Bitwise Operator

by Matt Slot

This month I want to discuss probably the most important programming practice I've adopted: internal state validation and error propogation. If you've read the books "Code Complete" and "Writing Solid Code" from Microsoft Press (or work in the same code base as someone who has), you may already be familiar with these concepts -- for you, I'm going to evangelize the techniques and provide some useful snippets that will help you get started.

The primary concept is that every possible compile or run time error should be flagged as soon as possible in the development process. By adding extra sanity checks and wrapping library functions, you wil reduce the number of stupid mistakes in an implementation, identify run-time errors faster, and cut your overall debugging time.

Code Validation

Whenever you write code, you make numerous assumptions about the inputs and run-time environment. Of course, while you're implementing it, the code is crystal clear and you've added plenty of comments: around specific lines, for code blocks, above whole functions, and even in the source or API header file.

But any real project takes months to complete, may have several new programmers, and require several stages of evaluation and redesign. In such an environment, it's quite easy for NULL pointers to propogate down or error codes to get brushed aside over time. This is a frequent source of obscure or hard-to-reproduce errors ("well, it works on my machine").

Generous use of parameter validation and internal validation is a good way to reduce such "evolutionary" problems. For each function, check every parameter that is passed against NULL or valid range of values. Every time a global variable is accessed, validate that it was properly initialized.
For functions that must be called in a specific order, test that the process is performed properly (typically tracking a state variable).

Error Propogation

Another way for problems to crop up are unexpected errors returned from library functions. Most developers quickly write up and unit test a batch of functions to "bootstrap" a specific feature, then go back and implement better error testing when everything works. In such a case, it's easy to disregard a reported error or fail to propogate it to the caller.

It's important to test the result of *every* function which can fail, even those that should simply never fail. For example, printf() can fail due to disk full errors (yes, it does happen). When I was first implementing such tests, I wrapped a call to the MacOS function Dequeue() (which returns an error if the specified element isn't found) to remove the first element. Obviously it should work as long as there is an element in the lsit, but in the end, I saved several hours of debugging time because the error check caught me passing the *address* of the element pointer.

Functions can be grouped into 4 categories. The first kind performs an action which may fail in the course of normal operation (allocating memory, writing to disk); such functions should always inform the caller if it fails. The second kind are functions which never fail (deallocating memory, zeroing a block of memory); these functions are typically declared void.

An abstraction layer is composed of functions which "wrap" another library, remapping parameters and error codes from one range (system-defined errors) into another (application-defined errors). Finally, event handler functions invoke several other functions (which may or may not fail), but have to handle the user's request from start to finish. This means that it anticipates any errors and displays an appropriate message ("couldn't save file, disk is full"), but doesn't pass it up because the problem has been handled.

Writing an Error Library

While it's great to write implement robust error checking and code validation in the source, it's a drag because of the impact on performance (not to mention spurious error messages). For this reason, it's

wise to compile 2 entirely different versions of my application or library, one for debugging (larger
and slower) and one for shipping (smaller and faster).

Now, writing 2 separate implementations is simply a waste of time, so most programmers compile the
code base twice. Using the preprocessor to define DEBUG let's them distinguish between versions, so
that each version is identical save the error code.

In the process of experimenting with rigorous error checking, I implemented several macros to aid
testing and propogation. Because I'm a C programmer who likes a few C++-isms, I actually adopted
terminology and design similar to the throw/catch metaphor.

Like the standard C library, my error library contains an Assert function for performing sanity checking of function parameters and internal state. Because such problems quickly appear during the
implementation and unit testing process, Assert exists only in the DEBUG library and compiles out to
an empty declaration in the non-DEBUG version. However, due to the importance of such problems,
an Assert will force the application to quit immediately (and spur the programmer to fix it on the spot).

Next is the Throw declaration, which aborts execution of the current function by jumping to cleanup
code at the end (using the nasty goto construct). This is useful for functions like saving documents,
where a single failure in the process should simply cancel its execution. In the DEBUG version, throwing an error displays a complete error description before aborting, but unlike the Assert, non-DEBUG versions still perform the test and abortive cleanup (an error saving a file needs to be
handled, even in shipping applications).

While some errors indicate critical problems with program execution, others can be considered "soft
errors". For example, after exchanging some network data an application normally releases the network endpoint, however on some systems the function to do that returns an error code. By wrapping the call with a Trace declaration, the DEBUG version displays an error message without
affecting program flow, but because the effect is generally benign the non-DEBUG application remains
silent.

Finally, in an abstraction layer, the key is simply determining the error code and how that value should
map into the application's own numbering scheme. For this reason, I added the Remap declaration
which works the exactly the same as Throw except that it reports two error values in the DEBUG version -- the old (system) and the new (remapped) code.

Basically, then, we have several sets of functions which can be sprinkled through a source file, which
will perform several types of validation and error propogation in DEBUG mode, but compile into
simplified handlers for non-DEBUG binaries.

To make these 4 types of functions more useful (and more readable), I've implemented variations that
flag NULL pointers, true conditions, or false (zero) conditions. Finally, a Catch declaration is placed
at the end of the function, right before cleanup code, as the target for abortive Throw declarations.

Sample Implementation

I have placed some sample code online, consisting of:

> http://www.AmbrosiaSW.com/~fprefect/bitwise/stddebug.h
>
> The main set of declarations, consisting of macros that wrap a standard Debug() function. These macros record the affected line and file, error code (or codes), a brief error description, and the desired behavior (to assert, throw, or trace).
>
> http://www.AmbrosiaSW.com/~fprefect/bitwise/stddebug.c
> http://www.AmbrosiaSW.com/~fprefect/bitwise/macdebug.c
>
> Each contains an implementation of Debug() appropriate to the platform. The standard file uses printf() to display the information to stderr, and the MacOS version uses DebugStr() to record a message in Macsbug. Feel free to write your own version of these functions, to record to file or display error dialogs -- but keep in mind that an application may call this function repeatedly while propogating an error up the calling stack, or even from software interrupt time.

There are several tricks used in this implementation. First, because it's convenient to simply wrap error-prone functions with the Throw or Trace macro, we have to be careful not to evaluate the conditional twice. For this reason, we declare a temporary error placeholder and use that through the
rest of the declaration.

Given that we needed a temporary variable declaration, we take advantage of a clever C construct. It
executes exactly once, lets us declare a variable within the braces, and even avoids the nested if-else
problem.

```
#define qMyMacro(x) do  long y; if (y=(x)) DoSomething(y);
while(0); if (SimpleFunction()) MyMacro(1); else MyMacro(0);
```

Anyway, you get the basic idea. I hope this inspires you to implement some rigorous error checking,
and to cut your debugging time dramatically. One thing that I'd really like to see is an implementation
that propogates a complete error structure instead of a simple value, much like the actual C++ throw/catch implementation.

Matt Slot, Bitwise Operator