Dead Programs Tell No Lies

Have you noticed that sometimes other people can detect that things aren’t well with you before you’re aware of the problem yourself? It’s the same with other people’s code. If something is starting to go awry with one of our programs, sometimes it is a library routine that catches it first. Maybe a stray pointer has caused us to overwrite a file handle with something meaningless. The next call to read will catch it. Perhaps a buffer overrun has trashed a counter we’re about to use to determine how much memory to allocate. Maybe we’ll get a failure from malloc. A logic error a couple of million instructions ago means that the selector for a case statement is no longer the expected 1, 2, or 3. We’ll hit the default case (which is one reason why each and every case/switch statement needs to have a default clause—we want to know when the “impossible” has happened).

It’s easy to fall into the “it can’t happen” mentality. Most of us have written code that didn’t check that a file closed successfully, or that a trace statement got written as we expected. And all things being equal, it’s likely that we didn’t need to—the code in question wouldn’t fail under any normal conditions. But we’re coding defensively. We’re looking for rogue pointers in other parts of our program trashing the stack. We’re checking that the correct versions of shared libraries were actually loaded.

All errors give you information. You could convince yourself that the error can’t happen, and choose to ignore it. Instead, Pragmatic Programmers tell themselves that if there is an error, something very, very bad has happened.

Tip 32
Crash Early

Crash, Don’t Trash

One of the benefits of detecting problems as soon as you can is that you can crash earlier. And many times, crashing your program is the best thing you can do. The alternative may be to continue, writing corrupted
data to some vital database or commanding the washing machine into its twentieth consecutive spin cycle.

The Java language and libraries have embraced this philosophy. When something unexpected happens within the runtime system, it throws a `RuntimeException`. If not caught, this will percolate up to the top level of the program and cause it to halt, displaying a stack trace.

You can do the same in other languages. If you don’t have an exception mechanism, or if your libraries don’t throw exceptions, then make sure you handle the errors yourself. In C, macros can be very useful for this:

```c
#define CHECK(LINE, EXPECTED)  
{ int rc = LINE;  
  if (rc != EXPECTED)
    ut_abort(__FILE__, __LINE__, #LINE, rc, EXPECTED); } 
void ut_abort(char *file, int ln, char *line, int rc, int exp) {
  fprintf(stderr, "%s line %d\n%s: expected %d, got %d\n",
            file, ln, line, exp, rc);
  exit(1);
}
```

Then you can wrap calls that should never fail using

```c
CHECK(stat("/tmp", &stat_buff), 0);
```

If it should fail, you’d get a message written to stderr:

```
source.c line 19
'stat("/tmp", &stat_buff)': expected 0, got -1
```

Clearly it is sometimes inappropriate simply to exit a running program. You may have claimed resources that might not get released, or you may need to write log messages, tidy up open transactions, or interact with other processes. The techniques we discuss in When to Use Exceptions, page 125, will help here. However, the basic principle stays the same—when your code discovers that something that was supposed to be impossible just happened, your program is no longer viable. Anything it does from this point forward becomes suspect, so terminate it as soon as possible. A dead program normally does a lot less damage than a crippled one.

Related sections include:
- Design by Contract, page 109
- When to Use Exceptions, page 125
Assertive Programming

There is a luxury in self-reproach. When we blame ourselves we feel no one else has a right to blame us.

- Oscar Wilde, The Picture of Dorian Gray

It seems that there’s a mantra that every programmer must memorize early in his or her career. It is a fundamental tenet of computing, a core belief that we learn to apply to requirements, designs, code, comments, just about everything we do. It goes

This can never happen...

“This code won’t be used 30 years from now, so two-digit dates are fine.”
“This application will never be used abroad, so why internationalize it?”
“count can’t be negative.” “This printf can’t fail.”

Let’s not practice this kind of self-deception, particularly when coding.

Tip 33

If It Can’t Happen, Use Assertions to Ensure That It Won’t

Whenever you find yourself thinking “but of course that could never happen,” add code to check it. The easiest way to do this is with assertions. In most C and C++ implementations, you’ll find some form of assert or _assert macro that checks a Boolean condition. These macros can be invaluable. If a pointer passed in to your procedure should never be NULL, then check for it:

```c
void writeString(char *string) {
    assert(string != NULL);
    ...
}
```

Assertions are also useful checks on an algorithm’s operation. Maybe you’ve written a clever sort algorithm. Check that it works:

```c
for (int i = 0; i < num_entries-1; i++) {
    assert(sorted[i] <= sorted[i+1]);
}
```

Of course, the condition passed to an assertion should not have a side effect (see the box on page 124). Also remember that assertions may be turned off at compile time—never put code that must be executed into an assert.
Don't use assertions in place of real error handling. Assertions check for things that should never happen: you don't want to be writing code such as

```c
printf("Enter 'Y' or 'N': ");
ch = getchar();
assert((ch == 'Y') || (ch == 'N')); /* bad idea! */
```

And just because the supplied assert macros call exit when an assertion fails, there's no reason why versions you write should. If you need to free resources, have an assertion failure generate an exception, longjmp to an exit point, or call an error handler. Just make sure the code you execute in those dying milliseconds doesn't rely on the information that triggered the assertion failure in the first place.

**Leave Assertions Turned On**

There is a common misunderstanding about assertions, promulgated by the people who write compilers and language environments. It goes something like this:

> Assertions add some overhead to code. Because they check for things that should never happen, they'll get triggered only by a bug in the code. Once the code has been tested and shipped, they are no longer needed, and should be turned off to make the code run faster. Assertions are a debugging facility.

There are two patently wrong assumptions here. First, they assume that testing finds all the bugs. In reality, for any complex program you are unlikely to test even a miniscule percentage of the permutations your code will be put through (see *Ruthless Testing*, page 245). Second, the optimists are forgetting that your program runs in a dangerous world. During testing, rats probably won't gnaw through a communications cable, someone playing a game won't exhaust memory, and log files won't fill the hard drive. These things might happen when your program runs in a production environment. Your first line of defense is checking for any possible error, and your second is using assertions to try to detect those you've missed.

Turning off assertions when you deliver a program to production is like crossing a high wire without a net because you once made it across in practice. There's dramatic value, but it's hard to get life insurance.

Even if you *do* have performance issues, turn off only those assertions that really hit you. The sort example above may be a critical part of
Assertions and Side Effects

It is embarrassing when the code we add to detect errors actually ends up creating new errors. This can happen with assertions if evaluating the condition has side effects. For example, in Java it would be a bad idea to code something such as

```java
while (iter.hasMoreElements()) {
    TestASSERT(iter.nextElement() != null);
    Object obj = iter.nextElement();
    // ....
}
```

The `nextElement()` call in the `ASSERT` has the side effect of moving the iterator past the element being fetched, and so the loop will process only half the elements in the collection. It would be better to write

```java
while (iter.hasMoreElements()) {
    Object obj = iter.nextElement();
    TestASSERT(obj != null);
    // ....
}
```

This problem is a kind of “Heisenbug”—debugging that changes the behavior of the system being debugged (see [URL 52]).

your application, and may need to be fast. Adding the check means another pass through the data, which might be unacceptable. Make that particular check optional, but leave the rest in.

Related sections include:

- *Debugging*, page 90
- *Design by Contract*, page 109
- *How to Balance Resources*, page 129
- *Programming by Coincidence*, page 172

2. In C-based languages, you can either use the preprocessor or use if statements to make assertions optional. Many implementations turn off code generation for the `assert` macro if a compile-time flag is set (or not set). Otherwise, you can place the code within an if statement with a constant condition, which many compilers (including most common Java systems) will optimize away.
Exercises

19. A quick reality check. Which of these “impossible” things can happen?

   1. A month with fewer than 28 days
   2. stat(".", &sb) == -1 (that is, can’t access the current directory)
   3. In C++: a = 2; b = 3; if (a + b != 5) exit(1);
   4. A triangle with an interior angle sum ≠ 180°
   5. A minute that doesn’t have 60 seconds
   6. In Java: (a + 1) <= a

20. Develop a simple assertion checking class for Java.

When to Use Exceptions

In Dead Programs Tell No Lies, page 120, we suggested that it is good practice to check for every possible error—particularly the unexpected ones. However, in practice this can lead to some pretty ugly code; the normal logic of your program can end up being totally obscured by error handling, particularly if you subscribe to the “a routine must have a single return statement” school of programming (we don’t). We’ve seen code that looks something like the following:

```c
retcode = OK;
if (socket.read(name) != OK) {
  retcode = BAD_READ;
} else {
  processName(name);
  if (socket.read(address) != OK) {
    retcode = BAD_READ;
  } else {
    processAddress(address);
    if (socket.read(telNo) != OK) {
      retcode = BAD_READ;
    } else {
      // etc, etc...
    }
  }
} return retcode;
```

Fortunately, if the programming language supports exceptions, you can rewrite this code in a far neater way:
retcode = OK;

try {
    socket.read(name);
    process(name);
    socket.read(address);
    processAddress(address);
    socket.read(telNo);
    // etc, etc...
} catch (IOException e) {
    retcode = BAD_READ;
    Logger.log("Error reading individual: "+ e.getMessage());
}

return retcode;

The normal flow of control is now clear, with all the error handling moved off to a single place.

What Is Exceptional?

One of the problems with exceptions is knowing when to use them. We believe that exceptions should rarely be used as part of a program's normal flow; exceptions should be reserved for unexpected events. Assume that an uncaught exception will terminate your program and ask yourself, "Will this code still run if I remove all the exception handlers?" If the answer is "no," then maybe exceptions are being used in nonexceptional circumstances.

For example, if your code tries to open a file for reading and that file does not exist, should an exception be raised?

Our answer is, "It depends." If the file should have been there, then an exception is warranted. Something unexpected happened—a file you were expecting to exist seems to have disappeared. On the other hand, if you have no idea whether the file should exist or not, then it doesn’t seem exceptional if you can’t find it, and an error return is appropriate.

Let’s look at an example of the first case. The following code opens the file /etc/passwd, which should exist on all Unix systems. If it fails, it passes on the FileNotFoundException to its caller.

```java
public void open_passwd() throws FileNotFoundException {
    // This may throw FileNotFoundException...
    inputstream = new FileInputStream("/etc/passwd");
    // ...
}
```
However, the second case may involve opening a file specified by the user on the command line. Here an exception isn’t warranted, and the code looks different:

```java
public boolean open_user_file(String name) throws FileNotFoundException {
    File f = new File(name);
    if (!f.exists()) {
        return false;
    }
    InputStream = new FileInputStream(f);
    return true;
}
```

Note that the FileInputStream call can still generate an exception, which the routine passes on. However, the exception will be generated under only truly exceptional circumstances; simply trying to open a file that does not exist will generate a conventional error return.

### Tip 34

Use Exceptions for Exceptional Problems

Why do we suggest this approach to exceptions? Well, an exception represents an immediate, nonlocal transfer of control—it’s a kind of cascading goto. Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code. These programs break encapsulation: routines and their callers are more tightly coupled via exception handling.

### Error Handlers Are an Alternative

An error handler is a routine that is called when an error is detected. You can register a routine to handle a specific category of errors. When one of these errors occurs, the handler will be called.

There are times when you may want to use error handlers, either instead of or alongside exceptions. Clearly, if you are using a language such as C, which does not support exceptions, this is one of your few other options (see the challenge on the next page). However, sometimes error handlers can be used even in languages (such as Java) that have a good exception handling scheme built in.
Consider the implementation of a client-server application, using Java's Remote Method Invocation (RMI) facility. Because of the way RMI is implemented, every call to a remote routine must be prepared to handle a RemoteException. Adding code to handle these exceptions can become tedious, and means that it is difficult to write code that works with both local and remote routines. A possible work-around is to wrap your remote objects in a class that is not remote. This class then implements an error handler interface, allowing the client code to register a routine to be called when a remote exception is detected.

**Related sections include:**
- *Dead Programs Tell No Lies*, page 120

**Challenges**
- Languages that do not support exceptions often have some other nonlocal transfer of control mechanism (C has longjmp/setjmp, for example). Consider how you could implement some kind of ersatz exception mechanism using these facilities. What are the benefits and dangers? What special steps do you need to take to ensure that resources are not orphaned? Does it make sense to use this kind of solution whenever you code in C?

**Exercises**

21. While designing a new container class, you identify the following possible error conditions:

1. No memory available for a new element in the add routine
2. Requested entry not found in the fetch routine
3. Nullable pointer passed to the add routine

How should each be handled? Should an error be generated, should an exception be raised, or should the condition be ignored?