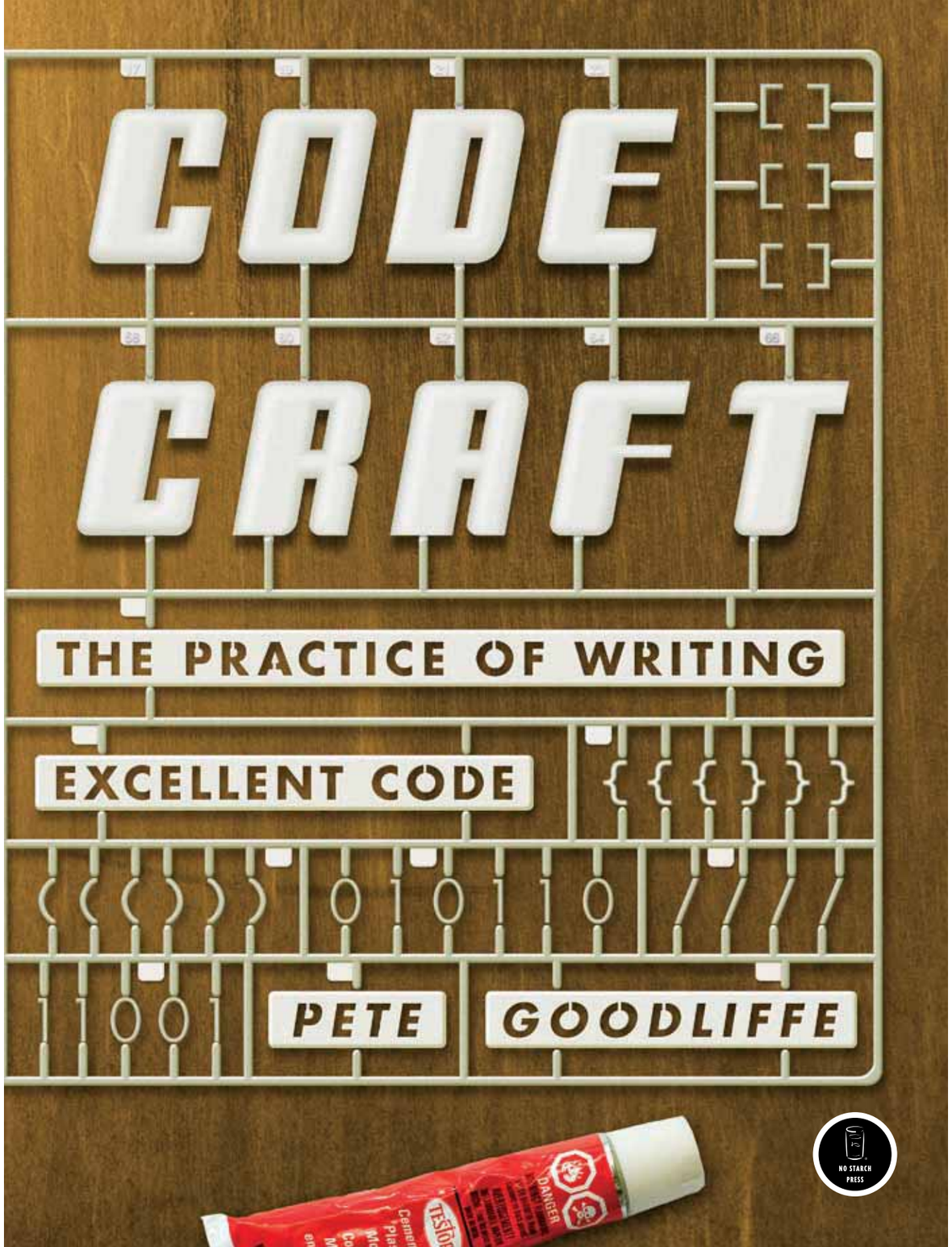

CODE
CRAFT
THE PRACTICE OF WRITING
EXCELLENT CODE
{{{}}}
((()))
010110
////
1001
PETE
GOODLIFFE
DANGER
NO STARCH
PRESS

6

TO ERR IS HUMAN

Dealing with the Inevitable—Error Conditions in Code

In this chapter:

- The types of errors we encounter
- Dealing with errors correctly
- How to raise errors
- Learning to program in the face of uncertainty

We know that the only way to avoid error is to detect it, that the only way to detect it is to be free to enquire.
—J. Robert Oppenheimer

At some point in life, everyone has this epiphany: *The world doesn't work as you expect it to.* My one-year-old friend Tom learned this when climbing a chair four times his size. He expected to get to the top. The actual result surprised him: He ended up under a pile of furniture.

Is the world broken? Is it wrong? No. The world has plodded happily along its way for the last few million years and looks set to continue for the foreseeable future. It's *our expectations* that are wrong and need to be adjusted. As they say: *Bad things happen, so deal with it.* We must write code that deals with the Real World and its unexpected ways.

This is particularly difficult because the world *mostly* works as we'd expect it to, constantly lulling us into a false sense of security. The human brain is

wired to cope, with built-in fail-safes. If someone bricks up your front door, your brain will process the problem, and you'll stop before walking into an unexpected wall. But programs are not so clever; we have to tell them where the brick walls are and what to do when they hit one.

Don't presume that everything in your program will always run smoothly. The world doesn't always work as you'd expect it to: You *must* handle all possible error conditions in your code. It sounds simple enough, but that statement leads to a world of pain.

From Whence It Came

To expect the unexpected shows
a thoroughly modern intellect.
—Oscar Wilde

Errors can and will occur. Undersirable results can arise from almost any operation. They are distinct from bugs in a faulty program because you *know* beforehand that an error can occur. For example, the database file you want to open might have been deleted, a disk could fill up at any time and your next save operation might fail, or the web service you're accessing might not currently be available.

If you don't write code to handle these error conditions, you will almost certainly end up with a *bug*; your program will not always work as you intend it to. But if the error happens only rarely, it will probably be a very subtle bug! We'll look at bugs in Chapter 9.

An error may occur for one of a thousand reasons, but it will fall into one of these three categories:

User error
The stupid user manhandled your lovely program. Perhaps he provided the wrong input or attempted an operation that's absolutely absurd. A good program will point out the mistake and help the user rectify it. It won't insult him or whine in an incomprehensible manner.

Programmer error
The user pushed all the right buttons, but the code is broken. This is the consequence of a bug elsewhere, a fault the programmer introduced that the user can do nothing about (except to try and avoid it in the future). This kind of error should (ideally) never occur.

There's a cycle here: Unhandled errors can cause bugs. And those bugs might result in further error conditions occurring elsewhere in your code. This is why we consider defensive programming an important practice.

Exceptional circumstances
The user pushed all the right buttons, and the programmer didn't mess up. Fate's fickle finger intervened, and we ran into something that couldn't be avoided. Perhaps a network connection failed, we ran out of printer ink, or there's no hard disk space left.

We need a well-defined strategy to manage each kind of error in our code. An error may be detected and reported to the user in a pop-up message box, or it may be detected by a middle-tier code layer and signaled to the client code programmatically. The same principles apply in both cases: whether a human chooses how to handle the problem or your code makes a decision—*someone* is responsible for acknowledging and acting on errors.

KEY CONCEPT *Take error handling seriously. The stability of your code rests on it.*

Errors are raised by subordinate components and communicated upward, to be dealt with by the caller. They are reported in a number of ways; we'll look at these in the next section. To take control of program execution, we must be able to:

- Raise an error when something goes wrong
- Detect all possible error reports
- Handle them appropriately
- Propagate errors we can't handle

Errors are hard to deal with. The error you encounter is often not related to what you were doing at the time (most fall under the "exceptional circumstances" category). They are also tedious to deal with—we want to focus on what our program *should* be doing, not on how it may go wrong. However, without good error management, your program will be brittle—built upon sand, not rock. At the first sign of wind or rain, it will collapse.

Error-Reporting Mechanisms

There are several common strategies for propagating error information to client code. You'll run into code that uses each of them, so you must know how to speak every dialect. Observe how these error-reporting techniques compare, and notice which situations call for each mechanism.

Each mechanism has different implications for the *locality of error*. An error is local in *time* if it is discovered very soon after it is created. An error is local in *space* if it is identified very close to (or even *at*) the site where it actually manifests. Some approaches specifically aim to reduce the locality of error to make it easier to see what's going on (e.g., error codes). Others aim to extend the locality of error so that normal code doesn't get entwined with error-handling logic (e.g., exceptions).

The favored reporting mechanism is often an architectural decision. The architect might consider it important to define a homogeneous hierarchy of exception classes or a central list of shared reason codes to unify error-handling code.

No Reporting

The simplest error-reporting mechanism is *don't bother*. This works wonderfully in cases where you want your program to behave in bizarre and unpredictable ways and to crash randomly.

If you encounter an error and don't know what to do about it, blindly ignoring it is *not* a viable option. You probably can't continue the function's work, but returning without fulfilling your function's contract will leave the world in an undefined and inconsistent state.

KEY CONCEPT *Never ignore an error condition. If you don't know how to handle the problem, signal a failure back up to the calling code. Don't sweep an error under the rug and hope for the best.*

An alternative to ignoring errors is to instantly abort the program upon encountering a problem. It's easier than handling errors throughout the code, but hardly a well-engineered solution!

Return Values

The next most simple mechanism is to return a success/failure value from your function. A boolean return value provides a simple yes or no answer. A more advanced approach enumerates all the possible exit statuses and returns a corresponding *reason code*. One value means *success*; the rest represent the many and varied abortive cases. This enumeration may be shared across the whole codebase, in which case your function returns a subset of the available values. You should therefore document what the caller can expect.

While this works well for procedures that don't return data, passing error codes back *with* returned data gets messy. If `int count()` walks down a linked list and returns the number of elements, how can it signify a list structure corruption? There are three approaches:

- Return a compound data type (or *tuple*) containing both the return value and an error code. This is rather clumsy in the popular C-like languages and is seldom seen in them.
- Pass the error code back through a function parameter. In C++ or .NET, this parameter would be passed by reference. In C you'd direct the variable access through pointers. This approach is ugly and nonintuitive; there is no syntactic way to distinguish a return value from a parameter.
- Alternatively, reserve a range of return values to signify failure. The `count` example can nominate all negative numbers as error reason codes; they'd be meaningless answers anyway. Negative numbers are a common choice for this. Pointer return values may be given a specific invalid value, which by convention is zero (or `NULL`). In Java and C#, you can return a `null` object reference.

  This technique doesn't always work well. Sometimes it's hard to reserve an error range—all return values are equally meaningful and equally likely. It also has the side effect of reducing the available range of success values; the use of negative values reduces the possible positive values by an order of magnitude.[1]

[1] If you used an `unsigned int` then the number of values available would increase by a power of two, reusing the `signed int`'s sign bit.

Error Status Variables

This method attempts to manage the contention between a function's return value and its error status report. Rather than return a reason code, the function sets a shared global error variable. After calling the function, you must then inspect this status variable to find out whether or not it completed successfully.

The shared variable reduces confusion and clutter in the function's signature, and it doesn't restrict the return value's data range at all. However, errors signaled through a separate channel are much easier to miss or willfully ignore. A shared global variable also has nasty thread safety implications.

The C standard library employs this technique with its `errno` variable. It has very subtle semantics: Before using any standard library facility, you must manually clear `errno`. Nothing ever sets a succeeded value; only failures touch `errno`. This is a common source of bugs, and it makes calling each library function tedious. To add insult to injury, not all C standard library functions use `errno`, so it is less than consistent.

This technique is functionally equivalent to using return values, but it has enough disadvantages to make you avoid it. Don't write your own error reports this way, and use existing implementations with the utmost care.

Exceptions

Exceptions are a language facility for managing errors; not all languages support exceptions. Exceptions help to distinguish the normal flow of execution from *exceptional* cases—when a function has failed and cannot honor its contract. When your code encounters a problem that it can't handle, it stops dead and throws up an *exception*—an object representing the error. The language run time then automatically steps back up the call stack until it finds some exception-handling code. The error lands there, for the program to deal with.

There are two operational models, distinguished by what happens after an exception is handled:

The termination model
: Execution continues after the handler that caught the exception. This behavior is provided by C++, .NET, and Java.

The resumption model
: Execution resumes where the exception was raised.

The former model is easier to reason about, but it doesn't give ultimate control. It only allows *error handling* (you can execute code when you notice an error), not *fault rectification* (a chance to fix the problem and try again).

An exception cannot be ignored. If it isn't caught and handled, it will propagate to the very top of the call stack and will usually stop the program dead in its tracks. The language run time automatically cleans up as it unwinds

WHISTLE-STOP TOUR OF EXCEPTION SAFETY

Resilient code must be *exception safe*. It must work correctly (for some definition of *correctly*, which we'll investigate below), no matter what exceptions come its way. This is true regardless of whether or not the code catches any exceptions itself.

Exception-neutral code propagates all exceptions up to the caller; it won't consume or change anything. This is an important concept for generic programs like C++ template code—the template types may generate all sorts of exceptions that template implementors don't understand.

There are several different levels of exception safety. They are described in terms of guarantees to the calling code. These guarantees are:

Basic guarantee

If exceptions occur in a function (resulting from an operation you perform or the call of another function), it will not leak resources. The code state will be *consistent* (i.e., it can still be used correctly), but it will not necessarily leave in a known state. For example: A member function should add 10 items to a container, but an exception propagates through it. The container is still usable; maybe no objects were inserted, maybe all 10 were, or perhaps every other object was added.

Strong guarantee

This is far more strict than the basic guarantee. If an exception propagates through your code, the program state remains completely unchanged. No object is altered, no global variables changed, nothing. In the example above, nothing was inserted into the container.

Nothrow guarantee

The final guarantee is the most restrictive: that an operation can *never* throw an exception. If we are exception neutral, then this implies the function cannot do anything else that might throw an exception.

Which guarantee you provide is entirely your choice. The more restrictive the guarantee, the more widely (re)usable the code is. In order to implement the strong guarantee, you will generally need a number of functions providing the nothrow guarantee.

Most notably, every destructor you write *must* honor the nothrow guarantee.* Otherwise, all exception handling bets are off. In the presence of an exception, object destructors are called automatically as the stack is unwound. Raising an exception while handling an exception is not permissible.

*That's the case in C++ and Java, at least. C# stupidly called `~X()` a *destructor*, even though it was a finalizer in disguise. Throwing an exception in a C# destructor has different implications.

the call stack. This makes exceptions a tidier and safer alternative to hand-crafted error-handling code. However, throwing exceptions through sloppy code can lead to memory leaks and problems with resource cleanup.[2] You must take care to write *exception-safe* code. The sidebar explains what this means in more detail.

The code that handles an exception is distinct from the code that raises it, and it may be arbitrarily far away. Exceptions are usually provided by OO languages, where errors are defined by a hierarchy of exception classes.

[2] For example, you could allocate a block of memory and then exit early as an exception propagates through. The allocated memory would leak. This kind of problem makes writing code in the face of exceptions a complex business.

A handler can elect to catch a quite specific class of error (by accepting a leaf class) or a more general category of error (by accepting a base class). Exceptions are particularly useful for signaling errors in a constructor.

Exceptions don't come for free; the language support incurs a performance penalty. In practice, this isn't significant and only manifests around exception-handling statements—exception handlers reduce the compiler's optimization opportunities. This doesn't mean that exceptions are flawed; their expense is justified compared to the cost of not doing any error handling!

Signals

Signals are a more extreme reporting mechanism, largely used for errors sent by the execution environment to the running program. The operating system traps a number of exceptional events, like a *floating point exception* triggered by the maths coprocessor. These well-defined error events are delivered to the application in signals that interrupt the program's normal flow of execution, jumping into a nominated *signal handler* function. Your program could receive a signal at any time, and the code must be able to cope with this. When the signal handler completes, program execution continues at the point it was interrupted.

Signals are the software equivalent of a hardware interrupt. They are a Unix concept, now provided on most platforms (a basic version is part of the ISO C standard [ISO99]). The operating system provides sensible default handlers for each signal, some of which do nothing, others of which abort the program with a neat error message. You can override these with your own handler.

The defined C signal events include program termination, execution suspend/continue requests, and math errors. Some environments extend the basic list with many more events.

Detecting Errors

How you detect an error obviously depends on the mechanism reporting it. In practical terms, this means:

Return values
: You determine whether a function failed by looking at its return code. This failure test is bound tightly to the act of calling the function; by making the call, you are implicitly checking its success. Whether or not you do anything with that information is up to you.

Error status variables
: After calling a function, you must inspect the error status variable. If it follows C's `errno` model of operation, you don't actually need to test for errors after every single function call. First reset `errno`, and then call any number of standard library functions back-to-back. Afterward, inspect `errno`. If it contains an error value, then one of those functions failed. Of course, you don't know what fell over, but if you don't care, then this is a streamlined error-detection approach.

Exceptions

If an exception propagates out of a subordinate function, you can choose to catch and handle it or to ignore it and let the exception flow up a level. You can only make an informed choice when you know what kinds of exceptions might be thrown. You'll only know this if it has been documented (and if you trust the documentation).

Java's exception implementation places this documentation in the code itself. The programmer has to write an *exception specification* for every method, describing what it can throw; it is a part of the function's signature. Java is the only mainstream language to enforce this approach. You cannot leak an exception that isn't in the list, because the compiler performs static checking to prevent it.[3]

Signals

There's only one way to detect a signal: Install a hander for it. There's no obligation. You can also choose not to install any signal handlers at all and accept the default behavior.

As various pieces of code converge in a large system, you will probably need to detect errors in more than one way, even within a single function. Whichever detection mechanism you use, the key point is this:

KEY CONCEPT *Never ignore* any *errors that might be reported to you. If an error report channel exists, it's there for a reason.*

It is good practice to always write error-detection scaffolding—even if an error has no implication for the rest of your code. This makes it clear to a maintenance programmer that you know a function may fail and have consciously chosen to ignore any failures.

When you let an exception propagate through your code, you are not ignoring it—you *can't* ignore an exception. You are allowing it to be handled by a higher level. The philosophy of exception handling is quite different in this respect. It's less clear what the most appropriate way to document this is—should you write a `try/catch` block that simply `rethrows` the exception, should you write a comment claiming that the code *is* exception safe, or should you do nothing? I'd favor documenting the exception behavior.

Handling Errors

Love truth, and pardon error.
—Voltaire

Errors happen. We've seen how to discover them and when to do so. The question now is: What do you do about them? This is the hard part. The answer largely depends on circumstance and the gravity of an error—whether it's possible to rectify the problem and retry the operation or to carry on regardless. Often there is no such luxury; the error may even

[3] C++ also supports exception specifications, but leaves their use optional. It's idiomatic to avoid them—for performance reasons, among others. Unlike Java, they are enforced at run time.

herald the beginning of the end. The best you can do is clean up and exit sharply, before anything else goes wrong.

To make this kind of decision, you must be informed. You need to know a few key pieces of information about the error:

Where it came from
: This is quite distinct from where it's going to be handled. Is the source a core system component or a peripheral module? This information may be encoded in the error report; if not, you can figure it out manually.

What you were trying to do
: What provoked the error? This may give a clue toward any remedial action. Error reporting seldom contains this kind of information, but you can figure out which function was called from the context.

Why it went wrong
: What is the nature of the problem? You need to know exactly what happened, not just a general *class* of error. How much of the erroneous operation completed? *All* or *none* are nice answers, but generally, the program will be in some indeterminate state between the two.

When it happened
: This is the locality of the error in time. Has the system only just failed, or is a problem two hours old finally being felt?

The severity of the error
: Some problems are more serious than others, but when detected, one error is equivalent to another—you can't continue without understanding and managing the problem. Error severity is usually determined by the caller, based on how easy it will be to recover or work around the error.

How to fix it
: This may be obvious (e.g., insert a floppy disk and retry) or not (e.g., you need to modify the function parameters so they are consistent). More often than not, you have to infer this knowledge from the other information you have.

Given this depth of information, you can formulate a strategy to handle each error. Forgetting to insert a handler for any potential error will lead to a bug, and it might turn out to be a bug that is hard to exercise and hard to track down—so think about every error condition carefully.

When to Deal with Errors

When should you handle each error? This can be separate from when it's detected. There are two schools of thought.

As soon as possible
: Handle each error *as* you detect it. Since the error is handled near to its cause, you retain important contextual information, making the error-handling code clearer. This is a well-known self-documenting code

technique. Managing each error near its source means that control passes through less code in an invalid state.

This is usually the best option for functions that return error codes.

As late as possible

Alternatively, you could defer error handling for as long as possible. This recognizes that code detecting an error rarely knows what to do about it. It often depends on the context in which it is used: A missing file error may be reported to the user when loading a document but silently swallowed when hunting for a preferences file.

Exceptions are ideal for this; you can pass an exception through each level until you know how to deal with the error. This separation of detection and handling may be clearer, but it can make code more complex. It's not obvious that you are deliberately deferring error handling, and it's not clear where an error came from when you do finally handle it.

In theory, it's nice to separate "business logic" from error handling. But often you can't, as cleanup is necessarily entwined with that business logic, and it can be more tortuous to write the two separately. However, centralized error-handling code has advantages: You know where to look for it, and you can put the abort/continue policy in one place rather than scatter it through many functions.

Thomas Jefferson once declared, "Delay is preferable to error." There is truth there; the actual *existence* of error handling is far more important than *when* an error is handled. Nevertheless, choose a compromise that's close enough to prevent obscure and out-of-context error handling, while being far enough away to not cloud normal code with roundabout paths and error handling dead ends.

KEY CONCEPT *Handle each error in the most appropriate context, as soon as you know enough about it to deal with it correctly.*

Possible Reactions

You've caught an error. You're poised to handle it. What are you going to do now? Hopefully, whatever is required for correct program operation. While we can't list every recovery technique under the sun, here are the common reactions to consider.

Logging

Any reasonably large project should already be employing a logging facility. It allows you to collect important trace information, and is an entry point for the investigation of nasty problems.

The log exists to record interesting events in the life of the program, to allow you to delve into its inner workings and reconstruct paths of execution. For this reason, all errors you encounter should be detailed in the program log; they are some of the most interesting and telling events of all. Aim to capture all pertinent information—as much of the previous list as you can.

For really obscure errors that predict catastrophic disaster, it may be a good idea to get the program to "phone home"—to transmit either a snapshot of itself or a copy of the error log to the developers for further investigation.

What you do *after* logging is another matter.

Reporting

A program should only report an error to the user when there's nothing left to do. The user does not need to be bombarded by a thousand small nuggets of useless information or badgered by a raft of pointless questions. Save the interaction for when it's really vital. Don't report when you encounter a recoverable situation. By all means, log the event, but keep quiet about it. Provide a mechanism that enables users to read the event log if you think one day they might care.

There *are* some problems that only the user can fix. For these, it is good practice to report the problem immediately, in order to allow the user the best chance to resolve the situation or else decide how to continue.

Of course, this kind of reporting depends on whether or not the program is interactive. Deeply embedded systems are expected to cope on their own; it's hard to pop up a dialog box on a washing machine.

Recovery

Sometimes your only course of action is to stop immediately. But not all errors spell doom. If your program saves a file, one day the disk will fill up, and the save operation will fail. The user expects your program to continue happily, so be prepared.

If your code encounters an error and doesn't know what to do about it, pass the error upward. It's more than likely your caller will have the ability to recover.

Ignore

I only include this for completeness. Hopefully by now you've learned to scorn the very suggestion of ignoring an error. If you choose to forget all about handling it and to just continue with your fingers crossed, *good luck*. This is where most of the bugs in any software package will come from. Ignoring an error whose occurrence may cause the system to misbehave inevitably leads to hours of debugging.

You can, however, write code that allows you to *do nothing* when an error crops up. Is that a blatant contradiction? No. It is possible to write code that copes with an inconsistent world, that can carry on correctly in the face of an error—but it often gets quite convoluted. If you adopt this approach, you must make it obvious in the code. Don't risk having it misinterpreted as ignorant and incorrect.

KEY CONCEPT *Ignoring errors does not save time. You'll spend far longer working out the cause of bad program behavior than you ever would have spent writing the error handler.*

Propagate

When a subordinate function call fails, you probably can't carry on, but you might not know what else to do. The only option is to clean up and propagate the error report upward. You have options. There are two ways to propagate an error:

- Export the same error information you were fed (return the same reason code or propagate exceptions).
- Reinterpret the information, sending a more meaningful message to the next level up (return a different reason code or catch and wrap up exceptions).

Ask yourself this question: Does the error relate to a concept exposed through the module interface? If so, it's okay to propagate that same error. Otherwise, recast it in the appropriate light, choosing an error report that makes sense in the context of your module's interface. This is a good self-documenting code technique.

Code Implications

Show me the code! Let's spend some time investigating the implications of error handling in our code. As we'll see, it is not easy to write good error handling that doesn't twist and warp the underlying program logic.

The first piece of code we'll look at is a common error handling structure. Yet it isn't a particularly intelligent approach for writing error-tolerant code. The aim is to call three functions sequentially—each of which may fail—and perform some intermediate calculations along the way. Spot the problems with this:

```
void nastyErrorHandling()
{
    if (operationOne())
    {
        ... do something ...
        if (operationTwo())
        {
            ... do something else ...
            if (operationThree())
            {
                ... do more ...
            }
        }
    }
}
```

Syntactically it's fine; the code will work. Practically, it's an unpleasant style to maintain. The more operations you need to perform, the more deeply nested the code gets and the harder it is to read. This kind of error handling quickly leads to a rat's nest of conditional statements. It doesn't reflect the actions of the code very well; each intermediate calculation could be considered the same level of importance, yet they are nested at different levels.

CRAFTING ERROR MESSAGES

Inevitably, your code will encounter errors that the user must sort out. Human intervention may be the only option; your code can't insert a floppy disk or switch on the printer by itself. (If it can, you'll make a fortune!)

If you're going to whine at the user, there are a few general points to bear in mind:

- Users don't think like programmers, so present information the way *they'd* expect. When displaying the free space on a disk, you might report `Disk space: 10K`. But if there's no space left, a zero could be misread as `OK`—and the user will not be able to fathom why he can't save a file when the program says everything's fine.
- Make sure your messages aren't too cryptic. *You* might understand them, but can your computer-illiterate granny? (It doesn't matter if your granny won't use this program—someone with a lower intellect almost certainly will.)
- Don't present meaningless error codes. No user knows what to do when faced with an `Error code 707E`. It is, however, valuable to provide such codes as "additional info"—they can be quoted to tech support or looked up more easily on a web search.
- Distinguish dire errors from mere warnings. Incorporate this information in the message text (perhaps with an `Error:` prefix), and emphasize it in message boxes with an accompanying icon.
- Only ask a question (even a simple one like *Continue: Yes/No?*) if the user fully understands the ramifications of each choice. Explain it if necessary, and make it clear what the consequence of each answer is.

What you present to the user will be determined by interface constraints and application or OS style guides. If your company has user interface engineers, then it's their job to make these decisions. Work with them.

Can we avoid these problems? Yes—there are a few alternatives. The first variant flattens the nesting. It is semantically equivalent, but it introduces *some* new complexity, since flow control is now dependent on the value of a new status variable, `ok`:

```
void flattenedErrorHandling()
{
    bool ok = operationOne();
    if (ok)
    {
        ... do something ...
        ok = operationTwo();
    }
    if (ok)
    {
        ... do something else ...
        ok = operationThree();
    }
    if (ok)
    {
        ... do more ...
    }
```

```
    if (!ok)
    {
        ... clean up after errors ...
    }
}
```

We've also added an opportunity to clean up after any errors. Is that sufficient to mop up all failures? Probably not; the necessary cleanup may depend on how far we got through the function before lightning struck. There are two cleanup approaches:

- Perform a little cleanup after each operation that may fail, then return early. This inevitably leads to duplication of cleanup code. The more work you've done, the more you have to clean up, so each exit point will need to do gradually more unpicking.

  If each operation in our example allocates some memory, each early-exit point will have to release all allocations made to date. The further in, the more releases. That will lead to some quite dense and repetitive error-handling code, which makes the function far larger and far harder to understand.
- Write the cleanup code once, at the end of the function, but write it in such a way as to only clean up what's dirty. This is neater, but if you inadvertently insert an early return in the middle of the function, the cleanup code will be bypassed.

If you're not overly concerned about writing *Single Entry, Single Exit (SESE)* functions, this next example removes the reliance on a separate control flow variable.[4] We do lose the cleanup code again, though. Simplicity renders this a better description of the actual intent:

```
void shortCircuitErrorHandling()
{
    if (!operationOne()) return;
    ... do something ...
    if (!operationTwo()) return;
    ... do something else ...
    if (!operationThree()) return;
    ... do more ...
}
```

A combination of this short-circuit exit with the requirement for cleanup leads to the following approach, especially seen in low-level systems code. Some people advocate it as the *only* valid use for the maligned `goto`. I'm still not convinced.

[4] Although this clearly isn't SESE, I contend that the previous example isn't, either. There *is* only one exit point, at the end, but the contrived control flow is simulating early exit—it *might as well* have multiple exits. This is a good example of how being bound by a rule like SESE can lead to bad code, unless you think carefully about what you're doing.

```
void gotoHell()
{
    if (!operationOne()) goto error;
    ... do something ...
    if (!operationTwo()) goto error;
    ... do something else ...
    if (!operationThree()) goto error;
    ... do more ...
    return;
error:
    ... clean up after errors ...
}
```

You can avoid such monstrous code in C++ using *Resource Acquisition Is Initialization (RAII)* techniques like smart pointers. (Stroustrup 97) This has the bonus of providing exception safety—when an exception terminates your function prematurely, resources are automatically deallocated. These techniques avoid a lot of the problems we've seen above, moving complexity to a separate flow of control.

The same example using exceptions would look like this (in C++, Java, and C#), presuming that all subordinate functions do not return error codes but instead throw exceptions:

```
void exceptionalHandling()
{
    try
    {
        operationOne();
        ... do something ...
        operationTwo();
        ... do something else ...
        operationThree();
        ... do more ...
    }
    catch (...)
    {
        ... clean up after errors ...
    }
}
```

This is only a basic exception example, but it shows just how neat exceptions can be. A sound code design might not need the `try/catch` block at all if it ensures that no resource is leaked and leaves error handling to a higher level. But alas, writing good code in the face of exceptions requires an understanding of principles beyond the scope of this chapter.

Raising Hell

We've put up with other people's errors for long enough. It's time to turn the tables and play the bad guy: Let's raise some errors. When writing a function, erroneous things will happen that you'll need to signal to your caller. Make sure you do—don't silently swallow any failure. Even if you're sure that the caller won't know what to do in the face of the problem, it *must* remain informed. Don't write code that lies and pretends to be doing something it's not.

Which reporting mechanism should you use? It's largely an architectural choice; obey the project conventions and the common language idioms. In languages with the facility, it is common to favor exceptions, but only use them if the rest of the project does. Java and C# really leave you with no choice; exceptions are buried deep in their execution run times. A C++ architecture may choose to forego this facility to achieve portability with platforms that have no exception support or to interface with older C code.

We've already seen strategies for propagating errors from subordinate function calls. Our main concern here is reporting fresh problems encountered during execution. How you determine these errors is your own business, but when reporting them, consider the following:

- Have you cleaned up appropriately first? Reliable code doesn't leak resources or leave the world in an inconsistent state, even when an error occurs, unless it's *really* unavoidable. If you do either of these things, it must be documented carefully. Consider what will happen the next time your code is called if this error has manifested. Ensure it will still work.
- Don't leak inappropriate information to the outside world in your error reports. Only return useful information that the caller understands and can act on.
- Use exceptions correctly. Don't throw an exception for unusual return values—the rare but not erroneous cases. Only use exceptions to signal circumstances where a function is not able to meet its contract. Don't use them non-idiomatically (i.e., for flow control).
- Consider using assertions (see "Constraints" on page 16) if you're trapping an error that should never happen in the normal course of program execution, a genuine programming error. Exceptions are a valid choice for this too—some assertion mechanisms can be configured to throw exceptions when they trigger.
- If you can pull forward any tests to compile time, then do so. The sooner you detect and rectify an error, the less hassle it can cause.
- Make it hard for people to ignore your errors. Given half a chance, someone *will* use your code badly. Exceptions are good for this—you have to act deliberately to hide an exception.

What kind of errors should you be looking out for? This obviously depends on what the function is doing. Here's a checklist of the general kinds of error checks you should make in each function:

- Check all function parameters. Ensure you have been given correct and consistent input. Consider using assertions for this, depending on how strictly your contract was written. (Is it an offense to supply bad parameters?)
- Check that invariants are satisfied at interesting points in execution.
- Check all values from external sources for validity before you use them. File contents and interactive input must be sensible, with no missing pieces.
- Check the return status of all system and other subordinate function calls.

AN EXCEPTION TO THE RULE

Exceptions are a powerful error reporting mechanism. Used well, they can simplify your code greatly while helping you to write robust software. In the wrong hands, though, they are a deadly weapon.

I once worked on a project where it was routine for programmers to break a `while` loop or end recursion by throwing an exception, using it as a non-local `goto`. It's an intersting idea, and kind of cute when you first see it. But this behavior is nothing more than an abuse of exceptions: It isn't what exceptions are idiomatically used for. More than one critical bug was caused by a maintenance programmer not understanding the flow of control through a complex, magically terminated loop.

Follow the idioms of your language, and don't write cute code for the sake of it.

Managing Errors

The common principle uniting the raising and handling of errors is to have a consistent strategy for dealing with failure, wherever it manifests. These are general considerations for managing the occurrence, detection, and handling of program errors:

- Avoid things that *could* cause errors. Can you do something that is guaranteed to work, instead? For example, avoid allocation errors by reserving enough resource beforehand. With an assured pool of memory, your routine cannot suffer memory restrictions. Naturally, this will only work when you know how much resource you need up front, but you often do.
- Define the program or routine's expected behavior under abnormal circumstances. This determines how robust the code needs to be and therefore how thorough your error handling should be. Can a function silently generate bad output, subscribing to the historic *GIGO* principle?[5]

[5] That is, *Garbage In, Garbage Out*—feed it trash, and it will happily spit out trash.

- Clearly define which components are responsible for handling which errors. Make it explicit in the module's interface. Ensure that your client knows what will always work and what may one day fail.
- Check your programming practice: *When* do you write error-handling code? Don't put it off until later; you'll forget to handle something. Don't wait until your development testing highlights problems before writing handlers—that's not an engineering approach.

KEY CONCEPT *Write all error detection and handling* now*, as you write the code that may fail. Don't put it off until later. If you must be evil and defer handling, at least write the detection scaffolding now.*

- When trapping an error, have you found a symptom or a cause? Consider whether you've discovered the source of a problem that needs to be rectified here or if you've discovered a symptom of an earlier problem. If it's the latter, then don't write reams of handling code here, put that in a more appropriate (earlier) error handler.

In a Nutshell

To err is human; to repent, divine; to persist, devilish.
—Benjamin Franklin

To err *is* human (but computers seem quite good at it, too). To handle these errors is divine.

Every line of code you write must be balanced by appropriate and thorough error checking and handling. A program without rigorous error handling will not be stable. One day an obscure error may occur, and the program will fall over as a result.

Handling errors and failure cases is hard work. It bogs programming down in the mundane details of the Real World. However, it's absolutely essential. As much as 90 percent of the code you write handles exceptional circumstances. (Bentley 82) That's a surprising statistic, so write code *expecting* to put far more effort into the things that can go wrong than the things that will go right.

Good programmers . . .

- Combine their good intentions with good coding practices
- Write the error-handling code *as* they write the main code
- Are *thorough* in the code they write, covering every error possibility

Bad programmers . . .

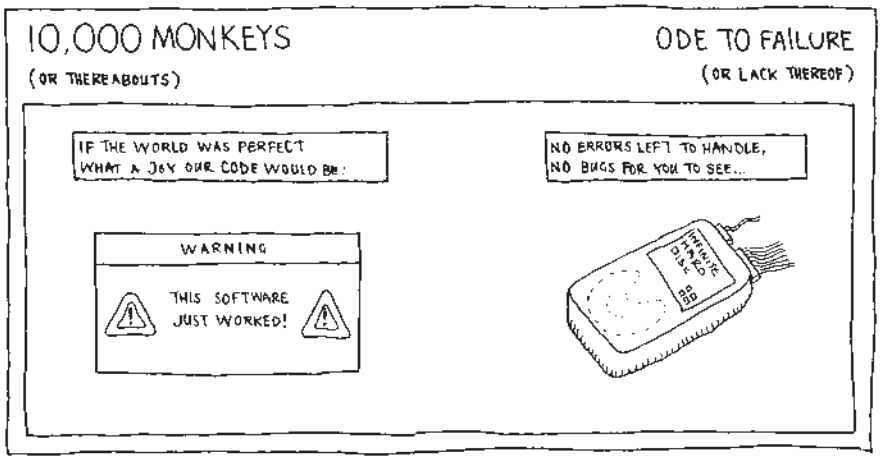
- Take a haphazard approach to writing code, with neither thought to nor review of what they're doing
- Ignore the errors that arise as they write code
- End up conducting lengthy debugging sessions to track down program crashes, because they never considered error conditions in the first place

See Also

Chapter 1: On the Defensive
Handing errors in context is one of the many defensive programming techniques.

Chapter 4: The Write Stuff
Self-documenting code ensures that error handling is integral to the code narrative.

Chapter 9: Finding Fault
Unhandled error conditions will manifest as bugs in the code. Here's how to squash them. (It's best to avoid them in the first place, though.)



Get Thinking

A detailed discussion of these questions can be found in the "Answers and Discussion" section on page 487.

Mull It Over

1. Are *return values* and *exceptions* equivalent error reporting mechanisms? Prove it.
2. What different implementations of *tuple* return types can you think of? Don't limit yourself to a single programming language. What are the pros and cons of using tuples as a return value?
3. How do exception implementations differ between languages?
4. Signals are an old-school Unix mechanism. Are they still needed now that we have modern techniques like exceptions?
5. What is the best code structure for error handling?
6. How should you handle errors that occur in your error-handling code?

Getting Personal

1. How thorough is the error handling in your current codebase? How does this contribute to the stability of the program?
2. Do you naturally consider error handling as you write code, or do you find it a distraction, preferring to come back to it later?
3. Go to the last (reasonably sized) function you wrote or worked on, and perform a careful review of the code. Find every abnormal occurence and potential error situation. How many of these were actually handled in your code?

   Now get someone else to review it. Don't be shy! Did they find any more? Why? What does this tell you about the code you're working on?
4. Do you find it easier to manage and reason about error conditions using *return values* or *exceptions*? Are you sure you know what is involved in writing exception-safe code?