Defensive Programming

Chapter 8 Code Complete

Contents

8.1 Protecting Your Program From Invalid Inputs

8.2 Assertions

8.3 Error Handling Techniques

8.4 Exceptions

8.5 Barricade Your Program to Contain the Damage Caused by Errors

8.6 Debugging Aids

8.7 Determining How Much Defensive Programming to Leave in Production Code

8.8 Being Defensive About Defensive Programming

The idea is based on defensive driving. In defensive driving, you adopt the mindset that you're never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won't be hurt. You take responsibility for protecting yourself even when it might be the other driver's fault.

This chapter describes how to protect yourself from the cold, cruel world of invalid data, events that can "never" happen, and other programmers' mistakes.

# 8.1. Protecting Your Program from Invalid Inputs

There are three general ways to handle garbage in:

1. **Check the values of all data from external sources.**

When getting data from a file, a user, the network, or some other external interface, check to be sure that the data falls within the allowable range.

**2. Check the values of all routine input parameters.**

Checking the values of routine input parameters is essentially the same as checking data that comes from an external source, except that the data comes from another routine instead of from an external interface.

The discussion in Barricade Your Program to Contain the Damage Caused by Errors, provides a practical way to determine which routines need to check their inputs.

**3. Decide how to handle bad inputs.**

Once you've detected an invalid parameter, what do you do with it? Depending on the situation, you might choose any of a dozen different approaches, which are described in detail in Error-Handling Techniques, later in this chapter.

The best form of defensive coding is not inserting errors in the first place. Using iterative design, writing pseudocode before code, writing test cases before writing the code, and having low-level design inspections are all activities that help to prevent inserting defects. They should thus be given a higher priority than defensive programming.

# 8.2. Assertions

* An assertion is code that's used during development—usually a routine or macro—that allows a program to check itself as it runs.
* When an assertion is true, that means everything is operating as expected. When it's false, that means it has detected an unexpected error in the code.
* Assertions are especially useful in large, complicated programs and in high-reliability programs.
* An assertion usually takes two arguments: a boolean expression that describes the assumption that's supposed to be true, and a message to display if it isn't.
* Normally, you don't want users to see assertion messages in production code; assertions are primarily for use during development and maintenance.
  + During production, they can be compiled out of the code so that the assertions don't degrade system performance.

## Guidelines for Using Assertions

### Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur.

* Assertions check for conditions that should never occur.
* Error handling typically checks for bad input data; assertions check for bugs in the code.
* If an assertion is fired for an anomalous condition, the corrective action is not merely to handle an error gracefully—the corrective action is to change the program's source code, recompile, and release a new version of the software.
* A good way to think of assertions is as executable documentation—you can't rely on them to make the code work, but they can document assumptions more actively than program-language comments can.

### Avoid putting executable code into assertions.

* Putting code into an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions.
  + Put executable statements on their own lines, assign the results to status variables, and test the status variables instead.

### Use assertions to document and verify preconditions and postconditions.

Preconditions and postconditions are part of an approach to program design and development known as "design by contract" (Meyer 1997). When preconditions and postconditions are used, each routine or class forms a contract with the rest of the program.

* Preconditions are the properties that the client code of a routine or class promises will be true before it calls the routine or instantiates the object.
* Postconditions are the properties that the routine or class promises will be true when it concludes executing.

Comments could be used to document preconditions and postconditions, but, unlike comments, assertions can check dynamically whether the preconditions and postconditions are true.

Example:

* If the variables latitude, longitude, and elevation were coming from an external source, invalid values should be checked and handled by error-handling code rather than by assertions.
* If the variables are coming from a trusted, internal source, however, and the routine's design is based on the assumption that these values will be within their valid ranges, then assertions are appropriate.

### For highly robust code, assert, and then handle the error anyway

For any given error condition a routine will generally use either an assertion or error-handling code, but not both. Some experts argue that only one kind is needed (Meyer 1997).

But real-world programs and projects tend to be too messy to rely solely on assertions.

On a large, long-lasting system, different parts might be designed by different designers over a period of 5–10 years or more. The designers will be separated in time, across numerous versions. Their designs will focus on different technologies at different points in the system's development. The designers will be separated geographically, especially if parts of the system are acquired from external sources. Programmers will have worked to different coding standards:

# 8.3. Error-Handling Techniques

Assertions are used to handle errors that should never occur in the code. How do you handle errors that you do expect to occur? Depending on the specific circumstances, you might:

* want to return a neutral value,
* substitute the next piece of valid data,
* return the same answer as the previous time,
* substitute the closest legal value,
* log a warning message to a file,
* return an error code,
* call an error-processing routine or object,
* display an error message,
* or shut down
* or you might want to use a combination of these responses.

### Return a neutral value.

Sometimes the best response to bad data is to continue operating and simply return a value that's known to be harmless.

* A numeric computation might return 0. A string operation might return an empty string, or a pointer operation might return an empty pointer.
* A drawing routine that gets a bad input value for color in a video game might use the default background or foreground color.

### Substitute the next piece of valid data.

When processing a stream of data, some circumstances call for simply returning the next valid data.

* If you're reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record.
* If you're taking readings from a thermometer 100 times per second and you don't get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

### Return the same answer as the previous time.

Example: if you detect a request to paint part of the screen an invalid color, you might simply return the same color used previously.

### Substitute the closest legal value.

In some cases, you might choose to return the closest legal value,

* Since my speedometer doesn't show negative speeds, when I back up it simply shows a speed of 0—the closest legal value.

### Log a warning message to a file.

When bad data is detected, you might choose to log a warning message to a file and then continue on.

* This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data.

### Return an error code.

* You could decide that only certain parts of a system will handle errors.
* Other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.

Set the value of a status variable

Return status as the function's return value Throw an exception by using the language's built-in exception mechanism

### Call an error-processing routine/object.

Another approach is to centralize error handling in a global error-handling routine or error-handling object.

* The advantage of this approach is that error-processing responsibility can be centralized, which can make debugging easier.
* The tradeoff is that the whole program will know about this central capability and will be coupled to it.
* If you ever want to reuse any of the code from the system in another system, you'll have to drag the error-handling machinery along with the code you reuse.

### Display an error message wherever the error is encountered.

This approach minimizes error-handling overhead; however, it does have the potential to spread user interface messages through the entire application, which can create challenges when you need to create a consistent user interface,

Attackers sometimes use error messages to discover how to attack a system.

### Shut down.

Some systems shut down whenever they detect an error. This approach is useful in safety-critical applications.

* if the software that controls radiation equipment for treating cancer patients receives bad input data for the radiation dosage, what is its best error-handling response? Should it use the same value as last time? Should it use the closest legal value? Should it use a neutral value? In this case, shutting down is the best option.

## Robustness vs. Correctness

As the video game and x-ray examples show us, the style of error processing that is most appropriate depends on the kind of software the error occurs in.

* Correctness means never returning an inaccurate result; returning no result is better than returning an inaccurate result.
* Robustness means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.
* Safety-critical applications tend to favor correctness to robustness.
* Consumer applications tend to favor robustness to correctness. Any result whatsoever is usually better than the software shutting down.

# 8.4. Exceptions

Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it.

If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception, essentially throwing up its hands and yelling, "I don't know what to do about this—I sure hope somebody else knows how to handle it!"

Exceptions have an attribute in common with inheritance: used judiciously, they can reduce complexity. Used imprudently, they can make code almost impossible to follow:

Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code. — Andy Hunt Dave Thomas

* The overriding benefit of exceptions is their ability to signal error conditions in such a way that they cannot be ignored (Meyers 1996).
* Throw an exception only for conditions that are truly exceptional.

Exceptions should be reserved for conditions that are truly exceptional—in other words, for conditions that cannot be addressed by other coding practices: for events that should never occur.

* Exceptions represent a tradeoff between a powerful way to handle unexpected conditions on the one hand and increased complexity on the other. Exceptions weaken encapsulation by requiring the code that calls a routine to know which exceptions might be thrown inside the code that's called.
  + If an error condition can be handled locally, handle it locally. Don't throw an uncaught exception in a section of code if you can handle the error locally.
  + Avoid throwing exceptions in constructors and destructors unless you catch them in the same place.
  + The exceptions thrown are part of the routine interface, just like specific data types are.

Example 8-7.

class Employee { ... public TaxId GetTaxId() throws EOFException { <-- 1 ... } ... } (1)Here is the declaration of the exception that's at an

Example 8-7. class Employee { ... public TaxId GetTaxId() throws EOFException { <-- 1 ... } ... } (1)Here is the declaration of the exception that's at an inconsistent level of abstraction.

Example 8-8. Good Java Example of a Class that Throws an Exception at a Consistent Level of Abstraction class Employee { ... public TaxId GetTaxId() throws EmployeeDataNotAvailable { <-- 1 ... } ... }

* Include in the exception message all information that led to the exception.

Every exception occurs in specific circumstances that are detected at the time the code throws the exception.

* Avoid empty catch blocks

Sometimes it's tempting to pass off an exception that you don't know what to do with,

* Know the exceptions your library code throws.

If you're working in a language that doesn't require a routine or class to define the exceptions it throws, be sure you know what exceptions are thrown by any library code you use.

Failing to catch an exception generated by library code will crash your program just as fast as failing to catch an exception you generated yourself. If the library code doesn't document the exceptions it throws, create prototyping code to exercise the libraries and flush out the exceptions.

* Consider building a centralized exception reporter.

One approach to ensuring consistency in exception handling is to use a centralized exception reporter.

* Handling errors with exceptions just because your language provides exception handling is a classic example of programming in a language rather than programming into a language.
* Finally, consider whether your program really needs to handle exceptions, period. As Bjarne Stroustrup points out, sometimes the best response to a serious run-time error is to release all acquired resources and abort. Let the user rerun the program with proper input (Stroustrup 1997).

## Built-in exceptions in Python:

<https://docs.python.org/3/library/exceptions.html>

# 8.5. Barricade Your Program to Contain the Damage Caused by Errors

Barricades are a damage-containment strategy.

The reason is similar to that for having isolated compartments in the hull of a ship. If the ship runs into an iceberg and pops open the hull, that compartment is shut off and the rest of the ship isn't affected.

* Figure 8-2. Defining some parts of the software that work with dirty data and some that work with clean data can be an effective way to relieve the majority of the code of the responsibility for checking for bad data
* This same approach can be used at the class level.
  + The class's public methods assume the data is unsafe, and they are responsible for checking the data and sanitizing it.
  + Once the data has been accepted by the class's public methods, the class's private methods can assume the data is safe.

## Relationship Between Barricades and Assertions

The use of barricades makes the distinction between assertions and error handling clean-cut.

* Routines that are outside the barricade should use error handling because it isn't safe to make any assumptions about the data.
* Routines inside the barricade should use assertions, because the data passed to them is supposed to be sanitized before it's passed across the barricade.
  + If one of the routines inside the barricade detects bad data, that's an error in the program rather than an error in the data.

# 8.6. Debugging Aids

Another key aspect of defensive programming is the use of debugging aids, which can be a powerful ally in quickly detecting errors.

## Don't Automatically Apply Production Constraints to the Development Version

A common programmer blind spot is the assumption that limitations of the production software apply to the development version.

* The production version has to run fast. The development version might be able to run slow.
* The production version has to be stingy with resources. The development version might be allowed to use resources extravagantly.

Be willing to trade speed and resource usage during development in exchange for built-in tools that can make development go more smoothly.

## Introduce Debugging Aids Early

The earlier you introduce debugging aids, the more they'll help.

Typically, you won't go to the effort of writing a debugging aid until after you've been bitten by a problem several times.

## Use Offensive Programming

Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running. Michael Howard and David LeBlanc refer to this approach as "offensive programming"

### Here are some ways you can program offensively:

#### A dead program normally does a lot less damage than a crippled one. — Andy Hunt Dave Thoma

#### Make sure asserts abort the program.

Don't allow programmers to get into the habit of just hitting the Enter key to bypass a known problem. Make the problem painful enough that it will be fixed.

#### Plan to Remove Debugging Aids

If you're writing code for your own use, it might be fine to leave all the debugging code in the program. If you're writing code for commercial use, the performance penalty in size and speed can be prohibitive. Plan to avoid shuffling debugging code in and out of a program. Here are several ways to do that:

* Use version-control tools and build tools like ant and make. Version-control tools can build different versions of a program from the same source files.

#### Determining How Much Defensive Programming to Leave in Production Code

One of the paradoxes of defensive programming is that during development, you'd like an error to be noticeable—you'd rather have it be obnoxious than risk overlooking it. But during production, you'd rather have the error be as unobtrusive as possible, to have the program recover or fail gracefully.