print(ostream&) function, that will echo the input and print an error comment.
- We throw Bad if both inputs are wrong.
- Because print is virtual, the rules would normally require definition of a virtual destructor. However, all the derived destructors are null default destructors, so it makes no difference whether they are ever called.

Example: Derived Exceptions

With derived exceptions, one exception handler can handle all variations. A virtual function produces a specific error report from a non-specific call.

- BadDate and BadTime are publicly derived from Bad.
- Their constructors need a ctor to call the Bad constructor.
- Both classes define a method for print() that displays a specific comment about the error, then calls the Bad::pr() method to finish the output.
- Define an empty virtual destructor.

Throwing an Exception

When you throw an exception.

- Space for the object is allocated in an area set aside by the runtime system .
- Then exception's constructor is called to initialize the object.
- The exception object is thrown up the line toward main.
- It may be intercepted along the way, modified, and rethrown.
- It may by caught by the intended handler and managed there.
- If not caught, it will cause the program to abort.
- Memory for the core exception object is handled automatically by the system.

Example: Throwing BadDate

Suppose you throw a BadDate exception.

- First, space for the object is allocated in an area set aside by the runtime system and a `BadDate` type-tag is attached to it.
- Then the BadDate constructor is called and it calls the Bad constructor in a ctor.
- The parameters are installed in the exception object.
- Finally, the exception is thrown up the line toward main.

An exception can be caught and rethrown if some function in the chain between the thrower and the catcher needs to add information to the exception that was not available where it was thrown.

Keywords `try` and `catch`

Exceptions are managed using `try ...catch` blocks.

- The `try` block contains the code that might throw exceptions.
- The following `catch` blocks define which exceptions will be caught and how to handle each.
- The order in which the handlers are written is important; A general case must come after all related specific cases.
- After handling the exception, the normal flow of control resumes with the line that follows the last `catch` block.

If an exception comes, and it is not covered by any catcher, it is not caught and it terminates execution.

Catching the Exception

Exceptions are handled by `catch` blocks.

- Each catcher accepts one kind of exception:
  `catch (bad_alloc bs){ ... }`
- If the handler does not use the information in the exception, the parameter name may be omitted.
- The first catcher that matches an exception will handle it.
- A base-class catcher will catch both base-class and derived-class exceptions.
- "`catch( ... )`" means "catch all exceptions".
- "`catch (exception)`" means "catch any exception derived from std::exception".